An Incremental Algorithm for Maintaining the Dominator Tree of a Reducible Flowgraph

G. Ramalingam
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598
rama@watson.ibm.com

Thomas Reps
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, WI 53706
reps@cs.wisc.edu

Abstract

We present a new incremental algorithm for the problem of maintaining the dominator tree of a reducible flowgraph as the flowgraph undergoes changes such as the insertion and deletion of edges. Such an algorithm has applications in incremental dataflow analysis and incremental compilation.

1. Introduction

The contribution of this paper is a new incremental algorithm for the problem of maintaining the dominator tree of a reducible flowgraph as the flowgraph undergoes changes such as the insertion and deletion of edges. The dominator tree plays an important role in several algorithms for program analysis and program optimization, and the need for updating the dominator tree of a flowgraph arises in various contexts. For instance, Carroll and Ryder [Car88] present an incremental dataflow analysis algorithm that makes use of dominator trees—it is necessary as a part of this algorithm to update the dominator tree of the flowgraph. The need to update the dominator tree can arise even in the context of batch compilation. For instance, dominator trees are used in the construction of the static single assignment (SSA) representation of programs [Cyt89, Cyt91]. As an optimizing compiler repeatedly applies optimizing transformations, it may be necessary to update the SSA representation of the program after each transformation.

The algorithm presented in this paper works only for reducible flowgraphs, but does identify edge insertions that introduce irreducibility. The idea behind our algorithm is to formulate the problem of constructing the dominator tree of a reducible flowgraph as a "pseudo" circuit value problem—the circuit value problem is essentially the problem of computing the fixed point of a collection of non-recursive equations [Alp90]. This lets us borrow some ideas from previous incremental algorithms for the circuit value problem. However, as we show in Section 3.1, the problem of constructing the dominator tree differs considerably from the circuit value problem and, consequently, we cannot directly use an incremental circuit evaluation algorithm for the dominator-tree update problem. What we do use is an incremental algorithm due to Alpern *et al.* [Alp90] for maintaining priorities in the dag induced by the forward edges of the flowgraph. Another key aspect of our algorithm is that the dominator tree is maintained using the link-cut-tree data structure due to Sleator and Tarjan [Sle83]. This dynamic data structure permits a number of operations to be performed efficiently.

The only previous algorithm for the problem of maintaining the dominator tree of a (reducible) flowgraph is due to Carroll and Ryder [Car88]. The algorithm given in this paper has a better worst-case complexity than the Carroll-Ryder algorithm. There are also good reasons to believe that our algorithm will be more efficient in practice also. A comparison of the two algorithms is given in Section 4.

The remainder of this paper is organized as follows. In Section 2 we review relevant terminology and notation. In Section 3 we present the incremental algorithm. We conclude in Section 4 with a comparison with previous work and a discussion of possible extensions.

2. Terminology and Notation

A ***flowgraph*** is a directed graph with a source vertex. We initially consider flowgraphs in which every vertex is reachable from the source vertex. A vertex u is said to ***dominate*** a vertex v in a flowgraph with source s iff every path from s to v passes through u. If u dominates v and $u \neq v$ then u is said to be a ***proper dominator*** of v. Domination is a special kind of relation that can be concisely represented by a tree, the ***dominator tree***, in which a vertex u is an ancestor of a vertex v iff u dominates v. The parent of a vertex u in the dominator tree will be referred to as the immediate dominator of u and will be denoted by ***idom(u)***.

An edge $x \longrightarrow y$ in a flowgraph is said to be a ***back edge*** if vertex y dominates vertex x and it is said to be a ***forward edge*** otherwise. A flowgraph is said to be a ***reducible flowgraph*** if the set of all forward edges induces an acyclic graph. Otherwise, it is said to be ***irreducible***. (There are other equivalent characterizations of reducibility—see [Aho86].)

This work was done while the first author was at the University of Wisconsin, and was supported in part by a David and Lucile Packard Fellowship for Science and Engineering, by the National Science Foundation under grant CCR-9100424, and by the Defense Advanced Research Projects Agency under ARPA Order No. 8856 (monitored by the Office of Naval Research under contract N00014-92-J-1937).

The usual definition of a reducible flowgraph assumes that all vertices are reachable from the source vertex. We relax this restriction and consider a flowgraph to be a reducible flowgraph if the set of vertices reachable from the source is a reducible flowgraph according to the above definition. This relaxation is useful in the context of incremental computation—in changing one reducible flowgraph into another reducible flowgraph by inserting and deleting edges it may be necessary to temporarily introduce unreachable vertices. However, the concept of domination still applies only to reachable vertices. In particular, the algorithm we outline in this paper will maintain the dominator tree of the subgraph induced by the reachable vertices. Also, the classification of an edge as a back edge or forward edge is meaningful only if the endpoints of the edge are reachable. By the "status" of an edge we mean its status as a back edge or forward edge.

Recall that an edge $u \longrightarrow v$ is a back edge iff v dominates u. Thus, any path from the source vertex s that contains a back edge $u \longrightarrow v$ must be a cyclic path since it contains at least two occurrences of the vertex v. This implies that the set of reachable vertices does not change if a back edge $u \longrightarrow v$ is removed from the graph, since a vertex is reachable iff there exists an acyclic path from the source vertex to that vertex. Similarly, the domination relation of a flowgraph does not change if a back edge is removed from the graph, since domination can be defined in terms of acyclic paths. Hence, the problems of maintaining the reachability information and the dominator tree of a reducible flowgraph are closely related to the problems of maintaining the reachability information and the dominator tree of a dag (the dag of forward edges), provided we have a way of identifying the forward edges and back edges of the graph. We now show that the status of an edge as a forward edge or back edge does not change as edges are inserted into and deleted from the graph as long as the graph remains reducible throughout this sequence of modifications. (We are talking only of edges whose endpoints remain reachable in the flowgraph.)

Proposition 2.1. Let G_1 and G_2 be two reducible flowgraphs such that $E(G_2) = E(G_1) \cup \{u \longrightarrow v\}$. The status of an edge that is reachable in both graphs is the same.

Proof. It follows easily from the definition of domination that the insertion of an edge can only shrink the domination relation—hence, if x dominates y in graph G_2 then x must dominate y in graph G_1, too. Consequently, a back edge in graph G_2 must be a back edge in graph G_1 too (assuming that the edge under consideration is not $u \longrightarrow v$). Equivalently, every forward edge in G_1 is a forward edge in G_2.

Going the other way, a back edge in G_1 must be a back edge in G_2 also, for the following reason: if $x \longrightarrow y$ is a back edge in G_1, then there is some path consisting only of forward edges from y to x in G_1; hence, there exists a path of forward edges from y to x in G_2 also; if $x \longrightarrow y$ were a forward edge in G_2, then the set of forward edges in G_2 would induce a cycle, contradicting the assumption that G_2 is reducible. □

Let us now consider the status of a newly inserted edge $u \longrightarrow v$. Since domination can be defined in terms of acyclic paths, insertion of an edge $u \longrightarrow v$ does not change the set of dominators of vertex u. Consequently, $u \longrightarrow v$ is a back edge iff v dominates u in the new graph iff v dominates u in the old graph. This is useful since we can determine the status of a newly inserted edge from the domination information about the original graph.

3. The Dynamic Dominator Tree Problem in Reducible Flowgraphs

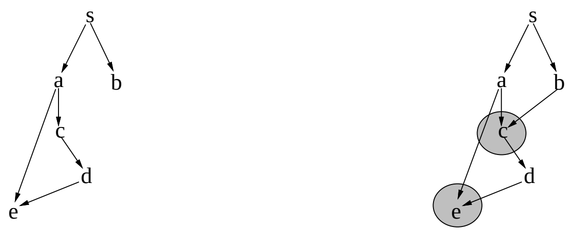
In this section we present an incremental algorithm for maintaining the dominator tree of the subgraph of reachable vertices of a flowgraph under the assumption that this subgraph remains reducible as the flowgraph undergoes modifications. We also outline a method for checking if the graph ever becomes irreducible during a sequence of modifications.

3.1. The Dominator Tree of a Directed Acyclic Graph

We saw in Section 2 that the dominator tree of a reducible flowgraph is the same as that of the dag of forward edges of that flowgraph. We will first present a simple batch algorithm for constructing the dominator tree of a dag. Linear time algorithms that construct the dominator tree of a dag are known [Har85, Och83], but the reason for the following presentation is that it suggests a possible way of incrementally maintaining the dominator tree of a dag. We will then use this idea for maintaining the dominator tree of a reducible flowgraph.

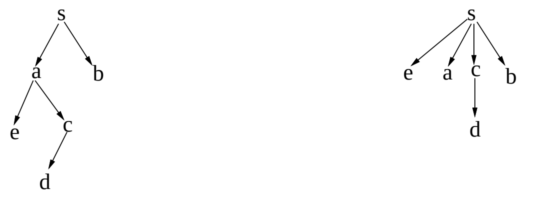
Consider a dag with a source vertex. Consider a vertex u in the dag with predecessors $v_1, \ldots, v_k$. A vertex w will properly dominate u iff it dominates all the vertices v_1 through v_k. In other words, $Dom(u) = \{u\} \cup \bigcap_{i=1}^{k} Dom(v_i)$, where $Dom(x)$ denotes the set of dominators of vertex x. Thus, if we have identified the set of dominators of v_1 through v_k, then the set of proper dominators of u can be obtained by intersecting these sets. In particular, the immediate dominator of u has to be the least common ancestor of v_1 through v_k in the dominator tree.

The previous paragraph suggests the following scheme for constructing the dominator tree of a dag in an incremental fashion. The vertices in the dag are visited in topological order and added to the dominator tree one by one. Initially the dominator tree consists only of the source vertex s. When a vertex u is visited, the least common ancestor w of all the predecessors v_1 through v_k of u in the partially constructed tree is identified. Now, u is made a child of w. Thus, the construction of the dominator tree of a dag can be



G

G’, obtained by inserting an edge into G



The dominator tree for G

The dominator tree for G’

Figure 3.1. An example to show how the problem of constructing the dominator tree of a dag differs from a circuit value problem. An edge $b \longrightarrow c$ is inserted into the dag G. The affected vertices in the resulting dag G', indicated by the shaded region, do not form a connected region.

considered to be a “pseudo” circuit value problem[1], where every vertex u other than the source vertex is associated with the equation

$$parent(u) = lca(v_1, \ldots, v_k),$$

where we use *lca* as an abbreviation for “least common ancestor”. The value computed for every vertex (other than the source) is its parent in the dominator tree.

Note that the above problem is not a true circuit value problem because the right-hand side of the above equation is not a “pure function” of the values associated with the predecessors of vertex u. Hence, the output value of a vertex (that is, its immediate dominator) can change even though the output value of none of its predecessors have changed. Consider, for example, the dag G shown in Figure 3.1. This dag is modified into dag G' by the insertion of an edge $b \longrightarrow c$. The vertex c is affected. (We say a vertex is affected if its immediate dominator changes.) The only successor of c, namely d, is not affected—its immediate dominator is still c. However, vertex e is affected, even though none of its predecessors are affected.

The above problem prevents us from using any incremental circuit evaluation algorithm directly. However, if we have a conservative approximation to the set of all affected vertices, then we can use ideas from the circuit value problem to update the dominator tree. In particular, we can visit all vertices that might possibly be affected in topological order, and determine their immediate dominators using the above equation. We can use priority-ordering to visit vertices in a topological sort order, and use the incremental algorithm of Alpern *et al.* [Alp90] for updating the priority-ordering when the graph undergoes changes. (A priority-ordering associates a priority with every vertex of a dag such that if there is a path from a vertex u to a vertex v then $priority(u) < priority(v)$.) We can represent the dominator tree using the link-cut tree data structure: The link-cut tree data structure, due to Sleator and Tarjan [Sle83], is a representation of a dynamic forest (collection of trees) that allows a number of operations to be performed efficiently. The only operations that are of interest to us are: a *maketree* operation that creates a new tree consisting of a single newly inserted vertex; a *link* operation that adds an edge from some vertex u in a tree T_1 to the root of another tree T_2, making T_2 a subtree of T_1; a *cut* operation that removes an edge from a tree, breaking it into two trees; an operation that returns the least common ancestor of two vertices, which can also be used to check for the ancestor-descendant relationship between two vertices. Each of these operations runs in $O(\log n)$ time, where n is the number of vertices in the forest.

---

[1] In the circuit value problem the input consists of a dag in which every vertex has an associated equation that specifies the output value to be computed for that vertex as a function of the output values of its predecessor vertices.

3.2. The Dynamic Dominator Tree Problem for Reducible Flowgraphs

We now return to the problem of maintaining the dominator tree of a reducible flowgraph. Let G denote the reducible flowgraph for which the dominator tree has to be maintained. The algorithm will maintain the following data structures and information: (a) The reachability status of every vertex will be maintained. (b) Let $R(G)$ denote the subgraph induced by the reachable vertices. The status of every edge in $R(G)$—whether it is a forward edge or back edge—will be maintained. (c) Let $F(G)$ denote the acyclic subgraph of $R(G)$ induced by the forward edges. Both $F(G)$ and a correct prioritization of $F(G)$ will be maintained. (d) The dominator tree $DT(G)$ of $F(G)$, which is the same as the dominator tree of $R(G)$ and G, will be maintained as a link-cut tree.

3.2.1. The Insertion of an Edge

We now consider the problem of updating all the above information when an edge $u \longrightarrow w$ is inserted into the graph. We will assume that the vertex u was originally reachable from the source vertex, since nothing needs to be done otherwise. We will first consider the simpler case where the vertex w was already reachable, which means that there is no change in the reachability status of vertices.

A Special Case: No Change in Reachability

In this case the set of vertices in $R(G)$ remains the same, while $u \longrightarrow w$ is added to the set of edges. We know from Proposition 2.1 that the insertion of $u \longrightarrow w$ does not change the status of any other edge in $R(G)$. Consequently, maintaining edge statuses requires only that we determine the status of the newly inserted edge. If vertex w dominates vertex u in the original graph, then the edge inserted is a back edge. Otherwise, it is a forward edge. We can check if v dominates u in the original graph in $O(log\ n)$ time, since a representation of the dominator tree as a link-cut tree enables us to check for the ancestor-descendant relation between two vertices in $O(log\ n)$ time.

If the edge $u \longrightarrow w$ is a back edge, then neither $F(G)$ nor $DT(G)$ changes, and nothing more needs to be done. If the edge is a forward edge, then we insert it into $F(G)$, and update the prioritization of $F(G)$ using the algorithm presented in [Alp90]. This algorithm also determines if the insertion of an edge into a dag introduces a cycle. Hence, we can determine at this point if the insertion of the edge $u \longrightarrow w$ makes the graph G irreducible. Updating the priorities takes $O(\gamma \log \gamma)$ time, where γ is a measure of the size of the change in the priorities. (A dag can have a number of valid priority-orderings. Consequently, when the dag is modified, its priority-ordering can be updated in a number of different ways, each of which might require updating the priorities of a different set of vertices. γ is the "size" of the "smallest" set of vertices updating whose priorities yields a valid priority-ordering, where the "size" of a set of vertices is the sum of the number of vertices and the number of edges incident on those vertices.)

We now consider the problem of updating $DT(G)$ itself. Let us now consider how the dominator tree can change following the insertion of a forward edge into the flowgraph.

Proposition 3.1. (See Figure 3.2.) Consider the insertion of an edge $u \longrightarrow w$ into a flowgraph where both u and w are already reachable. If a vertex y is affected by the insertion of the edge, then $idom(y)$ must properly dominate w and $idom(y)$ must be properly dominated by $lca(u,w)$. Further, the new immediate dominator of every affected vertex must be $lca(u,w)$.

Proof. We have assumed that vertex w was already reachable in the flowgraph. Hence, w must occur in the dominator tree of the original flowgraph. Let v denote $idom(w)$. Consider the least common ancestor of u and w. Consider an affected vertex y, that is, a vertex y whose immediate dominator changes. Let x denote $idom(y)$ in the original graph. We noted earlier that the insertion of an edge can only shrink the domination relation. Hence, y's new immediate dominator must have been a dominator of y in the original graph too—that is, y's new immediate dominator must be an ancestor of y in the original dominator tree.

What can we infer from the fact that x no longer dominates y? The insertion of the edge $u \longrightarrow w$ must have created a path from s to y that avoids x. It follows that the original graph must contain a path P_1 from s to u and a path P_2 from w to y both of which avoid vertex x. This implies that x cannot have been a dominator of u in the original graph, since otherwise the required path P_1 could not have existed. This also implies that x must have been a proper dominator of w in the original graph—otherwise, there must exist a path P_3 in the original graph from s to w that avoids x. Concatenating paths P_3 and P_2 yields a path from s to y in the original graph that avoids x, contradicting the assumption that x dominates y in the original graph.

Hence, x must be a proper ancestor of w, but cannot be an ancestor of u. In other words, x must be a proper ancestor of w and a proper descendant of $lca(u,w)$—that is, it must lie in the shaded region S shown in Figure 3.2. This establishes the first claim in the proposition.

Now, consider the second claim. Assume that y is an affected vertex. Note that $lca(u,w)$ dominates y even in the new graph, since there exists no path from s to u that avoids $lca(u,w)$. On the other hand, no ancestor t of y in the original dominator tree that is a proper descendant of $lca(u,w)$ can dominate y in the new graph—there exists a path P_2 from w to y that avoids t (since y is assumed to be an affected vertex), and there exists a path P_1 from s to u that avoids t. Consequently, the new immediate dominator of an affected vertex y must be $lca(u,w)$. □

For any two vertices p and q define $PossiblyAffected(p,q)$ to be the set $\{ r \mid idom(r)$ is a proper ancestor of q and a proper descendant of $lca(p,q) \}$. The above proposition shows that $PossibleAffected(u,w)$ is a

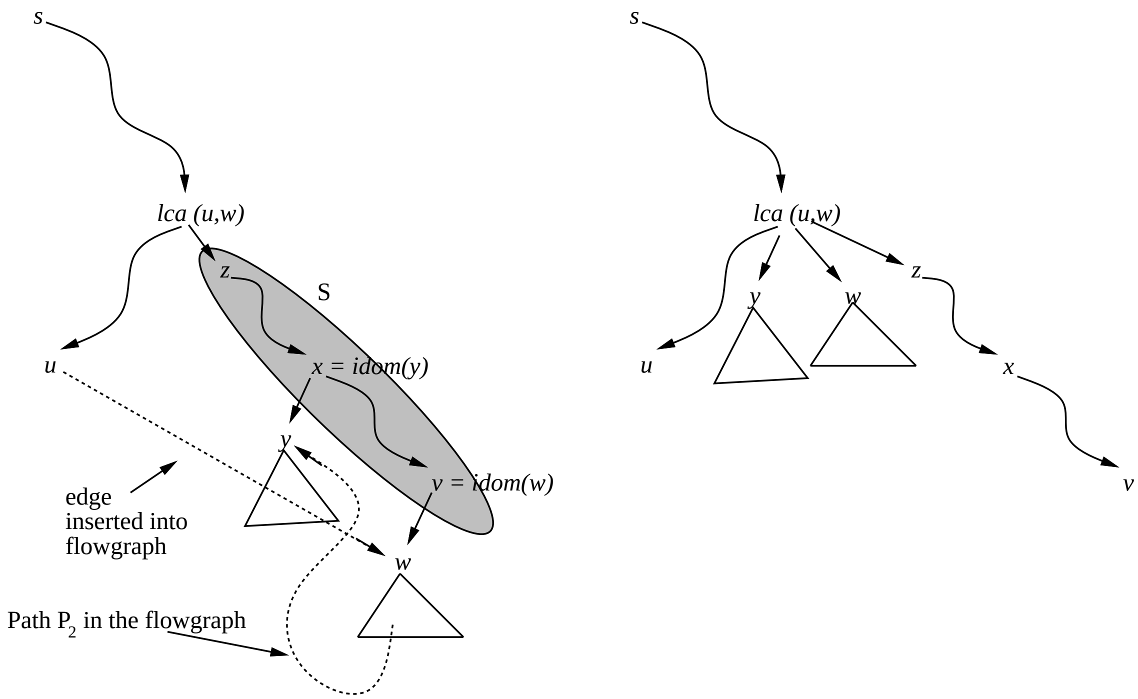


Figure 3.2. The change in the dominator tree of a flowgraph when an edge $u \longrightarrow w$ is inserted. Solid wavy lines in the figure indicate paths in the dominator tree, while solid straight lines indicate edges in the dominator tree. Dashed lines indicate edges and paths in the flowgraph. For any affected vertex y, $idom(y)$ must lie in the shaded region of the original dominator tree. Further, the new immediate dominator of any affected vertex is $lca(u,w)$.

conservative approximation to the set of affected vertices when an edge $u \longrightarrow w$ is inserted, provided both u and w are reachable in the original graph. We can “re-evaluate” all these vertices in increasing order of priority to determine the set of affected vertices and to update the dominator tree. The “re-evaluation” of vertices in $PossibleAffected(u,w)$ is necessary only to determine the vertices that are actually affected, since, as the above proposition shows, the new immediate dominator of every affected vertex y is $lca(u,w)$. This step takes time $O(\| PossibleAffected(u,w) \|^{\leftarrow} \log n)$, where, for any set X, $\| X \|^{\leftarrow}$ denotes the sum of the number of vertices in X and the number of edges coming into vertices in X.

The General Case

Consider the insertion of an edge $u \longrightarrow w$ where u is reachable but w may or may not be reachable. This can be processed very easily using a worklist algorithm that repeatedly invokes the special-case updating algorithm, as follows. Create a worklist that initially consists of just $u \longrightarrow w$. Every edge $x \longrightarrow y$ in the worklist will be processed as follows: If y is already marked reachable, we process edge $x \longrightarrow y$ using the special-case algorithm previously outlined. (Except for the particular case where the edge $x \longrightarrow y$ is the newly inserted edge $u \longrightarrow w$, the edge $x \longrightarrow y$ would have already been in the graph, but would not have been previously processed since vertex x was previously unreachable. Hence it is being processed now.) If y is marked unreachable then we do the following: y is marked reachable; the edge $x \longrightarrow y$ is marked as a forward edge; y is assigned a priority greater than x; y is made a child of vertex x in the dominator tree; and, finally, all the edges going out of y are added to the worklist.

However, it is possible to do the update somewhat more efficiently in the general case. Observe that the edges processed in the above worklist algorithm are of two types: edges of the form $x \longrightarrow y$ where y was already reachable, and edges of the form $x \longrightarrow y$ where y was not previously reachable. The algorithm presented below first identifies all the edges and vertices that need to be processed, and partitions them into these two classes, and processes them separately.

The algorithm is presented in Figure 3.4. The major steps involved in the algorithm are: (1) Determining the set R of vertices that become reachable. (2) Processing the subgraph $<R>$ induced by R. (3) Determining the set X of

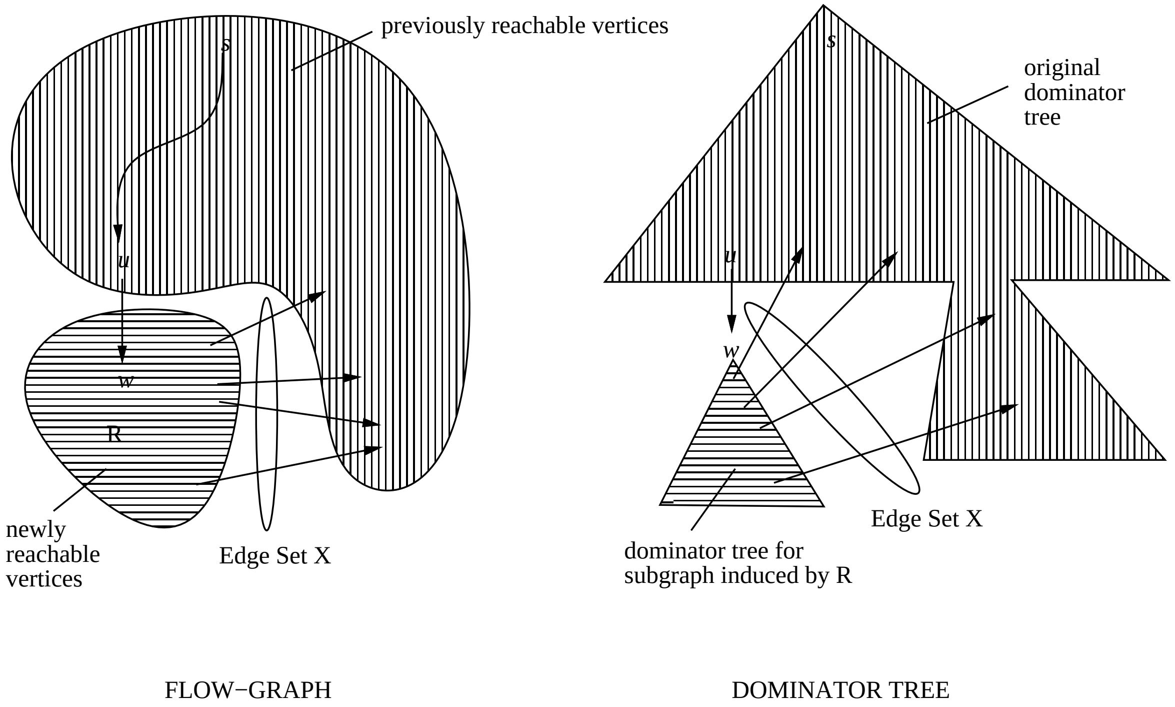


Figure 3.3. The change in the dominator tree of a flowgraph when an edge $u \longrightarrow w$ is inserted such that w becomes reachable. The flowgraph is shown on the left side. The tree on the right side shows how the new dominator tree would look like *if there was no edge from a newly reachable vertex to a previously reachable vertex—that is, if set X was empty.* If X is not empty, then the new dominator tree can be obtained from the dominator tree on the right by "processing the insertion of edges in F".

edges of the form $x \longrightarrow y$, where $x \in R$ and $y \notin R$. (4) Processing the set X of edges.

The set R of vertices that become reachable is obtained easily enough from a simple graph traversal starting from w. Processing the subgraph $<R>$ can be done using a batch algorithm as follows: The dominator tree D of $<R>$, with w as the source vertex, is computed using a batch algorithm. The status of edges in $<R>$ can be computed using a simple traversal of the dominator tree D. The tree D is made a subtree of the original dominator tree by making w a child of u. Priorities can be assigned to vertices in R using a simple batch algorithm—we just need to ensure that the new priorities are all greater than $priority(u)$.

The above steps are sufficient as long as there is no edge from a vertex in R to some previously reachable vertices. If there is such an edge, we identify the set X of edges from vertices in R to vertices outside R. Now we need to process the "insertion" of the edges in X using the special-case algorithm outlined previously. We could process these edges one by one, but that is not necessary. We know that $\bigcup_{x \longrightarrow y \in X} PossibleAffected(x,y)$ is an approximation to the set of affected vertices. This approximation to the affected vertices can be processed as before (*i.e.*, in order of increasing priorities).

The overall complexity of the algorithm consists of two components: the time spent on updating priorities, which is $O(\gamma \log \gamma)$,[2] and the time spent on the remaining steps, which is $O(\| VISITED \|^{\leftarrow} \log n)$, where $VISITED$ is the approximation to $AFFECTED$ identified by the algorithm.

3.2.2. The Deletion of an Edge

We now consider the problem of updating the dominator tree and the auxiliary information after the deletion of an edge $u \longrightarrow w$. Again, we first consider the special case where the edge-deletion does not change the reachability status of any vertex. The vertex w becomes unreachable following the deletion of $u \longrightarrow w$ iff $u \longrightarrow w$ was the only

---

[2] The Alpern *et al.* algorithm can take time $O(\gamma^2 \log \gamma)$ to update priorities after the insertion of a number of edges into the dag. The situation that arises in our algorithm is, however, a special one and can be processed in $O(\gamma \log \gamma)$ time.

```
procedure $InsertEdge_{DominatorTree}(G, u \longrightarrow w)$
declare
    $G$: a directed graph
    $u \longrightarrow w$: an edge to be inserted into $G$
begin
    Insert edge $u \longrightarrow w$ into $E(G)$
    if $reachable(u)$ then
        if not $reachable(w)$ then
            Compute the set of newly reachable vertices $R$ and mark them as reachable
            Compute the dominator tree $D$ of $<R>$, the subgraph induced by $R$, with $w$ as the source vertex
            Make $D$ a subtree of $u$ by performing $link(u,w)$
            Determine the status of all edges in $<R>$
            Correctly prioritize the dag induced by the forward edges of $<R>$,
                ensuring that the priorities assigned are greater than $priority(u)$
            Insert all forward edges in $<R>$ into $F(G)$
            Compute $X$, the set of edges from vertices in $R$ to vertices outside $R$
        else
            $X := \{ u \longrightarrow w \}$
        fi
        Determine the status of all edges in $X$
        Insert all the forward edges in $X$ into $F(G)$ and update the prioritization of $F(G)$
        $S := \bigcup_{x \longrightarrow y \in X} PossibleAffected(x,y)$
        for every $x \in S$ in increasing order of priority do
            $y$ := the least common ancestor of all predecessors in $F(G)$ of $x$
            if $y \neq idom(x)$ then
                $cut(idom(x),x)$
                $link(y,x)$
            fi
        od
    fi
end
```

Figure 3.4. An outline of the algorithm for updating the dominator tree of a reducible flowgraph after the insertion of an edge $v \longrightarrow w$ into graph G.

forward edge coming into w.

A Special Case: No Change in Reachability

We know there is no change in the reachability status of w if there is some other incoming forward edge at w. In this case, we remove the edge $u \longrightarrow w$ from $F(G)$ if the edge is a forward edge. Since the previous prioritization of $F(G)$ continues to be a correct prioritization of $F(G)$, no updating of priorities is required. The only non-trivial work is in updating the dominator tree.

We can very easily determine a good approximation to the set of affected vertices after the deletion of the edge $v \longrightarrow w$.

Proposition 3.2. Consider the deletion of an edge $u \longrightarrow w$ from a flowgraph, where both u and w remain reachable in the new graph too. If a vertex y is affected by the deletion of the edge, then y must be a sibling of w in the original dominator tree—that is, $idom(y) = idom(w)$ in the original dominator tree.

Proof. Assume we obtain graph G_2 from graph G_1 by deleting edge $u \longrightarrow w$. The insertion of the deleted edge back into G_2 will restore the dominator tree to its original form. In other words, the set of affected vertices when $u \longrightarrow w$ is inserted into G_2 is the same as the set of affected vertices when $u \longrightarrow w$ is deleted from G_1. We know from Proposition 3.1 that if a vertex y is affected by the insertion of edge $u \longrightarrow w$ into G_2 then it must be a sibling of w in the dominator tree for G_1. □

The General Case

The general-case updating algorithm for processing the deletion of an edge $u \longrightarrow w$ is similar to the general-case updating algorithm for processing an edge insertion. The algorithm performs the following steps: (1) Determine the set R of vertices that have become unreachable, (2) Determine the set F of edges of the form $x \longrightarrow y$, where $x \in R$ and $y \notin R$, and (3) Process the set F of edges, by essen-

```
procedure $DeleteEdge_{DominatorTree}(G, u \longrightarrow w)$
declare
    $G$: a directed graph
    $u \longrightarrow w$: an edge to be deleted from $G$
begin
    Remove edge $u \longrightarrow w$ from $E(G)$
    if $u \longrightarrow w$ is a forward edge then
        Remove edge $u \longrightarrow w$ from $F(G)$
        if $w$ has no incoming forward edges then
            Let $R$ be the descendants of $w$ in the dominator tree
            Mark vertices in $R$ unreachable
            Let $X$ be the set of forward edges from vertices in $R$ to vertices outside $R$
            Remove $X$ and all forward edges in <$R$> from $F(G)$
        else
            $X := \{ u \longrightarrow w \}$
        fi
        $S := \bigcup_{x \longrightarrow y \in X} Siblings(y)$
        for every $x \in S$ in increasing order of priority do
            $y :=$ the least common ancestor of all predecessors in $F(G)$ of $x$
            if $y \neq idom(x)$ then
                $cut(idom(x), x)$
                $link(y, x)$
            fi
        od
    fi
end
```

Figure 3.5. An outline of the algorithm for updating the dominator tree of a reducible flowgraph after the deletion of an edge $v \longrightarrow w$ from graph G.

tially working as though these edges have been deleted from the graph.

Note that the set R of vertices that have become unreachable is precisely the set of vertices dominated by w, that is, the set of vertices in the subtree T rooted at w in the original dominator tree. The set F of edges is essentially the set of edges $x \longrightarrow y$ with x in the subtree T and y not in T—this set of edges is obtained easily by traversing the subtree T. The "deletion" of the edges in F is processed as follows: $\bigcup_{x \longrightarrow y \in F} Siblings(y)$ is an approximation to the set of affected vertices, which can be processed using previously explained techniques.

The overall complexity of the edge-deletion procedure is $O(\| VISITED \|^{\leftarrow} \log n)$, where *VISITED* is the approximation to the set of affected vertices used by the algorithm.

4. Conclusion

In this paper we have presented an incremental algorithm for the problem of maintaining the dominator tree of a reducible flowgraph. The only other algorithm we are aware of for this problem is due to Carroll and Ryder [Car88, Car88a]. We now briefly compare these two algorithms and argue that the new algorithm is more efficient than the Carroll-Ryder algorithm.

Our algorithm has a better worst-case time complexity than the Carroll-Ryder algorithm. In the worst case, the algorithm presented in this paper can take $O(m \log n)$ time, where m is the number of edges and n is the number of vertices in the flowgraph. Note that m is $O(n)$ for typical flowgraphs. Carroll and Ryder do not describe the worst-case complexity of their algorithm in terms of n and m. In Carroll's thesis, the complexity of the Carroll-Ryder incremental algorithm is described in terms of "rotations" (see below): their algorithm can perform $\Omega(n^2)$ rotations in the worst case (even with sparse flowgraphs), and the cost of a rotation in the worst-case is $O((f+1) \times g \times d^2)$, where f is the maximum number of children of any vertex in the dominator tree, g is the maximum degree of any vertex in the control-flow graph augmented with certain "representative edges" (see below), and d is the height of the dominator tree. In the worst case, f, g, and d can each be $O(n)$, though f will usually be a small constant for typical flowgraphs. It is not obvious if the worst case for the number of rotations and the parameters f, g, and h can be simultaneously achieved, but the above clearly demonstrates that our algorithm has a better worst-case complexity.

However, worst-case analysis with the complexity expressed as a function of the (current) input size often does not characterize the complexity of incremental algorithms accurately, and the relative merits of incremental algorithms cannot be established by directly comparing their worst-case complexity measures. For example, the linear-time batch algorithm for constructing the dominator tree has a better complexity than both our incremental algorithm and the Carroll-Ryder incremental algorithm, which have a non-linear worst-case complexity, though the incremental algorithms are arguably better than the batch algorithm. In his thesis, Carroll presents experimental results showing that the Carroll-Ryder algorithm is faster than the almost-linear batch algorithm due to Lengauer and Tarjan [Len79].

There are two reasons why we believe that our algorithm will perform better in practice than the Carroll-Ryder algorithm. The first is that the Carroll-Ryder algorithm utilizes certain “representative edges” as auxiliary information, and maintaining this auxiliary information can be expensive both in time and space. (For every edge $u \longrightarrow v$ in the flowgraph, and for every vertex w that dominates u but not v, a representative edge $w \longrightarrow v$ needs to be maintained. In terms of the dominator tree, every flowgraph edge $u \longrightarrow v$ requires a representative edge $w \longrightarrow v$ for every vertex w on the path from u to $lca(u,v)$, excluding the end-points.) For example, a single edge-deletion (from a sparse flowgraph) that is processed in $O(\log n)$ time by our algorithm can result in the introduction of $\Omega(n^2)$ new representative edges, forcing the Carroll-Ryder algorithm to take $\Omega(n^2)$ time. The second reason is that the Carroll-Ryder algorithm restructures the dominator tree using “local rotation” operations, which move a subtree up or down one level at a time in the dominator tree. (In particular, an upward rotation makes a vertex the child of its original grandparent, while a downward rotation makes a vertex the child of one of its original siblings.) In contrast, our algorithm determines the new immediate dominator of vertices and moves each subtree only once, but such a move can take $O(\log n)$ time because of the use of link and cut operations. Consequently, one can easily create input modifications that are processed in $O(\log n)$ time by our algorithm for which the Carroll-Ryder algorithm performs $\Omega(n)$ rotations, each rotation requiring $\Omega(n)$ time.

One of the differences between our approach and the approach of Carroll and Ryder is that to handle graphs with unreachable vertices they extend the definition of dominator tree to that of a *dominator forest*: they consider a decomposition of the whole graph into a collection of flowgraphs, each with its own source vertex, such that every vertex is reachable from the source of the flowgraph it belongs to; the collection of the dominator trees of these flowgraphs constitute a dominator forest. If the graph has a unique minimal decomposition, then this approach is meaningful. However, a graph need not have a unique minimal decomposition, in general, and the dominator forest is not uniquely defined. Since the advantages of maintaining such a dominator forest are unclear, we have restricted our attention to the problem of maintaining the dominator tree of the reachable vertices.

Finally, we should mention that the algorithm described in Carroll’s thesis is capable of handling the simultaneous insertion and deletion of edges from the flowgraph. The algorithm presented in this paper can be adapted to handle the deletion of multiple edges or the insertion of multiple edges. An arbitrary change can be handled by processing all the deleted edges in one step, and then processing all the inserted edges in a second step. Further work is required to explore if the algorithm can be adapted to handle the insertion and deletion of edges simultaneously.

Another interesting direction for future research is suggested by a generalization of the link-cut tree data structure due to Cohen and Tamassia [Coh92, Coh92a]. Consider a tree in which each vertex is associated with some attributes, each of which is defined in terms of attributes of adjacent vertices. Cohen and Tamassia show how to efficiently maintain such an attributed tree dynamically when the attribute equations are linear expressions involving operators that form a semiring. Their algorithm performs link and cut operations in $O(\log n)$ time. Attribute values are not maintained but can be computed on demand in logarithmic time. Carroll and Ryder show that monotonic dataflow analysis problems can be reduced to an attribute evaluation problem over the dominator tree, and use Reps’s [Rep82, Rep83] incremental attribute updating algorithm to perform incremental dataflow analysis. Since the function spaces of distributive dataflow analysis frameworks form a semiring with respect to function composition and meet, it seems worthwhile exploring the possibility of using the Cohen and Tamassia algorithm, in conjunction with our incremental dominator tree algorithm, to perform incremental dataflow analysis.

References

Aho86.
Aho, A.V., Sethi, R., and Ullman, J.D., *Compilers: Principles, Techniques, and Tools,* Addison-Wesley, Reading, MA (1986).

Alp90.
Alpern, B., Hoover, R., Rosen, B.K., Sweeney, P.F., and Zadeck, F.K., “Incremental evaluation of computational circuits,” pp. 32-42 in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms,* (San Francisco, CA, Jan. 22-24, 1990), Society for Industrial and Applied Mathematics, Philadelphia, PA (1990).

Car88.
Carroll, M. and Ryder, B., “Incremental data flow update via attribute and dominator updates,” pp. 274-284 in *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages,* (San Diego, CA, January 13-15, 1988), ACM, New York, NY (1988).

Car88a.
Carroll, M.D., “Data flow update via dominator and attribute updates,” Ph.D. dissertation, Rutgers University, New Brunswick, NJ (May 1988).

Coh92a.
Cohen, R.F., “Combine and Conquer,” Ph.D. dissertation, Technical

Report No. CS-92-45, Department of Computer Science, Brown University, Providence, RI (October 1992).

Coh92.
Cohen, R.F. and Tamassia, R., “Combine and Conquer,” Technical Report No. CS-92-19, Department of Computer Science, Brown University, Providence, RI (April 1992).

Cyt89.
Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., and Zadeck, K., “An efficient method of computing static single assignment form,” pp. 25-35 in *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages,* (Austin, TX, Jan. 11-13, 1989), ACM, New York, NY (1989).

Cyt91.
Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., and Zadeck, F.K., “Efficiently computing static single assignment form and the control dependence graph,” *ACM Trans. Program. Lang. Syst.* **13**(4) pp. 451-490 (October 1991).

Har85.
Harel, D., “A linear time algorithm for finding dominators in flow graphs and related problems,” pp. 185-194 in *Proceedings of the Symposium on Theory Of Computing*, (1985).

Len79.
Lengauer, T. and Tarjan, R.E., “A fast algorithm for finding dominators in a flowgraph,” *ACM Transactions on Programming Languages and Systems* **1**(1) pp. 121-141 (July 1979).

Och83.
Ochranova, R., “Finding dominators,” pp. 328-334 in *Proceedings of the Foundations of Computation Theory*, (1983).

Rep82.
Reps, T., “Optimal-time incremental semantic analysis for syntax-directed editors,” pp. 169-176 in *Conference Record of the Ninth ACM Symposium on Principles of Programming Languages,* (Albuquerque, NM, January 25-27, 1982), ACM, New York, NY (1982).

Rep83.
Reps, T., Teitelbaum, T., and Demers, A., “Incremental context-dependent analysis for language-based editors,” *ACM Trans. Program. Lang. Syst.* **5**(3) pp. 449-477 (July 1983).

Sle83.
Sleator, D.D and Tarjan, R.E., “A data structure for dynamic trees,” *Journal of Computer and System Sciences* **26** pp. 362-391 (1983).