Dominator Tree Verification and Vertex-Disjoint Paths

Loukas Georgiadis[1] Robert E. Tarjan[1,2]

Abstract

We present a linear-time algorithm that given a flowgraph $G = (V, A, r)$ and a tree T, checks whether T is the dominator tree of G. Also we prove that there exist two spanning trees of G, T_1 and T_2, such that for any vertex v the paths from r to v in T_1 and T_2 intersect only at the vertices that dominate v. The proof is constructive and our algorithm can build the two spanning trees in linear time. Simpler versions of our two algorithms run in $O(m\alpha(m, n))$-time, where n is the number of vertices and m is the number of arcs in G. The existence of such two spanning trees implies that we can order the calculations of the iterative algorithm for finding dominators, proposed by Allen and Cocke [2], so that it builds the dominator tree in a single iteration.

1 Introduction

We consider a flowgraph $G = (V, A, r)$, which is a directed graph such that every vertex is reachable from a distinguished start vertex $r \in V$. Let n and m be, respectively, the number vertices and arcs in G. A vertex w *dominates* a vertex v if every path from r to v includes w. Let $dom(v)$ be the set of the vertices that dominate v. Obviously, r and v, the *trivial dominators* of v, are in $dom(v)$. For $v \neq r$, the *immediate dominator* of v, denoted by $d(v)$, is the unique vertex $w \neq v$ that dominates v and is dominated by all the vertices in $dom(v) - v$. The *(immediate) dominator tree* is a directed tree I rooted at r which is formed by the arcs $\{(d(v), v) \mid v \in V - r\}$. A vertex w dominates v if and only if w is an ancestor of v in I [1]. Thus I is a compact representation of the dominance relation.

Compilers use dominance information extensively during program analysis and optimization, for natural loop detection (which enables several optimizations), structural analysis [18], scheduling [19], and the computation of dependence graphs and static single-assignment forms [9]. Dominators are also used to identify pairs of equivalent line faults in VLSI circuits [4].

There is an $O(m\alpha(m, n))$-time algorithm [16] to compute dominators that has been used in many of these applications. Here α is a functional inverse of Ackermann's function, which is very slow-growing; for all practical purposes this algorithm runs in linear time. Indeed, it runs fast in practice [13], even though it has some conceptual complexities. There are even more complicated truly linear-time algorithms that run on random-access machines [3, 7] and on pointer machines [12].

The work described here was motivated by a question Steven Weeks (private communication, 1999) asked the second author: is there a simple way to verify the correctness of a dominator tree computed by such a complicated algorithm (or any algorithm)?

The relationship of verification to computation is highly problem-dependent. It is interesting, for example, to compare the situation for shortest path trees to that of minimum spanning trees. There is a very simple linear-time algorithm to verify a shortest path tree, but no linear-time algorithm to compute shortest path trees is known (for a comparison-based computation model). On the other hand, minimum spanning trees can be verified in linear time [10, 15, 6], but the known methods are complicated. By combining a linear-time verification algorithm with random sampling, one can actually find a minimum spanning tree in linear time [14].

In this paper we present a linear-time algorithm to verify a dominator tree. This algorithm is simpler than the known linear-time algorithms to find dominator trees. Also, an $O(m\alpha(m, n))$-time version of our algorithm is simpler than the $O(m\alpha(m, n))$-time algorithm for finding dominators: it requires only a standard set union data structure instead of a link-eval data structure [23]. Our work sheds light on the relationship between verification and computation of dominators, and we hope it will lead to a simpler linear-time algorithm to find dominators.

The second part of this paper deals with a related problem involving dominators. We give a linear-time algorithm for constructing two spanning trees T_1 and T_2 of G with the following *ancestor-dominance* property. For any vertex $v \neq r$ the paths from r to v in T_1 and T_2 intersect only at the dominators of v. The existence of such two spanning trees is an extension of a result of

[1]Department of Computer Science, Princeton University, Princeton NJ, 08540. E-mail: {lgeorgia,ret}@cs.princeton.edu. Research at Princeton University partially supported by the National Science Foundation, under the Aladdin Project, Grant No 112-0188-1234-12.

[2]Hewlett-Packard, Palo Alto CA.

Whitty [25] on vertex-disjoint paths in directed graphs. Whitty gave a constructive proof for the case where G has only trivial dominators (i.e., when $d(v) = r$ for all $v \neq r$); he claimed only a polynomial running time for the algorithm implied by his construction. Here we provide a surprisingly simple algorithm that constructs such two spanning trees in $O(m\alpha(m,n))$-time in the general case where G may have nontrivial dominators. Furthermore, by applying the techniques of Buchsbaum et al. [6], we can construct a linear-time version of our algorithm. (A more complicated linear-time algorithm is also implied by the work of Alstrup et al. [3]). Finally, we show that the existence of two spanning trees with the ancestor-dominance property implies an optimal ordering of the calculations in the iterative algorithm of Allen and Cocke [2], so that it builds the dominator tree in one iteration.

1.1 Notation. We assume that G is represented by adjacency lists. In particular, for each vertex $v \in V$ we have a list of its predecessors $pred(v) = \{u \mid (u,v) \in A\}$ and of its successors $succ(v) = \{u \mid (v,u) \in A\}$. A tree T is represented by parent pointers; $p_T(v)$ denotes the parent of v in T. The notation "$v \stackrel{*}{\rightarrow}_T u$" means that v is an ancestor of u in T, "$v \stackrel{+}{\rightarrow}_T u$" means that v is a proper ancestor of u in T, and "$v \rightarrow_T u$" means that $v = p_T(u)$. We omit the subscript when we refer to a depth-first search (DFS) tree D of G. We denote the subtree of T rooted at v by T_v. The tree path from u to v in T is denoted by $T(u,v)$. If T is a spanning tree of G, an arc $a = (u,v)$ in G is a *tree arc* (with respect to T) if $a \in T$, a *forward arc* if $u \stackrel{+}{\rightarrow}_T v$ and $a \notin T$, a *back arc* if $v \stackrel{+}{\rightarrow}_T u$, and a *cross arc* otherwise (u and v are unrelated in T). Finally, for any subset $U \subseteq V$, $\mathrm{NCA}(T,U)$ denotes the nearest common ancestor of U in T. If $T = D$ then we use the notation $\nu(u,v)$ to stand for $\mathrm{NCA}(D,\{u,v\})$.

2 Dominator Tree Verification

2.1 Necessary Condition. It is known that the proposed dominator tree T must satisfy the following condition [17]:

$$(2.1) \quad p_T(w) = \mathrm{NCA}(T, pred(w)),\ \forall\ w \in V - r.$$

This condition is not sufficient in general but it is sufficient for acyclic graphs. Figure 1 shows a counter-example for a graph that contains a cycle. Nonetheless, Condition (2.1) provides some useful information about the location of the dominators of each vertex, as we show in the next lemma.

LEMMA 2.1. *Let T be a tree that satisfies (2.1). Then for any $w \neq r$, $p_T(w)$ dominates w.*

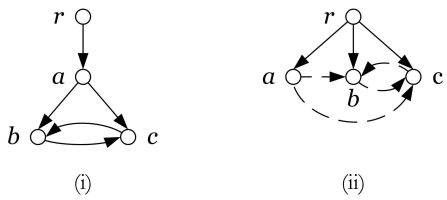

Figure 1: Condition (2.1) is not sufficient for graphs with cycles. (i) A flowgraph with 4 vertices. (ii) A tree T (solid arcs) that satisfies Condition (2.1) but is not the dominator tree of the graph, since vertices b and c are also dominated by a. The dashed arcs are non-tree arcs.

Proof. Let $u = p_T(w)$. Since T satisfies (2.1), for any z in $T_u - u$ we have $pred(z) \subseteq T_u$. Therefore any path from r to w must pass through u. Also since w is reachable from r there must be at least one such path, so u dominates w.

COROLLARY 2.1. *Let T be a tree that satisfies (2.1). For any $w \neq r$, $d(w) \in T_u$, where $u = p_T(w)$. Moreover, if $u \in pred(w)$ then $u = d(w)$.*

Hence, in a tree that satisfies Condition (2.1), the vertices on a path from the root to any vertex v are dominators of v, although they may comprise only a subset of $dom(v)$. Then if G contains only trivial dominators, Condition (2.1) is satisfied by a unique tree, where every $v \in V - r$ is a child of the root. The main idea of our verification algorithm is based on the previous observation and is the following: given G and T we construct for each vertex v that is not a leaf in T a subgraph $G_T(v)$, such that each $G_T(v)$ contains only trivial dominators if and only if T is the dominator tree of G. Thus we reduce the verification problem to the problem of testing if a graph contains only trivial dominators.

Our reduction is based on the idea of the *derived graph*, which we denote by G_T, and was introduced in [24]. We give the definition of the derived graph together with a linear-time procedure to construct it in the following section.

2.1.1 Derived graph. Let $G_T = (V, A_T, r)$. The set A_T contains an arc (u,v) in following two cases:

- $(u,v) \in A$ and $u = p_T(v)$.
- $u \neq v$, $p_T(u) = p_T(v)$ and $(u',v) \in A$, where $u \stackrel{*}{\rightarrow}_T u'$.

For any vertex w we define the *derived subgraph* $G_T(w)$ to be the subgraph of G_T induced by w and its children

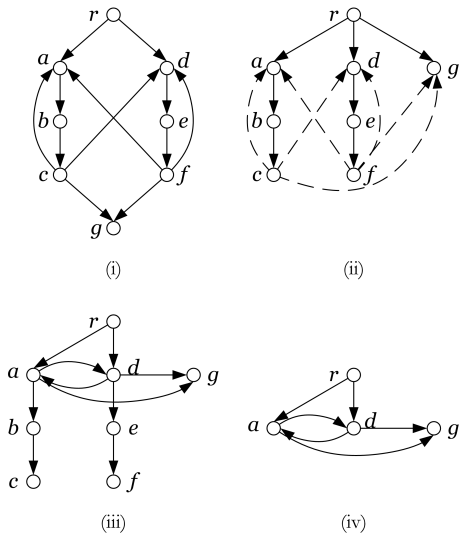

Figure 2: (i) The flowgraph $G = (V, A, r)$. (ii) The proposed dominator tree T. In this example $T = I$. The dashed arcs are non-tree arcs. (iii) The derived flowgraph $G_T = (V, A_T, r)$. The arcs (c, a) and (f, d) of G are eliminated. The arcs (c, d) and (c, g) of G correspond to (a, d) and (a, g) in G_T. The arcs (f, a) and (f, g) of G correspond to (d, a) and (d, g) in G_T. (iv) The derived subgraph $G_T(r)$.

in T. Figure 2 gives an example of these definitions. We have two kinds of arcs in G_T; arcs that lead from a parent to a child in T and arcs that lead from a vertex to one of its siblings in T. Each arc in G_T may correspond to several arcs in G. Any arc $(u, v) \in A$ such that $v \xrightarrow{*}_T u$ is ignored; this arc does not contribute any dominance information since all the vertices on $T(v, u)$ should be dominated by v. Our definition also excludes any arc $(u, v) \in A$ such that $\text{NCA}(T, \{u, v\}) \neq p_T(v)$. Note that if such an arc exists then T does not satisfy the necessary Condition (2.1) and our algorithm exits reporting that the input tree is not the dominator tree of G.

The next lemma shows that it suffices to verify that T is the dominator tree of G_T.

LEMMA 2.2. *T is the dominator tree of G if and only if T is the dominator tree of G_T.*

We omit the proof in this conference version of our paper. Now we describe an algorithm that builds G_T after verifying that T satisfies Condition (2.1). First we note that an equivalent way to state Condition (2.1) is that for each $x \neq r$ the following two properties must hold:

(P1) For all z in $pred(x)$, $p_T(x) \xrightarrow{*}_T z$.

(P2) Either $p_T(x)$ is in $pred(x)$ or there exist two distinct children of $p_T(x)$ in T, y_1 and y_2, and predecessors of x, z_1 and z_2, such that $y_i \neq x$ and $y_i \xrightarrow{*}_T z_i$, where $i = 1, 2$.

We start by constructing for each vertex v that is not a leaf in T a list of its children in T, which we denote by $chd_T(v)$. Then we perform a preorder walk of T. The first time we visit a vertex v we assign to it a preorder number and the last time (after visiting all the vertices in T_v) we compute the size of the subtree rooted at v; to do so, we assign $size(v) = \sum_{u \in chd_T(v)} size(u) + 1$. For simplicity, we will refer to the vertices of T by their preorder numbers. Then $v < u$ means that v was visited before u during the preorder walk. Since the vertices of a subtree are assigned consecutive numbers, $v \xrightarrow{*}_T u$ if and only if $v \leq u \leq v + size(v) - 1$. So, Property (P1) can be tested in constant time per edge. In order to test Property (P2) we need some additional information, which can also be collected during the preorder walk of T. When we visit a vertex v, for each $u \in succ(v)$ such that $v \neq p_T(u) \equiv w$ and $u \not\xrightarrow{*}_T v$ we insert v at the end of a list $predom(w)$. This list stores the predecessors v of the vertices u for which we want to verify that w is their immediate dominator. Figure 3 illustrates these definitions. Notice that since we visit the vertices in preorder, $predom(w)$ is sorted in ascending order. Moreover the list that represents $chd_T(w) = \{w_1, w_2, \ldots, w_k\}$ is also sorted with respect to the preorder numbering if we visit the children of v in the order given by the list. For each vertex $v \in predom(w)$ we want to find the child of w that is an ancestor of v, denoted by $a_T(w, v)$. Since the lists that represent both $chd_T(w)$ and $predom(w)$ are sorted in ascending order, we can find each $a_T(w, v)$ in time $O(|predom(w)| + |chd_T(w)|)$. Clearly $a_T(w, v)$ is the vertex w_i in the list $chd_T(w)$ that satisfies $w_i \leq v$ and $w_{i+1} > v$ if $1 \leq i < k$, or $w_k \leq v \leq w + size(w) - 1$. (If v does not satisfy any of these inequalities then Property (P1) does not hold.) Thus, we are essentially merging the two sorted lists that represent $chd_T(w)$ and $predom(w)$. Furthermore, a vertex v can be inserted in at most $|succ(v)|$ lists $predom$. Hence, the total time that will take us to compute $a_T(w, v)$ for all $w \in T$ and all $v \in predom(w)$ is proportional to $\sum_{w \in T}(|predom(w)| + |chd_T(w)|) \leq |A| + |V|$. After calculating $a_T(w, v)$ for each $v \in pred(u)$ such that $w = p_T(u)$, testing Property (P2) takes linear time.

As we mentioned earlier, if a test for (P1) or (P2) fails for some vertex, then our algorithm reports that $T \neq I$ and terminates. Otherwise, it uses the $a_T(w, v)$ information to construct the derived graph. The arcs $(p_T(v), v)$ in A are copied to A_T. For any other arc (u, v) in A, we include in A_T the corresponding arc

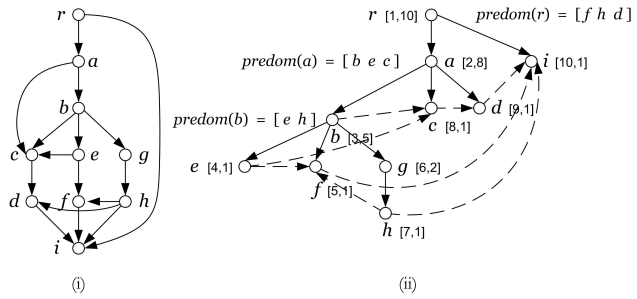

Figure 3: (i) A flowgraph with 10 vertices. (ii) The proposed dominators tree T. In this example $T = I$. The dashed arcs are non-tree arcs. The values inside the brackets correspond to the preorder number of a vertex and the size of its subtree. Also the nonempty lists *predom* are shown.

(z, v) where $z = a_T(p_T(v), u)$. Note that if we store the $a_T(w, v)$ values as we compute them in a linked list associated with v (which will be the list of predecessors of v in G_T), then these values will be sorted in ascending order because the vertices are visited in preorder in T. This fact enables us to avoid introducing multiple arcs in G_T.

2.2 Acyclic graphs. As we mentioned earlier, Condition (2.1) is necessary and sufficient for acyclic graphs. Hence, the verification procedure can accept T if it successfully completes the tests for (P1) and (P2) at each vertex.

2.3 Reducible graphs. If G is reducible [21] then for every back arc (u, v) with respect to a fixed depth-first search tree D of G, we have that v dominates u. Such arcs do not contribute any dominance information, and hence can be removed. The resulting graph is acyclic and has the same dominators as G, so we can apply the verification procedure we use for acyclic graphs.

2.4 General graphs. In the general case, for each derived sub-flowgraph $G_T(w) = (V_T(w), A_T(w), w)$, we have to check if all the vertices in $G_T(w)$ have only trivial dominators. If this is true then Corollary 2.1 implies that $d_{G_T}(v) = w$ for all $v \in V_T(w) - w$, and by Lemma 2.2 we have $d(v) = w$. In the next section we see how to verify in linear time that a graph has only trivial dominators.

2.4.1 Verifying Trivial Dominators. We will describe a subroutine that given a flowgraph $G = (V, E, r)$ checks whether r is the immediate dominator of every vertex $v \in V - r$. Initially we perform a DFS on G, which produces a DFS tree D and a preorder numbering for the vertices of G. We refer to the vertices by their preorder numbers in D, so $v < u$ means that v was visited before u during the DFS. Our verification procedure is based on the next simple observation.

LEMMA 2.3. *Let v be a vertex such that $d(v) \neq r$. Then there exists a vertex u such that $d(u) = p(u) \neq r$.*

Proof. Let u be the vertex that satisfies $d(v) \rightarrow u \stackrel{*}{\rightarrow} v$. Then we must have $d(u) = d(v)$. Otherwise there exists a path from r to u that avoids $d(v)$, which concatenated with the path $D(u, v)$ gives a path from r to v that avoids $d(v)$.

As the previous lemma implies, in order to verify that G has no nontrivial dominators it suffices to verify that there does not exist any vertex $w \in V - r$ that is dominated by $p(w) \neq r$. We can do so by computing for each vertex w the maximal strongly connected component $S(G, w)$ in G that contains only descendants of w in D. Formally, let $C(G, w)$ be the set of vertices z such that (z, w) is a cycle arc entering w. By convention $w \in C(G, w)$. We define

$$S(G, w) = \{v \mid w \stackrel{*}{\rightarrow} v \text{ and } \exists\ z \in C(G, w) \text{ such that } \exists \text{ path from } v \text{ to } z \text{ containing only descendants of } w\}.$$

Note that if $C(G, w) = \emptyset$ then $S(G, w) = \{w\}$. In order to compute and represent the $S(G, v)$ sets efficiently we define the operation $collapse(S, v)$, that collapses a set $S \subseteq V$ to a vertex $v \notin S$, as follows. For each $x \in S$ and $w \notin S \cup \{v\}$, if (x, w) exists we replace it with (v, w). Similarly, if (w, x) exists we replace it with (w, v). Finally we remove S. Let $G(n) = G$ and $I(n) = S(G, n) = \{n\}$. For $k = n-1, \ldots, 1$, we compute $I(k) = S(G(k+1), k)$ and $G(k) = collapse(I(k) - k, k)$. Notice that the sets $I(k) - k$ partition the set of vertices $\{i \mid 2 \leq i \leq n\}$. The sets $I(k)$ are called the *intervals* of G and can be found by computing $\nu(u, v)$ for all $(u, v) \in A$ and using disjoint-set union operations [22]. A simple implementation of these computations runs in $O(m\alpha(m, n))$-time [23]. Using the results of [11] and [6] they can be performed in linear time.

LEMMA 2.4. *Let k be any vertex such that $p(k) \neq 1$. Then $d(k) \neq p(k)$ if and only if there exists $j \in I(k)$ such that j has a predecessor in $G(k+1)$ that is not dominated by $p(k)$ in $G(k+1)$.*

Proof. Obviously $d(k) \neq p(k)$ if and only if there exists $j \in S(G, k)$ such that j has a predecessor that is not

TVerify()

for $k = n, n-1, \ldots, 2$ **do**
  $label(k) \leftarrow \min\{label((j,k)) \mid j \in pred(k)\}$
  $label(k) \leftarrow \min\{label(j) \mid j \in I(k)\}$
  if $label(k) = p(k)$ **then**
    return FALSE
  endif
done
return TRUE

Figure 4: Procedure TVerify returns FALSE if it finds a vertex that is dominated by its parent. It assumes that each arc $a = (u, v)$ is labeled so that $label(a) = v$ if it is a back arc, $label(a) = p(\nu(u, v))$ if it is a cross arc, and $label(a) = u$ if it is a tree or a forward arc.

dominated by $p(k)$. Note that if $i \in S(G, j)$ then $i \in S(G, k)$. Thus, $S(G, k) = \bigcup_{j \in I(k)} S(G, j)$ and the Lemma follows from the definition of $collapse(S, v)$.

Figure 4 gives the outline of our algorithm. It assumes that we have already computed the intervals of G and that for each arc $a = (u, v)$ we have computed a label such that

$$label(a) = \begin{cases} v, & a \text{ is a back arc} \\ u, & a \text{ is a forward or a tree arc} \\ p(\nu(u,v)), & a \text{ is a cross arc} \end{cases}.$$

The algorithm processes the vertices in reverse preorder. For each vertex k it computes $label(k)$, which is the minimum of the labels of the incoming arcs of k and of the labels of the vertices in $I(k)$. It is important to note that this is equivalent to labeling vertex k in $G(k)$ by setting $label(k)$ equal to the minimum label of all the arcs entering k in $G(k)$. If $label(k)$ equals $p(k)$ the algorithm exits and reports that k is dominated by $p(k)$. It is clear that algorithm TVerify runs in linear time given the intervals of G and the arc labels. Figure 5 gives an example of the execution of this algorithm.

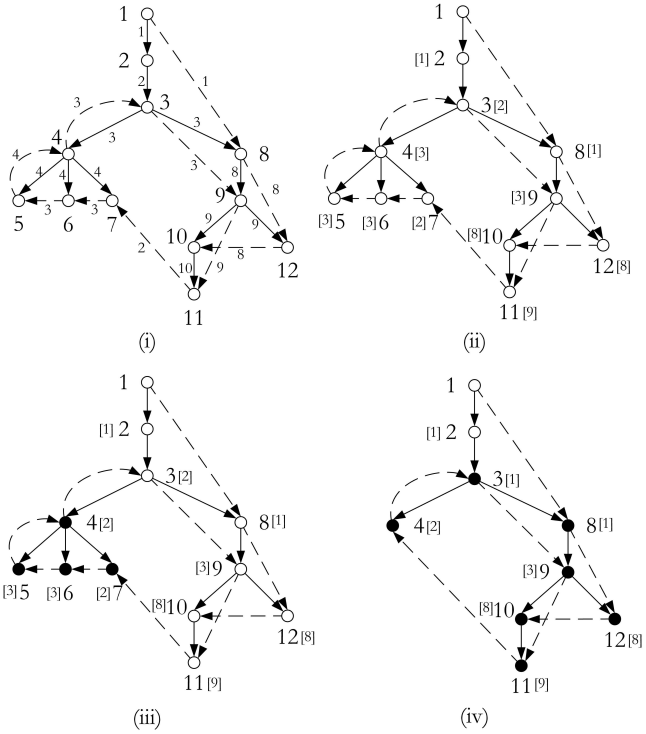

Figure 5: Example of the execution of TVerify. (i) The input graph with the arcs already labeled. The dashed arcs do not belong to the DFS tree. (ii) The graph after labeling each vertex by the minimum label of its incoming arcs. (For more clarity we present this step as being performed separately.) (iii) The situation when we process vertex 4. All the vertices with higher preorder numbers have already passed the test. We have $I(4) = \{4, 5, 6, 7\}$ (black nodes), so the new label of 4 is $2 < p(4)$. (iv) The situation when we process vertex 3. We have collapsed $I(4) - 4$ to 4 and now $I(3) = \{3, 4, 8, 9, 10, 11, 12\}$, so the new label of 3 is $1 < p(3)$.

LEMMA 2.5. *Algorithm TVerify is correct.*

Proof. We show by induction on k that if the algorithm does not return FALSE after processing k then k is not dominated by $j = p(k)$. The basis $k = n$ is straightforward: the only arcs that may enter n, excluding the tree arc, are forward arcs. Assuming that $j \neq 1$, $label(n) < j$ if and only if there exists a forward arc entering n. Suppose that the algorithm has correctly verified the dominators of the vertices $n, n-1, \ldots, k+1$. By Lemma 2.4 it suffices to show that for any $z \in I(k)$ we have $label(z) < j$ if and only if z is not dominated by j in $G(k+1)$. Clearly if no such z exists then $I(k)$ is dominated by j and the algorithm correctly reports failure. (Remember that for each vertex $z \in I(k) - k$ there can only exist tree, forward or cross arcs entering z in $G(k+1)$.) Now assume that there is such a z. If there exists an arc (x, z) such that $\nu(z, x) < j$ then by Lemma 2.4, k is not dominated by j in G and the algorithm sets $label(k) < j$. Next assume that $\nu(z, x) = j$. Let l be the sibling of k that is an ancestor of x in D. Since l passed the test we already know that it is not dominated by j, and therefore z is not dominated by j. The algorithm will set $label(k) < j$ and k correctly passes the test.

3 Two Spanning Trees

Let $G = (V, A, r)$ be a flowgraph. The paths $(v_0, v_1, \ldots, v_{k-1}, v_k)$ and $(u_0, u_1, \ldots, , v_{l-1}, u_l)$ are *vertex-disjoint* if $v_i \neq u_j$ for $1 \leq i \leq k-1$ and $1 \leq j \leq l-1$. The following theorem is a special case of a result shown by Whitty in [25].

THEOREM 3.1. [25] *Let $G = (V, A, r)$ be a flowgraph such that $d(v) = r$ for all $v \in V - r$. There exist two spanning trees of G, T_1 and T_2, such that for any $v \in V$ the paths $T_1(r, v)$ and $T_2(r, v)$ are vertex-disjoint.*

Whitty's proof is constructive and computes T_1 and T_2 in polynomial time. Anthony Wirth gave a simple construction for the case where G is acyclic [26]. His algorithm initially computes a topological order of the vertices and arranges the successor lists from deepest to shallowest successor, with respect to the topological order. Then with a second DFS it builds T_1, and afterwards it constructs the *residual* graph H by deleting the edges of T_1 that are not adjacent to the root. Finally the algorithm can pick *any* spanning tree of H to be T_2.

We can extend Theorem 3.1 to get the following result:

THEOREM 3.2. *Let $G = (V, A, r)$ be a flowgraph. There exist two spanning trees of G, T_1 and T_2, such that for any $v \in V$ the paths $T_1(r, v)$ and $T_2(r, v)$ intersect only at the vertices of $dom(v)$.*

This theorem can be proved by using the derived graph of Section 2.1.1 together with Theorem 3.1. We omit this proof from the conference version of our paper. Instead, we give a direct proof of Theorem 3.2, by providing an algorithm that computes two spanning trees that have the ancestor-dominance property for *any* flowgraph.

3.1 Construction of the Two Spanning Trees.

In this section we present a simple algorithm that constructs two spanning trees with the property of Theorem 3.2. Our algorithm uses the concept of *semidominators*, which was introduced by Lengauer and Tarjan [16].

3.1.1 Semidominators.

Let D be a depth-first search tree of G. We assign to each vertex a preoder number with respect to D and refer to the vertices by these numbers. An important property of DFS, which we will use repeatedly later, is stated in the next lemma.

LEMMA 3.1. [20] *If v and w are vertices of G such that $v \leq w$, then any path from v to w must contain a common ancestor of v and w in D.*

A path $P = (u = v_0, v_1, \ldots, v_{k-1}, v_k = v)$ is a *semidominator path* (abbreviated s-path) if $v_i > v$ for $1 \leq i \leq k-1$. The *semidominator* of vertex v is $s(v) = \min\{u \mid \text{there is an s-path from } u \text{ to } v\}$. For any vertex $w \neq r$, we have $d(w) \stackrel{*}{\rightarrow} s(w) \stackrel{+}{\rightarrow} w$ [16]. The next lemma describes some properties of the semidominators that will be useful in our construction.

LEMMA 3.2. [16] *Let $w \neq r$ and let u be a vertex for which $s(u)$ is minimum among vertices u satisfying $s(w) \stackrel{+}{\rightarrow} u \stackrel{*}{\rightarrow} w$. Then $s(u) \leq s(w)$ and $d(u) = d(w)$. Moreover, if $s(u) = s(w)$ then $d(w) = s(w)$.*

For any $v \in V - r$, we define $t(v)$ to be a predecessor of v that belongs to an s-path from $s(v)$ to v. Such vertices can be found easily during the computation of the semidominators. Using the previous lemma, we can show the following (the proof is omitted):

LEMMA 3.3. *Let w be a vertex such that $s(w)$ is not a predecessor of w. Then there exists a vertex $h(w)$ such that $\nu(w, t(w)) \stackrel{+}{\rightarrow} h(w) \stackrel{*}{\rightarrow} t(w)$ and $s(h(w)) = s(w)$.*

Now we consider the flowgraph $G_{\min} = (V, A_{\min}, r)$, where $A_{\min}$ is the multiset $\{(p(v), v) \mid v \in V - r\} \cup \{(t(v), v) \mid v \in V - r\}$. Notice that $A_{\min}$ contains two copies of $(p(v), v)$ for each v that satisfies $t(v) = p(v)$. Also every vertex has in-degree 2 except for r, which has in-degree 0. We can show that G and $G_{\min}$ have the same dominators. Then, since any spanning tree of $G_{\min}$ is also a spanning tree of G, it suffices to construct T_1 and T_2 for $G_{\min}$. Henceforth we will assume $G_{\min} \equiv G$.

3.2 Algorithm.

For any $v \neq r$, we define $\Sigma(v) = \{x \mid s(v) \stackrel{+}{\rightarrow} x \stackrel{*}{\rightarrow} v\}$ and $E(v) = \{x \mid x \in \Sigma(v) \text{ and } s(x) \leq s(y) \text{ for all } y \in \Sigma(v)\}$. Also we define $e(v)$ to be the minimum vertex in $E(v)$. Note that by Lemma 3.2 we have $s(e(v)) = s(v)$ if and only if $s(v) = d(v)$.

Figure 6 shows the method we use to build the two spanning trees; a blue tree B and a red tree R. We call a vertex $v \neq r$ *blue* if $(t(v), v) \in B$ and *red* otherwise. An equivalent way to state the construction is: color v blue if $s(e(v)) = s(v)$ or $e(v)$ is red; color v red otherwise. Figure 7 gives an example of the construction. Even though this construction is simple (given the function s), verifying its correctness is intricate. We prove first that B and R are acyclic and hence are trees, and second that corresponding paths in B and R are disjoint. Both of these steps require some preliminary ground work.

3.3 Properties of B and R.

We begin with two lemmas that relate the colors of certain vertices.

STrees()

for $k = 2, 3, \ldots, n$ **do**
  if $s(e(k)) = s(k)$ **or** $(t(e(k)), e(k)) \in R$ **then**
    { add $(t(k), k)$ to B;
      add $(p(k), k)$ to R }
  else
    { add $(t(k), k)$ to R;
      add $(p(k), k)$ to B }
  endif
done

Figure 6: Procedure STrees constructs two spanning trees B and R such that for any $v \neq r$ the paths $B(r, v)$ and $R(r, v)$ intersect only at the vertices in $dom(v)$.

LEMMA 3.4. *Let v and w be vertices such that $v \xrightarrow{*} w$, $s(v) = s(w)$, and $s(x) \geq s(v)$ for all x such that $v \xrightarrow{+} x \xrightarrow{+} w$. Then v and w are the same color.*

Proof. The hypotheses of the Lemma and the definition of the function e imply that $e(v) = e(w)$. This and $s(v) = s(w)$ imply that v and w are the same color.

Lemma 3.4 implies that B and R remain the same if, for each $v \neq r$, we let $e(v)$ be *any* vertex $x \in E(v)$.

LEMMA 3.5. *Let x, y and z be vertices that satisfy the following conditions:*

(i) $x \xrightarrow{+} y \xrightarrow{+} z$ *or* $x \xrightarrow{+} z \xrightarrow{+} y$,

(ii) $s(x) < s(y) < s(z) < x$,

(iii) $s(w) \geq s(x)$, *for all w such that* $\min\{y, z\} \xrightarrow{*} w \xrightarrow{*} \max\{y, z\}$, *and*

(iv) x *and* z *are the same color.*

Then y is the same color as x and z.

Proof. Suppose that the Lemma is false. Choose three vertices x, y and z that violate the Lemma and such that x is minimum. Since $x \in \Sigma(z)$, $s(e(z)) \leq s(x) < s(z)$. So $e(z)$ and z have different colors, which implies that $e(z) \neq x$. If $s(e(z)) = s(x)$ then Lemma 3.4 implies that $e(z)$ and x have the same color, a contradiction. Thus $s(e(z)) < s(x)$ and (iii) implies that $e(z) \xrightarrow{+} \min\{y, z\}$. Since y and z have different colors and $e(z) \in \Sigma(y)$, it must be the case that $e(y) \xrightarrow{*} s(z)$ and $s(e(y)) < s(e(z))$. But then $e(y), e(z)$ and x violate the Lemma, contradicting the choice of x.

Next we prove that B and R are trees.

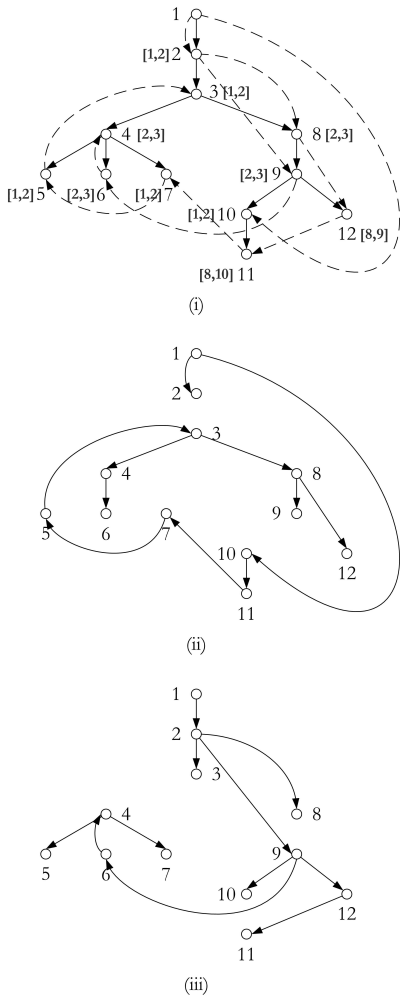

Figure 7: Example of the execution of STrees. (i) The input graph with the vertices already numbered with respect to a DFS tree D (solid arcs). The values inside the brackets correspond to $s(v)$ and $e(v)$. (ii) The blue spanning tree B. (iii) The red spanning tree R.

LEMMA 3.6. *Neither B nor R contains a cycle.*

Proof. We shall derive a contradiction to the assumption that B contains a cycle; the same argument applies to R. Given a cycle in B, let v be the minimum vertex on the cycle. Then $v \neq r$ (since r contains no incoming arcs), and $s(v) < v$. Also, by Lemma 3.1, $(t(v), v)$ is a cycle arc, v is blue and all vertices on the cycle are descendants of v. Let w be the first vertex after v on the cycle such that w is blue and $s(w) < v$. (If v is the only such vertex on the cycle, then $w = v$.) Then $v \xrightarrow{+} t(w)$, since $v = t(w)$ would imply $s(w) = t(w) = v$, which contradicts $s(w) < v$. By Lemma 3.3, $u = h(w)$ satisfies $v \xrightarrow{+} u \xrightarrow{*} t(w)$ and $s(u) = s(w) < v$. We claim

that u is blue. Indeed, v is a candidate for both $e(u)$ and $e(w)$, which means that $\max\{s(e(u)), s(e(w))\} \leq s(v)$. Also, the definition of $s(v)$ implies that $s(x) \geq s(v)$ for any vertex x that is a descendant of v and an ancestor of either u or w. It follows that $e(u) = e(w)$ is an ancestor of v, and hence u is the same color as w; namely, blue. Let z be the vertex on the cycle such that all vertices on the cycle from z to $t(w)$ (inclusive) are descendants of u, but the predecessor y of z on the cycle is not a descendant of u. There must be such a vertex z, since v is on the cycle but not a descendant of u. Furthermore, starting from v, z precedes w on the cycle. It cannot be the case that $z = u$, for then u would be on the cycle after v but before w, contradicting the choice of w. Thus $u \stackrel{+}{\rightarrow} z$, z is blue, and $t(z) = y$ is not a descendant of u. But $t(z)$ not a descendant of u implies $s(z) < u$. It cannot be the case that $s(z) < v$, for this would contradict the choice of w. Thus $v \stackrel{*}{\rightarrow} s(z) \stackrel{+}{\rightarrow} u \stackrel{+}{\rightarrow} z$. Then $s(e(z)) \leq s(u) < v < z$ and $e(z)$ is red. Therefore, by Lemma 3.4 and the fact $e(z) \in S(v)$, we have $s(v) < s(e(z)) < s(u)$. By Lemma 3.5, $e(z)$ must be blue, contradicting the fact that $e(z)$ is red.

To prove disjointness of paths in B and R, we need one technical lemma in addition to Lemmas 3.4 and 3.5. This lemma requires some more definitions. For each vertex $v \neq r$, define $\widehat{s}(v)$ as follows: if v is blue (red) $\widehat{s}(v)$ is the nearest ancestor x of v in B (R) such that $x < v$. By Lemma 3.6, B and R are trees rooted at r, which implies that $\widehat{s}(v)$ is well-defined. By Lemma 3.1 and the definition of function s, $s(v) \stackrel{*}{\rightarrow} \widehat{s}(v) \stackrel{+}{\rightarrow} v$. Let $\widehat{\Sigma}(v) = \{x \mid \widehat{s}(v) \stackrel{+}{\rightarrow} x \stackrel{*}{\rightarrow} v\}$ and $\widehat{E}(v) = \{x \mid x \in \widehat{\Sigma}(v) \text{ and } s(x) \leq s(y) \text{ for all } y \in \widehat{\Sigma}(v)\}$. Let $\widehat{e}(v)$ be the minimum vertex in $\widehat{E}(v)$.

LEMMA 3.7. *For any vertex $v \neq r$, either $\widehat{s}(v) = s(v)$, or $s(\widehat{e}(v)) < s(v)$ and $\widehat{e}(v)$ and v are different colors.*

Proof. The proof is by induction on v in decreasing order. If $\widehat{s}(v) = s(v)$ then the Lemma holds for v. Thus suppose $s(v) < \widehat{s}(v)$. If $(t(v), v)$ is a forward arc, $\widehat{s}(v) = t(v) = s(v)$, a contradiction. Thus $(t(v), v)$ is a cycle arc or a cross arc. Let $\nu = \nu(t(v), v)$ and $w = h(v)$. By Lemma 3.3, $\nu \stackrel{+}{\rightarrow} w \stackrel{*}{\rightarrow} t(v)$ and $s(w) = s(v)$. Lemma 3.1 implies that $\widehat{s}(v) \stackrel{*}{\rightarrow} \nu$. If $\nu \stackrel{+}{\rightarrow} e(v)$ and $s(e(v)) < s(v)$, then $\widehat{e}(v) = e(v)$ and the Lemma holds for v by the construction of B and R. Thus suppose $e(v) \stackrel{*}{\rightarrow} \nu$ or $s(e(v)) = s(v)$. We claim that in either case v and w are the same color. The definition of $s(v)$ implies that $s(x) \geq s(v)$ for all x such that $\nu \stackrel{+}{\rightarrow} x \stackrel{*}{\rightarrow} w$. If $e(v) \stackrel{*}{\rightarrow} \nu$, then $e(v) = e(w)$ and v and w are the same color. If $s(e(v)) = s(v)$, then $s(x) \geq s(v)$ for all x such that $s(v) \stackrel{+}{\rightarrow} x \stackrel{*}{\rightarrow} \nu$, which means that $s(e(w)) = s(v) = s(w)$, and again v and w are the same color.

Suppose v and w are both blue; the symmetric argument applies if they are both red. Let z be the nearest ancestor of $t(v)$ in B such that the parent y of z in B is not a descendant of w. Vertex z is blue, since w is blue and if $z \neq w$ the blue arc (y, z) entering z cannot be a tree arc. Also $z > v$ (follows from Lemma 3.1), so the Lemma holds for z by the induction hypothesis. The definition of $s(v)$ implies that $s(z) \geq s(v)$, because z is on $B(\widehat{s}(v), v)$. If $s(z) = s(v)$ and $\widehat{s}(z) = s(z)$, then $\widehat{s}(v) = \widehat{s}(z) = s(v)$, and the Lemma is true. Thus suppose $s(z) > s(v)$ or $\widehat{s}(z) > s(z)$. We claim that $s(\widehat{e}(z)) < s(v)$ and $\widehat{e}(z)$ is red. If $s(z) = s(v)$ and $\widehat{s}(z) > s(z)$, the claim follows since the Lemma holds for z. Suppose $s(z) > s(v)$. Then $z \neq w$ and $\widehat{s}(v) \stackrel{+}{\rightarrow} w \stackrel{+}{\rightarrow} z$ by the existence of the arc (y, z). Thus $s(e(z)) \leq s(\widehat{e}(z)) \leq s(w) = s(v) < s(z)$, and by the construction of B and R, $e(z)$ is red. If $\widehat{s}(z) > s(z)$, $\widehat{e}(z)$ is red since the Lemma holds for z. Also $s(\widehat{e}(z)) < s(v)$, since $s(\widehat{e}(z)) = s(v) = s(w)$ implies $\widehat{e}(z)$ is blue by Lemma 3.4. Since $s(\widehat{e}(z)) < s(v)$, $\widehat{e}(z) \stackrel{*}{\rightarrow} \nu$, which implies $\widehat{s}(z) \stackrel{*}{\rightarrow} \nu$ and $\widehat{s}(v) = \widehat{s}(z)$. Either $\widehat{e}(v) = \widehat{e}(z)$, in which case the Lemma holds for v, or $\nu \stackrel{+}{\rightarrow} \widehat{e}(v)$ and $s(\widehat{e}(v)) < s(\widehat{e}(z))$. In this case if $s(e(v)) = s(\widehat{e}(v))$ then $\widehat{e}(v)$ is red by Lemma 3.4 and the Lemma holds for v; if $s(e(v)) < s(\widehat{e}(v))$ then $\widehat{e}(v)$ is red by Lemma 3.5 applied to $e(v)$, $\widehat{e}(v)$, and $\widehat{e}(z)$, and the Lemma holds for v.

3.4 Vertex-disjointness. Now we are ready to prove vertex-disjointness. First we argue that it suffices to prove that for each v the paths $B(d, v)$ and $R(d, v)$ contain no common vertex other than d and v, where $d = d(v)$. Indeed, let $dom(v) = \{d_1 = r, d_2, \ldots, d_{k-1} = d, d_k = v\}$, where $d_i = d(d_{i+1})$ for $1 \leq i \leq k-1$. Then, d_i is an ancestor of d_{i+1} in both B and R, so the dominators of v appear in the same order in $B(r, v)$ and $R(r, v)$. Suppose now that $B(r, v)$ and $R(r, v)$ intersect at a vertex x such that d_B is the closest dominator of v in B that is an ancestor of x, and similarly d_R is the closest dominator of v in R that is an ancestor of x. Without loss of generality assume that d_B is an ancestor of d_R (in both B and R). If $d_B \neq d_R$ then the path $B(r, x)$ followed by $R(x, v)$ avoids d_R which is a contradiction. Hence $d_B = d_R = d_i$, and the paths $B(d_i, d_{i+1})$ and $R(d_i, d_{i+1})$ intersect at x.

LEMMA 3.8. *Let v be any vertex other than r, and let $d = d(v)$. Then the paths $B(d, v)$ and $R(d, v)$ contain no common vertex other than d and v.*

Proof. Suppose to the contrary that $B(d, v)$ and $R(d, v)$ both contain a vertex $w \notin \{d, v\}$. Let x_B and x_R be the

minimum vertices on $B(w,v)$ and $R(w,v)$ respectively. Neither $B(w,v)$ nor $R(w,v)$ contains d, since $B(d,v)$ and $R(d,v)$ are simple paths. In particular $d \notin \{x_B, x_R\}$. Assume $x_B \le x_R$; the symmetric argument applies if $x_R \le x_B$. We have that $d \xrightarrow{+} x_B$ and, by Lemma 3.1, $x_B \xrightarrow{*} x_R \xrightarrow{*} v$. If $x_B \neq w$ then x_B is blue since then it is entered by a blue nontree arc. Similarly if $x_R \neq w$ then x_R is red. We have $x_B \xrightarrow{+} v$, since $x_B = v$ implies $x_R = v$ and v is both blue and red since $w \neq v$, a contradiction.

Let u be a vertex of minimum $s(u)$ such that $x_B \xrightarrow{+} u \xrightarrow{*} v$. Since x_B does not dominate v, Lemma 3.2 implies $s(u) \xrightarrow{+} x_B$. By Lemma 3.7, either $\widehat{s}(u) = s(u)$, in which case $\widehat{s}(u) \xrightarrow{+} x_B$, or $s(\widehat{e}(u)) < s(u)$, which implies by the choice of u that $\widehat{e}(u) \xrightarrow{*} x_B$, and again $\widehat{s}(u) \xrightarrow{+} x_B$. We claim that u is red. Suppose to the contrary that u is blue. Then u cannot be on $B(x_B, v)$; if it were, $\widehat{s}(u)$ would be on $B(x_B, v)$, since $x_B \xrightarrow{+} u$; but every vertex on $B(x_B, v)$ is no less than x_B, contradicting $\widehat{s}(u) \xrightarrow{+} x_B$. Let x be the first vertex on $B(x_B, v)$ such that $u \xrightarrow{+} x \xrightarrow{*} v$. Then x is blue and $x_B \xrightarrow{*} \widehat{s}(x) \xrightarrow{+} u \xrightarrow{+} x \xrightarrow{*} v$. The definition of u implies $\widehat{e}(x) = u$. If $\widehat{s}(x) = s(x)$, then $e(x) = u$, and u is red by the construction of B and R since x is blue. If $\widehat{s}(x) > s(x)$, then $u = \widehat{e}(x)$ is red by Lemma 3.7 since x is blue.

Since u is red, $u \xrightarrow{*} x_R$, because if $x_R \xrightarrow{+} u$ an argument symmetric to that in the previous paragraph shows that u is blue. Since $x_B \xrightarrow{+} u \xrightarrow{*} x_R$, $x_B \neq w$, which implies that x_B is blue.

We claim that $s(u) \le \widehat{s}(x_B)$. Vertex w is on $B(\widehat{s}(x_B), x_B)$. Consider the path $B(\widehat{s}(x_B), w)$ followed by $R(w,v)$. This path avoids x_B. Let y be the first vertex along this path such that $x_B \xrightarrow{+} y \xrightarrow{*} v$. The part of the path from $\widehat{s}(x_B)$ to y is an s-path for y. Hence $s(y) \le \widehat{s}(x_B)$. By the choice of u, $s(u) \le s(y) \le \widehat{s}(x_B)$.

Next we claim that $s(u) < s(x_B)$. If $s(u) = s(x_B)$, u and x_B are the same color by Lemma 3.4, a contradiction. If $s(u) > s(x_B)$, then since $s(u) \le \widehat{s}(x_B)$, Lemma 3.7 gives $s(\widehat{e}(x_B)) < s(x_B)$ and $\widehat{e}(x_B)$ is red. But then $\widehat{e}(x_B), x_B$ and u violate Lemma 3.5.

Now we claim that w is not a descendant of u. Suppose to the contrary that $u \xrightarrow{*} w$. Then the path from $s(u)$ to x_B consisting of the s-path from $s(u)$ to u, followed by the tree path to w, followed by $B(w, x_B)$ is an s-path for x_B, giving $s(u) \ge s(x_B)$, a contradiction.

Since w is not a descendant of u, $w \neq x_R$. Hence x_R is red. Furthermore it cannot be the case that $u \xrightarrow{*} \widehat{s}(x_R)$, since w is on $R(\widehat{s}(x_R), x_R)$, which implies $\widehat{s}(x_R) \xrightarrow{*} w$. Thus $\widehat{s}(x_R) \xrightarrow{+} u$, and $s(\widehat{e}(x_R)) \le s(u)$. Also the path $R(\widehat{s}(x_R), w)$ followed by $B(w, x_B)$ is an s-path for x_B, which implies $s(x_B) \le \widehat{s}(x_R)$. If $s(x_R) = \widehat{s}(x_R)$ then $s(x_B) \le s(x_R)$, and since $s(u) < s(x_B)$, we have $s(e(x_R)) \le s(u) < s(x_R)$. So, by the construction of B and R, $e(x_R)$ is blue and Lemma 3.4 gives $s(e(x_R)) < s(u)$, which implies $e(x_R) \xrightarrow{+} x_B$. But then Lemma 3.5 for $e(x_R), u$ and x_B implies that u is blue, a contradiction. Hence $s(x_R) < \widehat{s}(x_R)$. Then $\widehat{e}(x_R)$ is blue by Lemma 3.7, and Lemma 3.4 implies $s(\widehat{e}(x_R)) < s(u)$. But then Lemma 3.5 for $\widehat{e}(x_R), u$ and x_B implies that u is blue, again a contradiction.

3.5 Running Time. The Lengauer-Tarjan algorithm computes all $s(v)$ and $e(v)$ in $O(m\alpha(m,n))$-time, which implies that we can implement algorithm STrees with the same time complexity. We can get a linear-time algorithm by using the techniques of Buchsbaum et al. [6]. (The details will appear separately [5].) Finally we note that the result of Alstrup et al. [3] implies a linear-time implementation of our algorithm, but it is much more complicated.

3.6 Iterative Algorithm. Allen and Cocke [2] showed that the dominance relation of a flowgraph is the maximal fixed point of the following set of equations:

$$dom'(v) = \Big(\bigcap_{u \in pred(v)} dom'(u) \Big) \cup \{v\}, \ \forall\ v \in V - r,$$

where $dom'(r) = \{r\}$. This fixed point can be computed iteratively. The iterative algorithm either initializes $dom'(v) \leftarrow V$ for all $v \neq r$, or excludes uninitialized $dom'(u)$ sets from the intersections in the above equations. Cooper et al. [8] observed that the algorithm does not need to keep each dom' set explicitly; it suffices to maintain the transitive reduction of the dominance relation, which is a tree T. Then, the intersection of $dom'(u)$ and $dom'(w)$ consists of the vertices on the path from $\mathrm{NCA}(T, \{u, w\})$ to r. Since the dominator tree must satisfy Condition (2.1), the iterative algorithm can be viewed as a process that modifies a tree T successively until (2.1) holds for all $w \in V - r$. The tree changes as the algorithm processes the arcs of the flowgraph. An arc (u,v), such that $u \in T$, is processed by computing $x = \mathrm{NCA}(T, \{u, v\})$ and setting $p_T(v) \leftarrow x$ if x is an ancestor of $p_T(v)$ in T. (If $v \notin T$ then we simply set $p_T(v) = u$.) Reference [13] states that there is an optimal ordering of the computations for the iterative algorithm. This is formalized in the following lemma.

LEMMA 3.9. *There exists an ordering σ of the arcs of G such that if the iterative algorithm processes the arcs according to σ, then it will construct the dominator tree of G in a single iteration.*

The proof is given in the full version of this paper. We note, however, that it is straightforward after defining σ, which we do next. Let T_1 and T_2 be two spanning trees of G that satisfy Theorem 3.2. We construct σ by concatenating a list σ_1 of the arcs of T_1 with a list σ_2 of the arcs of T_2. The arcs in σ_i are sorted lexicographically in ascending order with respect to a preorder numbering of T_i $(i = 1, 2)$. For acyclic graphs we can construct an optimal ordering after a single DFS; it suffices to order the arcs with respect to a topological order. This follows immediately from the fact that Condition (2.1) is necessary and sufficient for acyclic graphs. Experimental results show that the topological order is a good choice in practice [8, 13]. An interesting question is whether there are similar results that can speed up iterative algorithms for more general dataflow problems.

Acknowledgements. We thank Hal Gabow for pointing out [25] to us. We also would like to thank an anonymous referee for some helpful comments.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] F. E. Allen and J. Cocke. Graph theoretic constructs for program control flow analysis. Technical Report IBM Res. Rep. RC 3923, IBM T.J. Watson Research Center, 1972.

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

[4] 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.

[5] A. L. Buchsbaum, L. Georgiadis, H. Kaplan, A. Rogers, R. E. Tarjan, and J. R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, mst verification, and dominators. In preparation.

[6] A. L. Buchsbaum, H. Kaplan, A. Rogers, and J. R. Westbrook. Linear-time pointer-machine algorithms for least common ancestors, MST verification, and dominators. In *Proc. 30th ACM Symp. on Theory of Computing*, pages 279–88, 1998.

[7] 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 to appear.

[8] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Available online at `http://www.cs.rice.edu/~keith/EMBED/dom.pdf`.

[9] 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.

[10] B. Dixon, M. Rauch, and R. E. Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM Journal on Computing*, 21(6):1184–92, 1992.

[11] 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.

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

[13] L. Georgiadis, R. F. Werneck, R. E. Tarjan, S. Triantafyllis, and D. I. August. Finding dominators in practice. In *12th Annual European Symposium on Algorithms*, pages 677–688, 2004.

[14] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *Journal of the ACM*, 42(2):321–28, 1995.

[15] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18:263–70, 1997.

[16] 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.

[17] G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–296, 1994.

[18] M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. volume 5, pages 141–153, 1980.

[19] P. H. Sweany and S. J. Beaty. Dominator-path scheduling: A global scheduling method. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 260–263, 1992.

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

[21] R. E. Tarjan. Testing flow graph reducibility. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, pages 96–107, 1973.

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

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

[24] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, 1981.

[25] R. W. Whitty. Vertex-disjoint paths and edge-disjoint branchings in directed graphs. *Journal of Graph Theory*, 11:349–358, 1987.

[26] A. Wirth. Manuscript, 2003.