McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

An Efficient Incremental Algorithm for Maintaining Dominator Trees and its Application to ϕ-Nodes Update

Vugranam C. Sreedhar
Guang R. Gao
Yong-fong Lee†

ACAPS Technical Memo 77
July 22, 1994

† Application Development Technology Institute
IBM Software Solutions Division
San Jose, California

Abstract

Data flow analysis based on incremental approach may require that the dominator tree be correctly maintained at all times [CR88]. Previous solutions to the problem of incrementally maintaining dominator trees were restricted to reducible flowgraphs [RR94, CR88]. In this paper we present a new algorithm for incrementally maintaining the dominator tree of an arbitrary flowgraph, both reducible and irreducible, based on a program representation called the DJ-Graph [SG94, SGL94]. For the case where an edge is inserted, our algorithm is also faster than previous approaches. For the deletion case, our new incremental algorithm is likely to run fast on the average cases. We also discuss one important application of dominator tree update: incrementally updating ϕ-nodes of an arbitrary Sparse Evaluation Graph. This application is important in the context of incremental SSA form [CSS94].

Contents

List of Figures

1 Introduction

In the past few years, Static Single Assignment (SSA) form [CFR$^+$89, CFR$^+$91], Sparse Evaluation Graphs (SEG) [CCF91], and other related intermediate representations have been successfully used for efficient data flow analysis and program transformations [CLZ86, RWZ88, AWZ88, WZ85, WCES94]. Construction of these intermediate representations needs to first compute the dominance relation of a flowgraph. An inherent difficulty in aggressive program optimizations based on SSA is maintaining the correct SSA form after every program transformation. To achieve this one must correctly maintain the dominator tree of the flowgraph.

In this paper we present an algorithmic framework for incrementally maintaining the dominator tree of an arbitrary flowgraph. For this problem previous work most relevant to this paper includes only the Carroll-Ryder algorithm [CR88] and the Ramalingam-Reps algorithm [RR94]. Both methods are restricted to reducible flowgraphs. By contrast, our approach can handle irreducible as well as reducible flowgraphs. Our algorithms only make updates based on the nodes which are "affected" by the corresponding insertion or deletion of an edge (thus reducing the total cost); furthermore, the properties of DJ-graph are applied to efficiently identify these affected nodes. For the case where an edge is inserted, our algorithm has a $O(E)$ time complexity where E is the number of edges in the flowgraph — better than previous approaches [CR88, RR94]. For the deletion case, our new incremental algorithm is also competitive in term of runtime. It is expected to to run faster on the average cases while not compromising its worst-case time complexity.

To incrementally maintain the dominator tree, we use properties of DJ-graphs [SG94, SGL94] and iterated dominance frontiers [CFR$^+$91]. A DJ-graph is nothing but a dominator tree augmented with join edges that potentially lead to merge points in its corresponding flowgraph. In a flowgraph, an edge $x \rightarrow y$ is a join edge if x does not strictly dominate y. DJ-graphs allow us to solve a number of flowgraph problems [SGL94, SG94]. In particular, it enables us to design an algorithm for computing ϕ-nodes for an arbitrary Sparse Evaluation Graph (SEG) in linear time and space [SG94]. In this paper, we go one step further and show how to update a DJ-graph in response to changes made in its corresponding flowgraph. Since DJ-graphs are dominator trees augmented with join edges, the problem of updating DJ-graphs subsumes the problem of updating dominator trees.

Once the DJ-graph is updated, we next show how to incrementally maintain the set of ϕ-nodes for an arbitrary SEG. We are not aware of any literature on this problem. Although we recently introduced a linear time algorithm for computing ϕ-nodes, we believe in practice incrementally maintaining ϕ-nodes would be faster than the (exhaustive) linear time algorithm. In addition, our solution to the ϕ-nodes update problem forms a basis for a complete incremental data flow framework based on SEGs (a subject of future work).

Organization. Section 2 introduces notation and terminology that will be used later. Section 3 illustrates the difficulty in updating dominator trees, and defines our problem statement of updating DJ-graphs. Sections 4 and 5 together present our DJ-graph update algorithm in handling edge insertions and deletions, respectively. Section 6 applies results obtained thus far to updating the ϕ-nodes of an SEG. Section 7 discusses related work. Finally, Section 8 gives our concluding remarks.

2 Background and Notation

A **flowgraph** is a directed graph $G = (N, E, \mathrm{START})$, where N is the set of nodes, E is the set of flow edges, and $\mathrm{START} \in N$ is a distinguished start node.[1] Figure 1(a) shows an example flowgraph. A *path* of length n is a sequence of edges $x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_n$, where every $x_i \rightarrow x_{i+1} \in E$.

Given a flowgraph of a program, we define the reachable subgraph $\mathsf{REACH}(\mathrm{START})$ to be a subgraph of G such that all nodes in $\mathsf{REACH}(\mathrm{START})$ are reachable from START.

In the reachable subgraph $\mathsf{REACH}(\mathrm{START})$, a node x **dominates** another node y iff all paths from START to y pass through x. We write x **dom** y to indicate that x dominates y, and write x **!dom** y if x does not dominate y. If x **dom** y and $x \neq y$, then x **strictly dominates** y. We write x **sdom** y to indicate x strictly dominates y, and write x **!sdom** y if x does not strictly dominate y. The dominance relation is reflexive and transitive, and can be represented by a tree, called the dominator tree. If x is a parent node of y in the dominator tree, then x **immediately dominates** y, and we write $idom(y)$ to denote the immediate dominator of y. Given a node x in the dominator tree, we define $SubTree(x)$ to be the dominator sub-tree rooted at x. Note that the nodes in $SubTree(x)$ is simply the set of all nodes dominated by x.

For each node in the dominator tree we associate a *level number* that is the depth of the node from the root. We write $x.level$ to indicate the level number of a node x.

The **dominance frontier** $DF(x)$ of a node x is the set of all y such that x dominates a predecessor of y, but x does not strictly dominate y [CFR+91].

We can extend the definition of dominance frontier $DF(S)$ to a set of nodes S:

$$DF(S) = \bigcup_{x \in S} DF(x)$$

We define **iterated dominance frontier** $IDF(S)$ for a set of nodes S as the limit of the increasing sequence:

$$\begin{aligned} IDF_1(S) &= DF(S), & (1)\\ IDF_{i+1}(S) &= IDF(S \cup IDF_i(S)) & (2) \end{aligned}$$

If S is a set, we will use the notation $|S|$ to represent the number of elements in the set.

[1] Note that a flowgraph does not need to be connected; that is, some nodes may not be reachable from the START node.

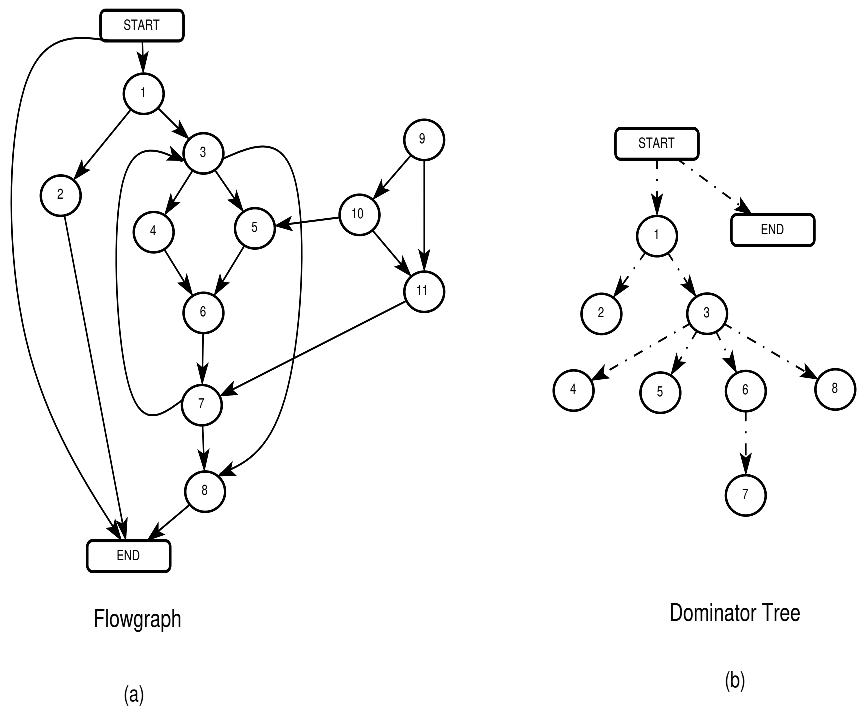

Figure 1: A flowgraph and its dominator tree

Recently we proposed a linear time algorithm for computing the iterated dominance frontier for a set of nodes, using a program representation called the DJ-graph [SG94]. A DJ-graph is a dominator tree augmented with join edges. A flowgraph edge is a join edge if its source node does not strictly dominate its destination node. In constructing the DJ-graph for the reachable subgraph REACH(START), we first build its dominator tree and then appropriately insert J-edges. Figure 2 shows the DJ-graph of our example flowgraph given in Figure 1(a). To see how a join edge is inserted, let us consider node 3. It has two incoming edges $1 \to 3$ and $7 \to 3$. Of these two edges only $7 \to 3$ is a join edge (since node 1 strictly dominates 3, $1 \to 3$ is not a join edge).

> **Clarification:** We would like to clarify some terminology in order to avoid confusion in later discussions. Dominator trees and DJ-graphs are constructed only for the reachable subgraph REACH(START). Whenever we use the phrase "a flowgraph and its DJ-graph," we are refering to " the reachable subgraph REACH(START) and its DJ-graph."
>
> A DJ-graph consists of a dominator tree and join edges. Therefore, whenever we use the term dominator tree, we also mean a DJ-graph without J-edges.
>
> Since a flowgraph and its DJ-graph contain the same set of nodes, whenever we use the term *node* x, the node x may belong to either the flowgraph or the DJ-graph, unless

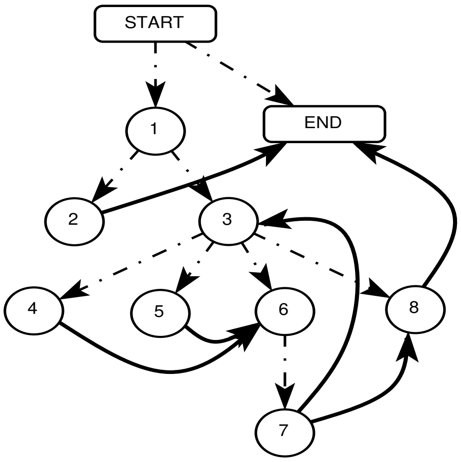

Figure 2: The DJ-graph of the example flowgraph

explicitly specified using subscript notation with f for flowgraph and d for DJ-graph. When we describe a graph property without using subscripts, the property holds on both a flowgraph and its DJ-graph.

Finally, the iterated dominance frontier of a node y is the same when computed on either the DJ-graph or the flowgraph. Thus, all of $IDF(y)$, $IDF(y_f)$ and $IDF(y_d)$ will mean the same thing.

Using the DJ-graph we can easily compute the dominance frontier for a node x based on Lemma 2.1. When computing $DF(x)$ we examine all the J-edges "coming out" of $SubTree(x_d)$. The destination nodes of these J-edges are in the dominance frontier of x when they satisfy certain level contraint [SG94]. We have also extended the above intuition and proposed a linear time algorithm for computing the iterated dominance frontier for a set of nodes [SG94].[2].

Lemma 2.1

A node z is in $DF(x)$ if and only if there exists a node y_d in $SubTree(x_d)$ such that $y_d \rightarrow z_d$ is a J-edge and $z_d.level \leq x_d.level$.

3 Dominator Update: Motivation and Problem Definition

Carroll and Ryder pointed out in [CR88]:

[2]For convenience we have briefly presented our linear time algorithm in Appendix B

> "The inherent difficulty in the dominator update problem lies in the 'non-locality' of domination, to wit, given two nodes x and y in the flow graph, whether x dominates y depends on the presence or absence of paths through nodes arbitrarily far from either x or y. Adding or removing a single flow graph edge – an act which can add or remove large numbers of paths – can thus affect domination between nodes arbitrarily far from the altered edge."

Consider our example flowgraph and its dominator tree shown in Figure 1(a) and (b), respectively. Let us see what happens when a new edge $2 \rightarrow 4$ is inserted. With this new edge, the dominance relation for many nodes is affected. For example, in the newly modified flowgraph, node 3 will no longer dominate 4. Actually node 3 ceases to dominate 6 and 8 as well. The reason for this is obvious: By inserting $2 \rightarrow 4$ we have created alternative paths from START to nodes 4, 6 and 8 that do not pass through node 3. Notice that the insertion of $2 \rightarrow 4$ has affected node 8 that is far from both nodes 2 and 4. The situation is also true (but opposite) when an edge is deleted; this deletion can again affect nodes arbitrarily far away.

Also notice, in the above example, the flowgraph becomes irreducible after $2 \rightarrow 4$ is inserted. Previous approaches to this problem that we are aware of, the Carroll-Ryder and the Ramalingam-Reps algorithms, cannot proceed further when this happens. Both methods use properties of flowgraphs and the dominance relation that are only valid in reducible flowgraphs. In our approach, we use properties of DJ-graphs to update dominator trees. These properties are valid for both reducible and irreducible flowgraphs. Consequently, we can handle irreducible flowgraphs while the previous approaches could not.

One of our important observations is that the dominator tree alone is not sufficient to capture all the path information in a flowgraph; it only gives one type of *path* relationship among nodes in a flowgraph – meaning, "a node x dominates another node y iff all *paths* from START to y must pass through x". But adding or deleting an edge can affect the dominance relation between nodes arbitrarily far apart, as was observed by Carroll and Ryder. Therefore, a key question to be answered in the rest of this paper is: how can we efficiently compute the set of affected nodes no matter how far away they may be from the updated edge? (A node is called *affected* iff its immediate dominator will change because of a flowgraph update.)

The problem we consider in the next two sections is the DJ-graph update problem: We want to maintain the *DJ-graph* for the reachable subgraph of a flowgraph in which changes are made one at a time. This problem subsumes the dominator tree update problem. Algorithms 4.1 and 5.1 presented in Sections 4 and 5, respectively, collectively form our incremental algorithm for maintaining the DJ-graph. Algorithms 4.1 and 5.1 handle situations where x and y are reachable from START both before and after an update. We will handle other situations as special cases.

4 Dominator Update: Insertion of an Edge

In this section, we present a simple algorithm for updating the DJ-graph (and hence the dominator tree) of a flowgraph in response to a flowgraph edge insertion. Recall that when an edge $x_f \rightarrow y_f$ is inserted, it can affect nodes arbitrarily far away. One of our key observations is that all the *affected* nodes must be in the set $\{y\} \cup IDF(y)$. The reason for this is as follows: By inserting the edge $x_f \rightarrow y_f$, we may have created a path (in the flowgraph) from START_f to a node, say u_f, in $\{y_f\} \cup IDF(y_f)$ such that the path includes $x_f \rightarrow y_f$ but bypasses the immediate dominator of u_f. However, not all the nodes in $\{y\} \cup IDF(y)$ will be truly affected. We will formally prove later that affected nodes are only those nodes that additionally satisfying certain level constraint.

Once we find the *exact* set of affected nodes, we need to answer: What will be their new immediate dominators after the edge insertion? First note that $nca(x_d, y_d)$ will definitely dominate all the affected nodes. Ramalingam and Reps gave a stronger claim to answer this question: Not only does $nca(x_d, y_d)$ dominate all the affected nodes, but also it is actually their immediate dominator. We will utilize this claim in our algorithm too.

Before presenting our algorithm for handling an edge insertion, we want to clarify the following point: We do not incrementally maintain the iterated dominance frontiers (IDFs) for all the nodes in our approach. There are two reasons. First, we have a very efficient algorithm for computing $IDF(y)$ in linear time when we are told that $x \rightarrow y$ is *the* newly inserted edge [SG94]. Second, if we were to update IDFs, we would need to pay extra space and time overhead. The time required for updating IDFs for all the nodes, can be much longer than that spent in computing $IDF(y)$ for one single node y.

4.1 Algorithm

Let $x_f \rightarrow y_f$ be the newly inserted flowgraph edge. Assume that both x_f and y_f are reachable from START even before the insertion. Algorithm 4.1 gives a procedure to restructure the DJ-graph in response to changes in the dominance relation when $x_f \rightarrow y_f$ is inserted. There are two supporting functions used in the algorithm: (1) $link(x_d, y_d, Jedge\ or\ Dedge)$ inserts into the DJ-graph a new edge $x_d \rightarrow y_d$ of the appropriate type; and (2) $cut(x_d, y_d)$ deletes $x_d \rightarrow y_d$.

Algorithm 4.1 *This algorithm updates a DJ-graph when a new edge $x_f \rightarrow y_f$ is inserted in its flowgraph.*

UpdateInsertEdge(x_d, y_d)
{

1: $z_d = nca(x_d, y_d)$
2: $\mathsf{DomAffected}_\mathsf{I}(y_d) = \{w_d | w_d \in (\{y_d\} \cup IDF(y_d))$ **and** $w_d.level > z_d.level + 1\ \}$
3: **if**$(z_d! = x_d)$
4: $link(x_d, y_d, Jedge)$ `/* if` $z_d = x_d$ `then we should insert a D-edge. See step` $\boxed{\mathbf{11}}$ `*/`

```
5:      foreach w_d ∈ DomAffected_I(y_d) do
6:         u_d = idom(w_d) ; /* u_d is the (old) immediate dominator */
7:         cut(u_d, w_d) /* cut the old D-edge */
8:         if(u_f → w_f is a flowgraph edge)
9:            link(u_d, w_d, Jedge) /* insert a J-edge from u_d to w_d */
10:        endif
11:        link(z_d, w_d, Dedge) /* new D-edge */
12:        UpdateLevelNumber(w_d)  /* Update the level for nodes in SubTree(w)  */
13:     endfor

}
```

In this algorithm, we first compute $z_d = nca(x_d, y_d)$. Then we compute, at step $\boxed{\mathbf{2}}$, the set of affected nodes $\mathsf{DomAffected}_{\mathsf{I}}(y_d) = \{w_d | w_d \in (\{y_d\} \cup IDF(y_d)$ **and** $w_d.level > z_d.level + 1\}$. There are two conditions for a node to be in $\mathsf{DomAffected}_{\mathsf{I}}(y_d)$: (1) it must be in $y_d \cup IDF(y_d)$, and (2) its level number must be greater than $nca(x_d, y_d).level + 1$. In using our linear time algorithm for computing $IDF(y_d)$ on-the-fly, we thus can avoid processing any node whose level number is not greater than $nca(x_d, y_d).level + 1$ [SG94].[3]

Once we have computed $\mathsf{DomAffected}_{\mathsf{I}}(y_d)$, we pull up each affected node w_d and make z_d (i.e., $nca(x_d, y_d)$) its new immediate dominator (step $\boxed{\mathbf{11}}$). We also delete the D-edge to every affected node from its (old) immediate dominator (step $\boxed{\mathbf{7}}$). If there is a flowgraph edge between these two nodes, we insert a J-edge in place of the deleted D-edge at step $\boxed{\mathbf{8}}$. Finally, we update the level number for all the descendant nodes of every affected nodes at step $\boxed{\mathbf{12}}$.

Example. Consider our example flowgraph in Figure 1(a) and its DJ-graph in Figure 2. Let us insert a new edge $2 \rightarrow 4$ in the flowgraph. The resulting flowgraph is shown in Figure 3(a). From the DJ-graph (in Figure 2) we can find $nca(2, 4) = 1$ and compute $\mathsf{DomAffected}_{\mathsf{I}}(4) = \{4, 6, 8\}$. After this we pull up all the affected nodes and make node 1 their new immediate dominator. At the same time, we remove D-edges $3 \rightarrow 4$, $3 \rightarrow 6$ and $3 \rightarrow 8$. Since node 2 does not dominate 4, we insert a new J-edge $2 \rightarrow 4$. We also insert J-edges $3 \rightarrow 4$ and $3 \rightarrow 8$ in the DJ-graph, because their counterparts exist in the flowgraph. The updated DJ-graph is given in Figure 3(b).

4.2 Other Cases

Let $x_f \rightarrow y_f$ be the newly inserted edge in the flowgraph. Here we will describe how to handle other cases where the reachability of x_f and y_f from START_f is different from what we have assumed in the previous subsection. The first case is where x_f is not reachable, for which we do nothing because we maintain the DJ-graph only for the reachable subgraph $\mathsf{REACH}(\mathrm{START})$.

[3] For convenience we provide a high-level description of our linear time algorithm in Appendix B

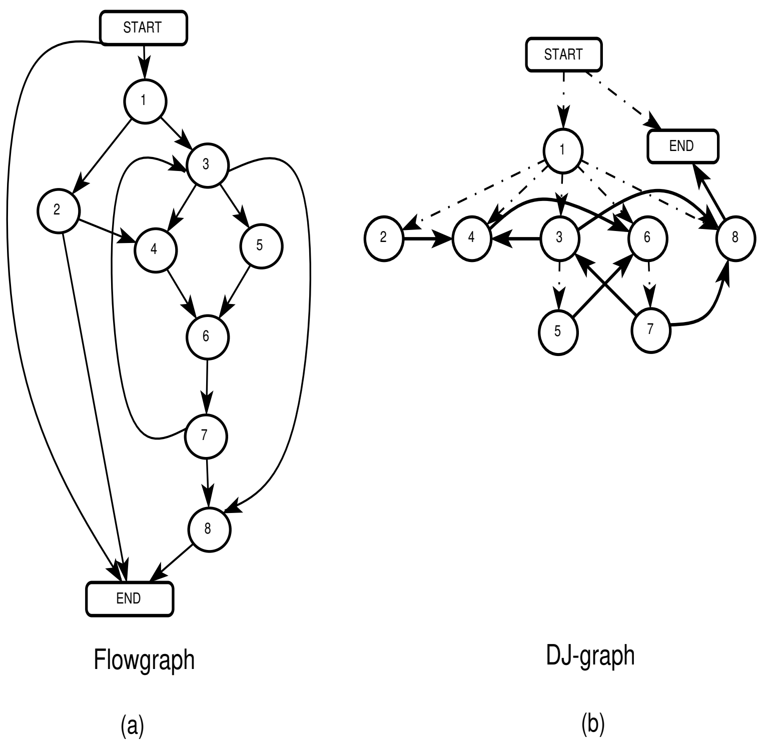

Figure 3: The flowgraph and its DJ-graph after $2 \to 4$ inserted

In the second case, y_f becomes reachable only after the edge insertion. For this we first build (using exhaustive algorithm) the DJ-subgraph for the sub-flowgraph induced by nodes reachable from y_f but not reachable from START_f. In constructing this DJ-subgraph, we treat y_d as its root. Since x must dominate y, we then insert a D-edge from x_d to y_d (to connect the newly built DJ-subgraph with the existing DJ-graph). Finally, from the viewpoint of updating DJ-graph, we pretend each edge $u_f \to v_f$ to be a newly inserted flowgraph edge, where u_f becomes reachable only after $x_f \to y_f$ is inserted, and v_f is reachable even before the edge insertion. This surprisingly corresponds to the case discussed in the previous subsection. To complete the DJ-graph update, we invoke Algorithm 4.1 once for each $u_f \to v_f$.

For example, node 9 is unreachable in Figure 1(a), but becomes reachable after we insert $1 \to 9$. Therefore, we first construct the DJ-subgraph for the sub-flowgraph induced by nodes 9, 10, and 11. Then we insert a D-edge from 1 to 9. Finally, we use Algorithm 4.1 to update the DJ-graph once for $10 \to 5$ and once for $11 \to 7$ as if they were newly inserted flowgraph edges.

4.3 Correctness and Complexity

In this subsection, we argue the correctness of Algorithm 4.1 (Theorem 4.1) and analyze its complexity (Theorem 4.2). Without loss of generality, we assume that x_f and y_f are reachable both before and after the insertion of $x_f \to y_f$.

Theorem 4.1 is the main theorem that establishes the correctness of Algorithm 4.1. Its proof is based on Lemmas 4.1 and 4.4. Lemma 4.1 claims that a unique node will be the new immediate dominator of all the affected nodes. Lemma 4.4 gives a sufficient and necessary condition to determine the *exact* set of affected nodes. Its validity is further based on two other lemmas: Lemma 4.2 and Lemma 4.3. Lemma 4.2 claims that if $u \in IDF(y)$ then $idom(u)$ strictly dominates y. Lemma 4.3 establishes a relation between any two nodes u and v when $v \notin IDF(u)$ and v is reachable from u.

Lemma 4.1 claims that $nca(x_d, y_d)$, before the DJ-graph is updated, will be the new immediate dominator of all the affected nodes (after the update). This is one of the key result for the correctness of our algorithm. It also forms the basis for our second key result (Lemma 4.4).

Lemma 4.1

Let $x_f \to y_f$ be a newly inserted edge and let $z_d = nca(x_d, y_d)$. Then z must immediately dominate every affected node after the insertion.

Proof:

Let w be any affected node after $x_f \to y_f$ is inserted. Then z will still dominate w after the insertion. Let v be the new immediate dominator of w after the insertion. Suppose that v were not z. Then z **sdom** v **sdom** w. Consider any flowgraph path P: $z_f \xrightarrow{*} x_f \to y_f \xrightarrow{*} w_f$. P must include v_f since $v = idom(w)$. There are two possibilities:

1. Node v_f is in $z_f \xrightarrow{*} x_f$. This implies that v_f is also on any path from z_f to x_f before $x_f \to y_f$ is inserted. Therefore, v should have been the nearest common ancestor of x and y. This contradicts $z_d = nca(x_d, y_d)$.
2. Node v_f is in $y_f \xrightarrow{*} w_f$. This implies that the insertion of $x_f \to y_f$ should not have affected v. Therefore, v is already w's immediate dominator before the edge insertion, so w should not be affected. This contradicts our assumption that w is an affected node.

Since neither possibility can be true, v must be the same as z; that is, z must be the immediate dominator of any affected node. ∎

To prove our second key result (Lemma 4.4), we need the following two supporting lemmas. Lemma 4.2 establishes a relation between a node w and the immediate dominator of a node $u \in IDF(w)$.

Lemma 4.2

If $u \in IDF(w)$, then $idom(u)$ **sdom** *w.*

Proof:

Let $s = idom(u)$. We will first show that if $u \in DF(w)$ then s **sdom** w. Using this result we will prove the validity of the lemma.

Assume that s does not strictly dominate w. Then there must be a path from START to w that does not pass through s. Since $u \in DF(w)$, there must be a predecessor node p strictly dominated by w (follows from Lemma 2.1). From this we can immediately see that there is a path from START $\xrightarrow{*} w \xrightarrow{+} p \rightarrow u$ in the DJ-graph that does not pass through s, contradicting the assumption that $s = idom(u)$. Therefore s **sdom** w.

Now, if $u \in IDF(w)$, then there exist nodes $t_0, \ldots, t_k$ such that $u = t_0$, $w = t_k$, and $t_i = DF(t_{i+1})$, where $0 \leq i \leq k-1$. So we have $idom(t_i)$ **sdom** t_{i+1}, $0 \leq i \leq k-1$. Consequently, $idom(t_i)$ **dom** $idom(t_{i+1})$, $0 \leq i \leq k-1$. Hence $idom(t_0)$ **dom** $idom(t_k)$; that is, $idom(u)$ **dom** $idom(w)$. This implies $idom(u)$ **sdom** w. ∎

The next supporting lemma establishes a relation between any two nodes u and w such that u is reachable from w but is not in $IDF(w)$.

Lemma 4.3

Let u be reachable from another node $w \neq u$, and let $u \notin IDF(w)$. Then there exists a node $s \in w \cup IDF(w)$ such that s **sdom** *u.*

Proof:

Assume w !**sdom** u (otherwise, the proof is trivial).

Since u is reachable from w, let P be a path from w to u. Let s be the node on P such that $s \in IDF(w)$ and is closest to u among all nodes which are on P and are in $IDF(w)$. Such a s must exists (e.g. in the extreme w can be such a s).

Now we claim that all nodes on P between s and u (excluding u) must be dominated by s. We show this claim by contradiction. Assume that some of these nodes are not dominated by s. Let t be one such nodes that is closest to s on P. Then by definition of dominance frontiers, t must be in $DF(s)$ (hence in $IDF(w)$) – a contradiction (since we know s is the only node in $IDF(w)$ which lies between s and u!)

Based on this claim, we state that u must be strictly dominated by s. Otherwise, u will be in $DF(s)$ (hence in $IDF(w)$) which contradict to our lemma assumption. ∎

Next we present our second key result. Using this lemma we can determine the exact set of nodes that are indeed affected when an flowgraph edge $x_f \to y_f$ is inserted.

Lemma 4.4

Let $x_f \to y_f$ be a newly inserted edge in the flowgraph, and let $z = nca(x, y)$. Let $\mathsf{DomAffected}_{\mathsf{I}}(y) = \{v | v \in (\{y\} \cup IDF(y))$ **and** $v_d.level > z_d.level + 1\}$*. Then a node u is* **affected** *iff* $u \in \mathsf{DomAffected}_{\mathsf{I}}(y)$*.*

Proof:

The "if" part: We want to show that if u is in $\mathsf{DomAffected}_{\mathsf{I}}(y)$, then it is **affected**. Let $w = idom(u)$ before $x_f \to y_f$ is inserted. Since u is in $\mathsf{DomAffected}_{\mathsf{I}}(y)$, $u_d.level > z_d.level + 1$. This implies $w_d.level = u_d.level - 1 > z_d.level$. Thus we first conclude that w_d must be a proper descendant of z_d and cannot be the same as z_d.

The fact of $u \in \mathsf{DomAffected}_{\mathsf{I}}(y)$ also implies $u = y$ or $u \in IDF(y)$. If $u = y$, then w **sdom** y. If $u \in IDF(y)$, then $w = idom(u)$ **sdom** y by Lemma 4.2. In summary, w must strictly dominate y. But w **!sdom** x because otherwise w, in place of z, would have been the nearest common ancestor of x and y.

Now consider the insertion of $x_f \to y_f$ in the flowgraph. Since w does not strictly dominate x, the edge insertion creates at least one path $\mathrm{START}_f \xrightarrow{*} x_f \to y_f \xrightarrow{*} u_f$ that bypasses w_f both when $u = y$ and when $u \in IDF(y)$. Consequently, u is truly affected.

only if Here we will show that if u_d is affected then u_d is in $\mathsf{DomAffected}_{\mathsf{I}}(y_d)$. First we will show that if u_d is afftected then $u_d.level > z_d.level + 1$. Assume to the contrary that $u_d.level \leq z_d + 1$. There are two cases

1. First let us consider the case the $u_d.level = z_d.level + 1$. So z_d must immediately dominate u_d. Now since z_d dominates both x_d and y_d by inserting an edge $x_d \to y_d$ we have not changed the path relationship as far as nodes z_d and u_d is concerned. In other words, by introducing the edge $x_d \to y_d$ in the DJ-graphs, all paths from START to u_d must still pass through z_d (the immediate dominator of u_d). Therefore u_d is not affected.

2. Now assume that $u_d < z_d + 1$. It is easy to see that all paths from START to u_d must still pass through w_d (the immediate dominator node before the insertion). This contradicts the assumption that u_d is affected.

Therefore, $u_d.level$ must be strictly greater than $z_d + 1$.

Next we will show that u_d is either y_d or is in $IDF(y_d)$. It is obvious to see that that if u_d is same as y_d, u_d is in $\mathsf{DomAffected_I}(y_d)$.

Now assume that $u_d \neq y_d$. We will show that $u_d \in IDF(y_d)$. Assume to the contrary that u_d is not in $IDF(y_d)$. Since u_d is reachable from y_d, either y_d strictly dominate u_d or there must be a node s_d that strictly dominates u_d and $s_d \in IDF(y_d)$ (follows from Lemma 4.3). In both cases we can see that u_d is not affected. But we know that u_d is indeed affected, so u_d must be in $IDF(y_d)$.

■

Finally, we state our first main theorem that establishes the correctness of Algorithm 4.1.

Theorem 4.1

Algorithm 4.1 correctly updates the DJ-graph of a flowgraph when a new edge $x_f \rightarrow y_f$ is inserted in the flowgraph.

Proof:

The proof of the theorem easily follows from 4.1 and 4.4. ■

Next we state the second main theorem that gives the worst-case time complexity of Algorithm 4.1.

Theorem 4.2

Assume that both x_f and y_f are reachable before the insertion of $x_f \rightarrow y_f$. Then the worst-case time complexity of Algorithm 4.1 is $O(e)$, where e is the number of edges in the DJ-subgraph induced by the nodes in $SubTree(nca(x_d, y_d))$.

Proof:

From Lemma 4.1 we know $nca(x_d, y_d)$ must immediately dominate all the affected nodes. Now to determine which nodes are indeed affected we use Lemma 4.4. Since we can compute $IDF(y)$ in linear time with respect to the number of outgoing D-edges and J-edges from the nodes in $SubTree(y_d)$

(see [SG94] for details), the set $\mathsf{DomAffected}_\mathsf{I}(y)$ can also be computed in linear time. In addition, the time complexity for computing $nca(x_d, y_d)$ is linear with respect to the number of nodes in $SubTree(nca(x_d, y_d))$, and the functions $link()$ and $cut()$ take $O(1)$ time. Therefore, the worst-case time complexity for updating the DJ-graph in response to a flowgraph edge insertion is $O(e)$. Note that we never visit nodes outside of $SubTree(nca(x_d, y_d))$ in the update process. ∎

5 Dominator Update: Deletion of an Edge

In this section, we show how to update a DJ-graph when a flowgraph edge $x_f \rightarrow y_f$ is deleted. The effect of deleting an edge is opposite and complementary to that of inserting the same edge. When inserting $x_f \rightarrow y_f$, we pull up the affected nodes in the DJ-graph to the level of $nca(x_d, y_d).level+1$. Note that if $\mathsf{DomAffected}_\mathsf{I}(y_d)$ is not empty, it must include y_d. Therefore, all the other affected nodes will be siblings to y_d after the DJ-graph update. By contrast, when deleting $x_f \rightarrow y_f$, we pull down these affected nodes. In the deletion case, however, we have two major difficulties. First, it is hard to find the exact set of affected nodes. Second, the new immediate dominators of the affected nodes can be different.

To overcome the first difficulty, we use a conservative approximation for the set of affected nodes. This set of *possibly affected* nodes is $\mathsf{DomAffected}_\mathsf{D}(y_d) = \{w_d | w_d \in IDF(y_d) \textbf{ and } w_d.level = y_d.level\}$.[4] Since this set is a safe approximation, not every node in the set will be pulled down in response to a flowgraph edge deletion. To overcome the second difficulty while avoiding an exhaustive algorithm, we observe that the new immediate dominator of any affected node must be a descendant of $idom(y_d)$. Therefore, if any update on the DJ-graph needs to happen, it will only affect the nodes in $SubTree(idom(y_d))$. Based on these two observations, we present an efficient incremental algorithm for only computing the possibly affected nodes' *new dominators*, which are then used to compute their new *immediate* dominators.

5.1 Algorithm

As pointed out previously, our approach to updating a DJ-graph in response to a flowgraph edge deletion will be centered at an incremental algorithm for computing the new dominators for all the possibly affected nodes. In the following, we will explain how we transform the *exhaustive* Purdom-Moore algorithm into an *incremental* algorithm that will be much more efficient in practice. The original Purdom-Moore algorithm is given in Appendix A for reference.

[4] $\mathsf{DomAffected}_\mathsf{D}(y_d)$ can possibly be reduced by removing any node v_d such that $idom(y_f) \rightarrow v_f$ is a flowgraph edge, since v_d can never be pulled down. For simplicity, we use only the two given conditions to determine if a node is in $\mathsf{DomAffected}_\mathsf{D}(y_d)$.

Let $Dom(v)$ denote the set of dominators for node v. Our modifications to the exhaustive Purdom-Moore algorithm are the following: First, $z_d = idom(y_d)$ will continue to dominate every possibly affected node in $\mathsf{DomAffected}_{\mathsf{D}}(y_d)$ after the update, although it may no longer be its immediate dominator. We, therefore, can focus on the nodes in $SubTree(z_d)$ while ignoring other. Consequently, we will compute the $Dom()$ sets for all the possibly affected nodes as if no other nodes existed outside of $SubTree(z_d)$. Second, we will monitor the $Changes$ condition (step $\boxed{\mathbf{37}}$ in Algorithm A.1) only for the possibly affected nodes. Third, whenever $Dom(w_d)$ changes for a node $w_d \in \mathsf{DomAffected}_{\mathsf{D}}(y_d)$, the $Dom()$ set also changes for every descendant v_d of w_d. This seems to indicate that we also must observe the $Changes$ condition for any descendant of every possibly affected node. Fortunately, $Dom(v_d)$ can be "partially deduced" from $Dom(w_d)$ as explained later.

For our algorithm design we partition all the nodes in $SubTree(z_d)$ into three classes:

1. **PossiblyAffected:** The set of nodes in this class is the same as $\mathsf{DomAffected}_{\mathsf{D}}(y_d)$.

2. **PseudoAffected:** This set consists of any *proper* descendant of a **PossiblyAffected**node. Note that a **PseudoAffected** node's immediate dominator does not change, but its dominators may change.

3. **NotAffected:** This set is defined to capture all the nodes not in the first two classes. Neither does a **NotAffected** node's immediate dominator change, nor do its dominators.

For example, consider deleting $2 \rightarrow 4$ from the flowgraph in Figure 3(a). $IDF(4) = \{3, 6, 8\}$. By examining the DJ-graph in Figure 3(b), we will find that (1) nodes 3, 4, 6 and 8 are **PossiblyAffected** nodes; (2) nodes 5 and 7 are **PseudoAffected** nodes; and (3) nodes 1 and 2 are **NotAffected** nodes.

Next we partition $Dom(v_d)$ for each node $v_d \in SubTree(idom(y_d))$ into two parts: the static part $Dom_{st}(v_d)$ and the dynamic part $Dom_{dy}(v_d)$. The static part captures nodes that will continue to dominate v_d after the update. For example, consider the DJ-graph shown in Figure 3(b). Node 1 and 2 will dominate node 2 even after the edge deletion, so $Dom_{st}(2) = \{1, 2\}$.

By contrast, the dynamic part of a $Dom()$ set normally will change during the fixed point iteration. We are interested in $Dom_{dy}()$ for only the **PossiblyAffected** nodes because we want to compute their immediate dominators. Therefore, we carefully initialize and manipulate the $Dom()$ set in such a way that we update the $Dom()$ set only for the **PossiblyAffected** nodes.

Let $z_d = idom(y_d)$. The first step of our approach is to initialize the $Dom()$ set for each node according to its class as follows.

$v_d \in$ **PossiblyAffected.** For each **PossiblyAffected** node v_d, $Dom_{st}(v_d) = \emptyset$ and $Dom_{dy}(v_d)$ = all the nodes in $SubTree(z_d)$. We basically need to recompute the dominators for each **PossiblyAffected** node. For instance, consider the DJ-graph in Figure 3(b). Node 4 is in **PossiblyAffected** and $Dom_{dy}(4) = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

$v_d \in$ **PseudoAffected.** Recall that the immediate dominator of a **PseudoAffected** node will not change, but its $Dom()$ set may. Let v_d be a **PseudoAffected** node. By definition, there will be a unique node $w_d \in$ **PossiblyAffected** that strictly dominates v_d. Here we will discuss how to initialize the static part and compute the dynamic part of v_d.

Let S be the set of nodes on the dominator tree path from w_d to v_d (excluding w_d). All the nodes in S will still dominate v_d even after the update, so they are used to initialize $Dom_{st}(v_d)$. For example, consider the DJ-graph in Figure 3(b) again. The set S for 5 is $\{5\}$, so $Dom_{st}(5) = \{5\}$. Now for the dynamic part, we can see that the nodes in $Dom()$ that is not accounted for by the static part are all the nodes dominating w_d. This implies that $Dom_{dy}(v_d)$ is the same as $Dom(w_d)$. Return to our example; $Dom_{dy}(5) = Dom(3) = Dom_{dy}(3) = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

$v_d \in$ **NotAffected.** For a **NotAffected** node v_d, neither its $Dom()$ set nor its immediate dominator will change. Therefore, we initialize $Dom_{st}(v_d)$ to be all the nodes dominating v_d, and make its dynamic part empty. Again consider the DJ-graph in Figure 3. Node 2 is a **NotAffected** node, and $Dom_{st}(2) = \{1, 2\}$.

Given the above initialization we present the following algorithm. When the algorithm terminates, the dominators for each **PossiblyAffected** node w_d will be in $Dom(w_d)$. Using this information we can easily determine the immediate dominators for all the possibly affected nodes. After this we can easily update the DJ-graph accordingly.

Algorithm 5.1 *This algorithm incrementally computes the dominators for all the possibly affected nodes when $x_f \rightarrow y_f$ is deleted from the flowgraph. (Assume that both nodes are still reachable after the deletion.)*

UpdateDomDel(x, y)
{

DomAffected$_{\mathsf{D}}(y_d) = \{w_d | w_d \in (\{y_d\} \cup IDF(y_d))$ **and** $w_d.level = y_d.level\}$
if$(x_d != idom(y_d))$ `/*` do not cut if $x_d \rightarrow y_d$ is a D-edge! `*/`
  $cut(x_d, y_d)$
$z_d = idom(y_d)$
Partition all the nodes in $SubTree(z_d)$ into **PossiblyAffected**, **PseudoAffected**, and **NotAffected**.
Initialize each node $w_d \in SubTree(z_d)$ as described in the main text.
while$(Changes$ to any $Dom(w_d)$ such that $w_d \in$ **PossiblyAffected** occur$)$ **do**
  foreach $w_d \in$ **PossiblyAffected do**
    $DomTemp = SubTree(z_d)$ `/* temporary variable */`
    foreach $p_f \in Pred(w_f)$ **do** `/*` p_f is an immediate predecessor in the flowgraph `*/`
      if$(p_d \in$ **PseudoAffected**$)$

25: $Dom_{dy}(p_d) = Dom(u_d)$, where $u_d \in$ **PossiblyAffected** and u_d **sdom** p_d
26: **endif**
27: $DomTemp = DomTemp \cap (Dom_{st}(p_d) \cup Dom_{dy}(p_d))$
28: **endfor**
29: $Dom_{dy}(w_d) := \{w_d\} \ \cup \ DomTemp$
30: **endfor**
31: **endwhile**
32: Compute the immediate dominators for all the possibly affected nodes.
33: Update the DJ-graph accordingly.

}

In the above algorithm, after the completion of step $\boxed{\textbf{23}}$ **foreach** loop, the variable $DomTemp$ contains the intersection of the $Dom()$ sets from a node's immediate predecessors. There are two key points to note in this algorithm: (1) We check the $Changes$ condition only for the nodes in **PossiblyAffected**; (2) If an immediate predecessor p_d at step $\boxed{\textbf{24}}$ belongs to **PseudoAffected**, we make $Dom_{dy}(p_d)$ be $Dom(u_d)$, where $u_d \in$ **PossiblyAffected** and u_d **sdom** p_d. We, therefore, do not explicitly recompute the dynamic part of $Dom(p_d)$.

Example. Consider the flowgraph and its DJ-graph shown in Figure 2. Assume that $2_f \rightarrow 4_f$ is to be deleted from the flowgraph. To update the DJ-graph we first incrementally compute the $Dom()$ set for every possibly affected node. Since $\mathsf{DomAffected}_{\mathsf{D}}(4) = \{4, 3, 6, 8\}$, we have **PossiblyAffected** $= \{3, 4, 6, 8\}$, **PseudoAffected** $= \{5, 7\}$, and **NotAffected** $= \{1, 2\}$. Also $idom(4) = 1$. The table below gives the initial values of the $Dom()$ sets.

$Node$	$Node\ Type$	$Dom_{st}()$	$Dom_{dy}()$
3	**PossiblyAffected**	$\emptyset$	$\{1, 2, 3, 4, 5, 6, 7, 8\}$
4	**PossiblyAffected**	$\emptyset$	$\{1, 2, 3, 4, 5, 6, 7, 8\}$
6	**PossiblyAffected**	$\emptyset$	$\{1, 2, 3, 4, 5, 6, 7, 8\}$
8	**PossiblyAffected**	$\emptyset$	$\{1, 2, 3, 4, 5, 6, 7, 8\}$
5	**PseudoAffected**	$\{5\}$	$Dom(3)$
7	**PseudoAffected**	$\{7\}$	$Dom(6)$
1	**NotAffected**	$\{1\}$	$\emptyset$
2	**NotAffected**	$\{1, 2\}$	$\emptyset$

Next we show the fixed point iteration for the possibly affected nodes as follows. Our example converges in one iteration!

$Node$	$Pred_f$	$DomTemp = \cap_{p \in Pred_f} Dom(p)$	$Dom_{dy}() =$ $DomTemp \cup Node$
3	$\{1, 7\}$	$\{1\} \cap \{1, 2, 3, 4, 5, 6, 7, 8\} = \{1\}$	{3,1}
4	$\{3\}$	$\{1, 3\}$	$\{1, 3, 4\}$
6	$\{4, 5\}$	$\{1, 3, 4\} \cap \{1, 3, 5\} = \{1, 3\}$	$\{1, 3, 6\}$
8	$\{3, 7\}$	$\{1, 3\} \cap \{1, 3, 6, 7\} = \{1, 3\}$	$\{1, 3, 8\}$

Observe that when computing $DomTemp = \cap_{p_f \in Pred_f} Dom(p_d)$ we always select the latest value of $Dom(p_d)$. In addition, whenever such a p_d is a **PseudoAffected** node, we update its $Dom_{dy}()$ set at step $\boxed{\mathbf{24}}$. In the actual implementation, we do not need to update $Dom_{dy}(p_d)$ for $p_d \in$ **PseudoAffected**. Using pointer data structures we can easily point to the set $Dom(u_d)$, where u_d is an ancestor of p_d and $u_d \in$ **PossiblyAffected**.

Once we have computed the $Dom()$ sets, we can easily determine the new immediate dominators for all the affected nodes. In our example, node 1 is the new immediate dominator of 3, which is the new immediate dominator of 4, 6 and 8. With this information we can proceed to update the DJ-graph. The updated DJ-graph is shown in Figure 2.

5.2 Other Cases

Let $x_f \to y_f$ be the edge to be deleted from the flowgraph. Here we will describe how to handle other cases where the reachability of x_f and y_f from START_f is different from what we have assumed. If x is not reachable then we do nothing, because we only intend to maintain the DJ-graph for the reachable subgraph $\mathsf{REACH}(\text{START})$.

Now assume that x is reachable and y becomes unreachable after deletion. In this case, we remove from the DJ-graph all the nodes in $SubTree(y_d)$ and their incident edges. This is because if removing $x_f \to y_f$ makes y_f become unreachable, then all the nodes strictly dominated by y will be unreachable either. Next, we remove the D-edge $x_d \to y_d$. Finally, we update the reachability status for all the nodes that become unreachable due to the edge deletion.

5.3 Correctness and Complexity

In this section we prove the correctness of Algorithm 5.1. Let $x_f \to y_f$ be deleted from the flowgraph. First of all notice that nodes not in $SubTree(idom(y_d))$ can never be pulled down and made the descendants of nodes in the sub-tree. So we will focus only on the nodes in the sub-tree. Given this we will then show, in Lemma 5.1, that $Dom(w_d)$, for $w_d \in$ **PossiblyAffected**, computed by Algorithm 5.1, will be same as the $Dom_{pm}(w_d)$ computed using the exhaustive Purdom and Moore's algorithm by iterating only over all the nodes in the $SubTree(idom(y_d))$.

Recall that when a new edge $x_f \to y_f$ is inserted in the flowgraph, all the affected nodes in the corresponding DJ-graph are pulled up to the same level. Therefore when an edge $x_f \to y_f$ is deleted from the flowgraph, only those nodes that are in $\{y\} \cup IDF(y)$ and are at the same level as y_d can be possibly affected. We will use this fact in Lemma 5.1 to argue that it is sufficient to compute the $Dom()$ only for the possibly affected nodes.

Lemma 5.1

Let $x_f \rightarrow y_f$ be the edge that is deleted from a flowgraph, and let $w_d \in$ **PossiblyAffected**. *Then $Dom(w_d) = Dom_{pm}(w_d)$, where $Dom()$ is the dominator set computed using Algorithm 5.1 and $Dom_{pm}()$ is the dominator set computed using the exhaustive Purdom and Moore's algorithm by iterating over the nodes in $SubTree(idom(y_d))$.*

Proof:

First of all notice that the immediate dominator for nodes not in **PossiblyAffected** can never change. We will show that the Dom() set computed using our algorithm will be same as $Dom_{pm}()$ for nodes in **PossiblyAffected**.

Let $w \in$ **PossiblyAffected**, and let $p \in Pred_f(w)$. Recall that in the exhaustive Purdom and Moores' algorithm,

$$Dom_{pm}(w) = \{w\} \cup \cap_{p \in Pred_f(w)} Dom_{pm}(p)$$

From the above formula one can see that a node s_f can dominate w_f if only if s_f also dominates all the predecessor nodes p_f of w_f.

Also recall that we begin the iteration (in the exhaustive algorithm) by initializing the $Dom()$ set for all nodes to be all the nodes in the $SubTree(idom(y))$, except the node $idom(y)$ which is initialized to itself (a la overestimation). Using this observation, we prove the rest of the result.

Let $p_f \in Pred_f(w_f)$. We know that p is either in **PossiblyAffected**, **PseudoAffected**, or **NotAffected**. We will first show that for each category that p may belong to, its $Dom()$ set will be either a correct estimate or an overestimate of the actual (final) $Dom(p)$. In other words, if a node u dominates p it will be in the set $Dom(p)$.

1. $p \in$ **NotAffected**. It is obvious that $Dom(p)$ is a correct one.
2. $p \in$ **PseudoAffected**. Here $Dom(p) = Dom_{st}(p) \cup Dom_{dy}(p)$. The nodes in the static part will always dominate p. At step $\boxed{\textbf{25}}$ we assign the dynamic part to $Dom(u)$, where is an ancestor node of p and $u \in Aff$. The union of the two will be an overestimate. In other words, if a node s dominates p it will be in $Dom(p)$.
3. $p \in$ **PossiblyAffected**. The $Dom()$ will be an overestimate.

Now if a node s dominates all the predecessors p of w, it will appear in the $Dom()$ set of all the predecessors. Therefore s will also be in the intersection of the $Dom()$ set of all the predecessors. This means that s will dominate w.

Next we will show that the $Dom()$ of w will never increase. This easily follows by the property of intersection. Also, it will never be the case that all the predecessor nodes of w are in **PossiblyAffected**. (If this were the case, the intersection would never remove nodes from the $Dom()$ set initially estimated to be all of $SubTree(idom(y))$.)

Next we will show that when the algorithm terminates, the $Dom(w_d)$ will not contain any node u that will not dominate w. In other words, if u **!dom** w then $u \notin Dom(w)$ when the algorithm terminates. Assume to the contrary that u is in the $Dom(w)$, and u does not (in reality) dominate w. The only way u could be in $Dom(w)$ is because u is also in every $p \in Pred_f(w)$. This is not possible because then u would dominate w, contradicting the initial assumption.

∎

In the worst case, the time complexity of Algorithm 5.1 is same as the Purdom-Moore algorithm. In practice, however, we expect our algorithm to be much faster in the average case.

5.4 Updating Dominators For Reducible Flowgraphs

In the previous (sub) sections we presented an incremental version of Purdom and Moore's algorithm for the deletion case. This algorithm is applicable for maintaining the dominator tree of arbitrary flowgraphs.

If we restrict ourselves to reducible graphs, we can improve the time complexity of the deletion algorithm to $O(|E| \times log(|E|))$. The approach we take is similar to that of Ramalingam and Reps, except that we do not use priority scheme based on psuedo-circuit value problem. Our method uses properties of iterated dominance frontiers.

A key observation we made for reducibe graphs is that if we consider only the forward flowgraph then the following claim is true: Let u and v be any two nodes in the forward flowgraph then it is not possible that $u \in DF(v)$ **and** $v \in DF(u)$. Using this key point we can now easily order the nodes in the iterated dominance frontiers of a node w in the forward flowgraph as follows: Let u and v be any two nodes in the $IDF(w)$, then $u \prec v$ if $v \in IDF(u)$.

Now let us consider how to update the DJ-graph when an edge $x_f \rightarrow y_f$ is deleted. The first step is to compute the iterated dominance frontiers of the node y_f, such that all the nodes in $IDF(y_f)$ are at the same level. Now assign priorities to nodes in $IDF(y_f)$ such that a node u gets a lower priority than v if $u \prec v$. Once we have the priorities assigned to nodes we can use the method of Ramalingam and Reps to compute the new immediate dominators of all the affected nodes (step [18] to [24] in Figure 3.4 in their paper [RR94]).

Note that, unlike the method of Ramalingam and Reps method, we never incrementally maintain the priorities of the nodes in the forward flowgraph. We can compute the priorities on-the-fly in linear time. Also, their priority scheme is based on psuedo-circuit value problem due to Alpern et. al. [AHR$^+$90]. The time complexity of the above method is same as that of Ramalingam and Reps' algorithm (if we use Sleator and Tarjan's dynamic data structures).

6 Application: Incremental ϕ-Nodes Update

6.1 Motivation

In this section, we will use results obtained so far to design an incremental algorithm for updating the ϕ-nodes of an SEG. We are not aware of any published literature on this problem. Traditionally, an SEG is constructed by first computing the ϕ-nodes for an initial set of sparse nodes, denoted as N_α. These sparse nodes are called the non-identity transfer function nodes, since they have certain effects on their incoming data flow information. The set of ϕ-nodes, denoted as N_ϕ, are exactly the iterated dominance frontier of N_α. Recall that we can compute the iterated dominance frontier for an arbitrary set of nodes in linear time [SG94]. A node will appear in an SEG iff it is in $N_\alpha \cup N_\phi$. The construction of an SEG is completed by directly connecting nodes that produce information to nodes that either combine or consume information.

To illustrate the problem of incrementally updating the ϕ-nodes, let us consider the SEG shown in Figure 4(a). This SEG is derived from the flowgraph in Figure 1(a) with $N_\alpha = \{2, 7\}$. Now if we insert a new flowgraph edge from 2 to 4, the SEG also needs to change accordingly. This is because node 2 is in the SEG, and $IDF(2)$ changes. Before $2 \to 4$ is inserted, $IDF(2) = \{\text{END}\}$; after that $IDF(2) = \{2, 3, 4, 6, 7, 8, \text{END}\}$. Notice that we include more nodes in $IDF(2)$ in response to the edge insertion. The situation is opposite when an edge is deleted.

In this paper we focus on cases where N_α is not a part of the change. For example, the *edge placement* problem is such an application. Also, without loss of generality we will focus on cases where both x and y are reachable before and after updates. We briefly discuss the extension to more general situations at the end of this section.

6.2 Insertion of an Edge

In this subsection we discuss the effect of inserting a new flowgraph edge on the ϕ-nodes of an SEG. Let $G_s = (N_s, E_s)$ be an SEG. Suppose $x_f \to y_f$ is a newly inserted edge in the flowgraph. Then we want to incrementally update, if needed, the ϕ-nodes for G_s in order to reflect the flowgraph change. It is possible that inserting a new edge in a flowgraph may not affect an SEG. How do we know that inserting $x_f \to y_f$ will indeed affect an SEG? Furthermore, if this is the case, then how do we find the new set of ϕ-nodes for the SEG?

Consider the DJ-graph shown in Figure 2, where we can see $8 \notin DF(3)$ (because 3 **sdom** 8). After inserting $2 \to 4$ in the flowgraph, we update the DJ-graph. The updated DJ-graph is shown in Figure 3, where we can see $8 \in DF(3)$ (because 3 no longer strictly dominates 8). What this example illustrates is: When a node, say w_d, is pulled up in the DJ-graph, we may have to additionally include w in the dominance frontiers of some of the nodes previously strictly dominated w. Conversely, one could ask the following question: Is

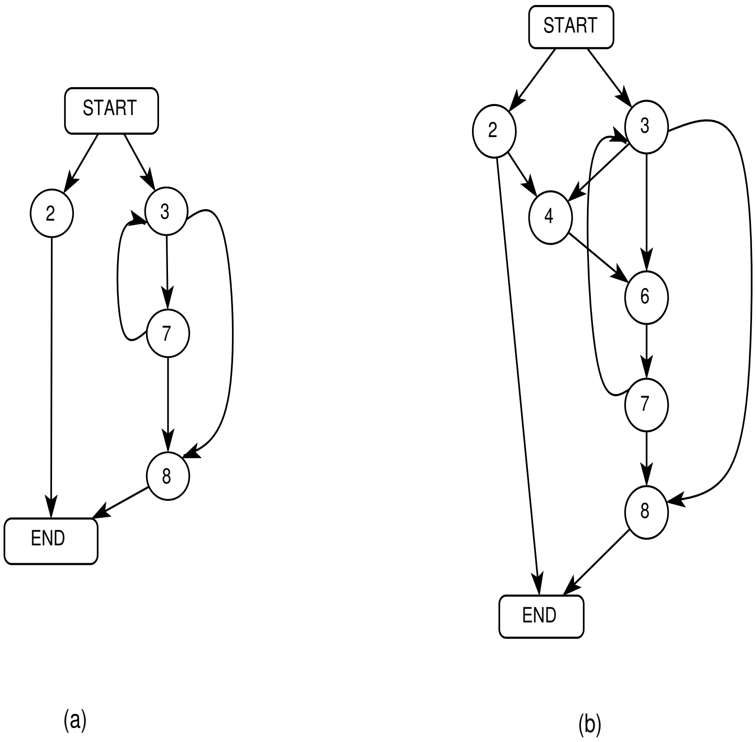

Figure 4: SEG before and after insertion of the flowgraph edge

it sufficient to look at only nodes in the $\mathsf{DomAffected}_{\mathsf{I}}(y)$? The answer is yes. The reason for this is as follows: By definition we know if v strictly dominate u, u can never be in the $DF(v)$. Therefore if v **sdom** u before updating the DJ-graph (due to insertion) and v **!dom** u after updating the DJ-graph, then v must be included in the dominance frontier of u after the update.

Also notice that by inserting $x_f \to y_f$, y_f *may* have to be included in the dominance frontiers of x (and some of its ancestor nodes). If x strictly dominates y before and after updates, then we never include y in $DF(x)$. This case can happen only when x is a proper ancestor of y before the update (so $nca(x_d, y_d) = x_d$). Otherwise, we include $y \in DF(x)$. In this case we also have to include y in the dominance froniter of all the nodes that dominate x, but does not strictly dominate y itself (for obvious reasons).

One important observation to notice is that whenever a new edge is inserted the iterated dominance frontier set of a node never decreases.[5] This means that by inserting a new edge, we never remove ϕ-nodes from a SEG; either the set of ϕ-nodes (for a SEG) increases or remains

[5]This is not true with dominance frontiers of a node. By inserting a new edge, it is possible for the dominance frontier set to increase or decrease.

the same.

Using the above key observations, we give a simple method for updating ϕ-nodes that can be implemented as part of the DJ-graph update algorithm.

Before actually pulling up a node w_d at step $\boxed{\mathbf{11}}$ of Algorithm 4.1, we use the *Phi* flag to mark every node u_d (excluding w_d and z_d) on the reverse tree path from w_d to z_d, indicating that the iterated dominance frontiers for these marked nodes will change after the update. We also do this for x_d, if x_d is not an ancestor of y_d. If a node v_d is already marked *Phi*, we need not visit the ancestors of v_d, because they would have already been marked *Phi* by some previous visit. After the DJ-graph is updated, the (iterated) dominance froniters for all the marked nodes will change. So if a node u that is marked *Phi* is in a SEG, we have to include $IDF(u)$ in the SEG.

6.3 Deletion of an Edge

The effect of deleting a flowgraph edge on an SEG is complementary to the effect of inserting an edge. Let $x_f \rightarrow y_f$ be the deleted flowgraph edge. We outline our ϕ-nodes update approach to handling an edge deletion as follows: The first step is to update the DJ-graph using Algorithm 5.1. In the end of the DJ-graph update, we also find the set $\mathsf{DomAffected}(y_d)$, which includes all the nodes *actually* pulled down. Let $w_d \in \mathsf{DomAffected}(y_d)$. When w_d is pulled down, we must exclude w_d from the dominance frontiers of nodes that will strictly dominate w_d after the update (but did not before the update). Also, when $x_f \rightarrow y_f$ is deleted, y_f may no longer be in the dominance frontiers of nodes that dominated x but did not strictly dominate y before the update. Let us use the *Phi* flag to mark these nodes. Consequently, we may have to remove from the SEG some ϕ-nodes that were previously in the iterated dominance frontiers of the marked nodes. But before actually deleting any of such nodes from the SEG, we must make sure it is neither in the initial set of sparse nodes nor in the dominance frontier of another node that is not affected. Based on the above intuition, one can easily devise a simple algorithm to remove ϕ-nodes from a SEG when an edge $x_f \rightarrow y_f$ is deleted.

6.4 Discussion

In this paper we have mainly focused on cases where there are no changes to N_α. This case is important and arises in practical optimizations, like the edge placement problem. Also, our solution to this restricted case forms a foundation upon which we intend develop a new framework for incremental analysis based on SEGs. In this framework we intend to include all sorts of flowgraph updates. This framework also subsumes the (open) problem of incrementally updating SSA forms. Many aggressive transformations are based on SSA form, and incrementally maintaining SSA form is very important to speed-up the whole compilation process. The incremental analysis based on SEG is a subject of future research, and will not be discussed further in this paper.

7 Related Work

In this paper we have presented solutions to the DJ-graph update and the ϕ-nodes update problems. The DJ-graph update problem subsumes that of updating dominator trees. Previously work, most relevant to our approach, for the dominator tree update problem includes the Carroll-Ryder algorithm [CR88] and the Ramalingam-Reps algorithm [RR94]. Both these algorithms are restricted to the class of *reducible* flowgraphs. By contrast, our algorithm can handle *irreducible* as well as reducible flowgraphs.

The Carroll-Ryder algorithm uses two local properties: *niceness* and *deepness*. Niceness is a property for edges; deepness is a property for nodes. Non-nice edges and non-deep nodes cannot exist in the dominator tree of a reducible graph. Using these two local properties and the notion of *representative edges*, they maintain the dominator tree of a reducible flowgraph. However, it could be expensive to maintain such auxiliary information as representative edges. Ramalingam and Reps have presented detailed comparisons between their work and the Carroll-Ryder algorithm. Their comparisons are applicable to our work. We direct interested readers to their paper [RR94].

Ramalingam and Reps [RR94] use properties of the pseudo-circuit value problem to assign priorities to nodes during updates [AHR+90]. Priorities can be assigned to nodes only if the *forward flowgraph*[6] is a directed acyclic graph. This property is only true for reducible flowgraphs. During the insertion of a new edge, if a cycle is detected in the forward flowgraph, then their algorithm immediately signals that this insertion has introduced irreducibility into the flowgraph. After that they do not allow further insertions and deletions of edges, because they cannot assign priorities to nodes in the forward flowgraph containing cycles.

Our approach is comparable to Ramalingam and Reps', but with a number of significant differences. First, our dominator tree update algorithm is more general and is not restricted to only reducible flowgraphs.

Second, in the insertion case, our algorithm can find the precise set of affected nodes, but the Ramalingam-Reps algorithm cannot. They need to use a conservative set of possibly affected nodes. When an edge $x_f \to y_f$ is deleted, Ramalingam and Reps make all the sibling nodes of y_d be affected. Our set of possibly affected nodes, although not precise, is always a subset of y_d's sibling nodes; therefore, it is a better approximation than theirs.

Third, in the worst case, the time complexity of our algorithm for the insertion case is $O(E)$, while that of Ramalingam and Reps' algorithm is $O(E \times \log E)$. Recall that, since we are handling updates of DJ-graphs, it is crucial that we update the levels of nodes too. This means that we definitely have to visit all the descendants of the affected nodes to update their level numbers. This will also be true for Ramalingam and Reps' algorithm, if they too update the level information in the dominator tree. For the deletion case, although ours is slower

[6]Intuitively, a forward flowgraph is nothing but a reducible flowgraph with all 'back edges' removed. An edge in a flowgraph is a back edge if its destination node dominates its source node [ASU86].

than Ramalingam and Reps', our new algorithm is likely to improve the running time on the average cases.

In a recent paper [JPP94], Johnson et al. introduced the Program Tree Structure (PST) for performing fast program analysis. The PST represents a program with a hierarchy of single-entry, single-exit regions. Using the PST, Johnson in his thesis proposed an algorithm for updating dominator trees [Joh94]. His approach is to identify regions, corresponding to sub-flowgraphs, where the dominance relation may no longer be correct because of a flowgraph edge update. Once a region is identified, he applies an exhaustive algorithm for all the nodes in this region. Obviously, in the worst-case there can be only one node in a PST, so that they must exhaustively recompute the dominance relation for all the nodes in the flowgraph. Compared to Johnson's work, ours is truly an incremental algorithm in nature. We update the dominance relation only for the (possibly) affected nodes.

For the ϕ-nodes update problem, we are not aware of any published work that provides a general solution for arbitrary SEGs. In a recent technical report [CSS94], Choi et al. proposed an approach to incremental computation of SSA form. They can maintain the SSA form of a program in response to some specific loop transformations and the insertion/deletion of variable definitions/uses. Their approach cannot handle arbitrary flowgraph changes, but can be very efficient in practice because it is tailored to particular transformations. It would be interesting to investigate how to emulate a typical loop transformation as a series of edge insertions/deletions.

8 Conclusion and Future Work

We have presented our algorithms for incrementally maintaining DJ-graphs (and thus dominator trees) and for incrementally updating the ϕ-nodes for an arbitrary SEG. Our approach to incremental dominator tree update has the advantage over previous approaches in that it can handle *irreducible* flowgraphs. Our algorithm for the ϕ-nodes update problem is the first proposal to handling *arbitrary* SEGs. We believe SEGs are suitable for incremental data flow analysis. In the future we will apply and/or modify existing incremental algorithms operated on SEGs to design an SEG-based incremental data flow analysis technique.

Acknowledgement

The first and second authors thank the National Sciences and Engineering Research Council (NSERC) and the Canadian Centers of Excellence (IRIS) for their continued support of this research. We also thank Tom Marlowe for his insightful comments.

References

[AHR+90] B. Alpern, R. Hoover, B. Rosen, B. K. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42, 1990.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers–Principles, Techniques, and Tools*. Addison-Wesley Publishing Co., 1986.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, 1991.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method for computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, Texas, January 11–13, 1989. ACM SIGACT and SIGPLAN.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):452–490, October 1991.

[CLZ86] Ron Cytron, Andy Lowry, and Kenneth Zadeck. Code motion of control structures in high-level languages. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–85, St. Petersburg Beach, Florida, January 13–15, 1986. ACM SIGACT and SIGPLAN.

[CR88] M. Carroll and B. G. Ryder. Incremental data flow update via attribute and dominator updates. In *ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, pages 274–284, January 1988.

[CSS94] J. D. Choi, V. Sarkar, and E. Schonberg. Incremental computation of static single assignment form. Unpublished manuscript, 1994.

[Joh94] Richard Johnson. *Dependence Based Program Analysis*. PhD thesis, Cornell University, Ithaca, New York, August 1994.

[JPP94] R. Johnson, D. Pearson, and K. Pingali. The program tree structure: Computing control regions in linear time. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 171–185, 1994.

[RR94] G. Ramalingam and T. Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, January 1994.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, pages 12–27, January 1988.

[SG94] Vugranam C. Sreedhar and Guang R. Gao. Computing ϕ-nodes in linear time using DJ-graphs. Technical Report ACAPS Memo 75, School of Computer Science, McGill University, January 1994.

[SGL94] Vugranam C. Sreedhar, Guang R. Gao, and Yong-fong Lee. DJ-graphs and their applications to flowgraph analyses. Technical Report ACAPS Memo 70, McGill University, May 1994.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, January 1994.

[WZ85] Mark Wegman and Ken Zadeck. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 291–299. ACM SIGACT and SIGPLAN, January 1985.

A Purdom and Moore's Algorithm

Here we give the original Purdom and Moore's exhaustive algorithm for computing the dominator set. The version of the algorithm is adapted from [ASU86].

Algorithm A.1 (Finding Dominators.) *The following is an iterative algorithm for computing the dominator set.*

$Dom(\text{START}) = \{\text{START}\}$
foreach $n \in N - \{\text{START}\}$ **do**
  $Dom(n) = N$
`/* end initialization */`
while *Changes* to any $Dom(n)$ occur **do**
  foreach $n \in N - \{\text{START}\}$ **do**
    $Dom(n) = \{n\} \cup \cap_{p \in Pred(n)} Dom(p)$

B Sreedhar and Gao's Algorithm

Here we briefly introduce Sreedhar and Gao's linear time algoritm [SG94] for computing the iterated dominance frontiers for a set of node S. The algorithm is based on two key observations:

1. Let y be an ancestor of x on the dominator tree. If $DF(x)$ has already been computed before the computation of $DF(y)$, then $DF(x)$ need not be recomputed when computing $DF(y)$. However, the reverse may not be true; therefore the order of the computation is crucial. We order the computation of the relevant set of nodes in a bottom-up fashion using the levels of the nodes in the dominator tree.

2. The computation of $DF(x)$ itself is guided by D-edges in a top-down fashion. When computing $DF(x)$ we only need to examine the J-edges $y \to z$, where y is a node in the dominator sub-tree rooted at x and z is a node whose level is no greater than the level of x.

To order the relevant set of nodes we use a data structure called the *PiggyBank*, which is nothing but an array of pointers to nodes, with index i storing nodes at level i.

An high-level description of the complete algorithm is given below (for details please see [SG94]). Let S be the initial set, whose iterated dominance frontiers we are interested. Initially the *PiggyBank* is empty.

Step 1 Deposit all nodes in S into the *PiggyBank*. Remember that if i is the level number of a node then it is inserted at the index i in the *PiggyBank*.

Step 2 Pick a node w from the *PiggyBank* such that $w.level$ is the highest level of all the nodes currently in the *PiggyBank*. (Remove w from the *PiggyBank*.)

Step 3 Compute $DF(w)$ using Lemma 2.1 by walking down the $SubTree(w)$ while avoiding nodes that have been previously visited. Deposit all the nodes u in $DF(w)$ back into the *PiggyBank*, if u was not previously put in the *PiggyBank* and u is not in S. All such u's will be the candidate nodes for $IDF(S)$.

Step 4 Repeat Step 2 and 3 (in that order), until there are no more nodes in the *PiggyBank*.