McGill University
School of Computer Science
ACAPS Laboratory

DJ-Graphs and Their Applications to Flowgraph Analyses

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

ACAPS Technical Memo 70
May 11, 1994

Advanced Compilers, Architectures and
Parallel Systems Group

†Application Development Technology Institute
IBM Software Solutions Division
555 Bailey Avenue
San Jose, CA 95141

A version of this report has been submitted for publication

ACAPS Laboratory • School of Computer Science • 3480 University St. • Montreal • PQ • Canada • H3A 2A7
Correspondence to: V. C. Sreedhar (sreedhar@acaps.cs.mcgill.ca)

Contents

Abstract

In this paper we elaborate on the properties of DJ-graphs — a new program representation recently introduced for flowgraph analysis [SG94]. We present three applications of DJ-graphs for flowgraph analysis and the main contributions of this paper are as follows.

- We design a simple algorithm for computing *multiple-node immediate dominators* that has a better (polynomial-time) worst-case time complexity than the existing algorithm [Gup92]. We show how DJ-graphs can be used to reduce the search space when computing multiple-node immediate dominators.
- We show how to identify loops, both reducible and irreducible, in a flowgraph using its DJ-graph by extending Tarjan's approach. The advantage of our method is being able to identify the bodies of arbitrarily nested (reducible and irreducible) loops, with the only restriction that a sequence of consecutively nested irreducible loops will be identified as a single irreducible loop (region).
- We present a simple algorithm to compute dominance frontiers and the related sets using DJ-graphs, which is linear both in space and time.

1 Introduction

Compile-time program optimization, especially for large programs, requires suitable representations for performing fast analysis and program transformation. Program analysis can be divided into two parts: data flow analysis and control flow analysis. Most often data flow analysis propagates data flow information with data structures derived from control flow analysis.[1] Thus, the first step in any optimizing compiler for high-performance architectures is control flow analysis.

Control flow analysis is performed on a graph representation of a program called the Control Flow Graph (CFG). The CFG nodes represent program statements and edges represent flow of control from one statement to another.[2] Given the CFG of a program, control flow analysis determines some pertinent properties of the control flow structure such as dominance relation, dominance frontier, control dependence relation, iterated dominance relation, and so on. These properties are the foundation upon which many intermediate program representations have been derived. For example, determining flowgraph reducibility can use dominance relation; constructing Static Single Assignment (SSA) form or Sparse Evaluation Graphs (SEGs) may require the computation of iterated dominance frontiers.

Recently, we introduced the DJ-graph as a representation for computing ϕ-nodes for an arbitrary SEG in linear time [SG94]. The skeleton of a DJ-graph is the dominator tree, which we augmented with *join* edges. A CFG edge is called a join edge if its source node does not strictly dominate its destination node. The DJ-graph allowed us to design a surprisingly simple, linear algorithm for computing ϕ-nodes. In the process of developing the linear algorithm we found many properties of the DJ-graph that are useful in performing other control flow analyses. In this paper we expound the properties of DJ-graphs that are pertinent to these analyses. We have developed algorithms for three control flow analyses applications and illustrate the power of the DJ-graph for performing these analyses. The main results are summarized below.

[1] Solving flow-insensitive interprocedural data flow problems, such as MOD and REF, may not need the knowledge of control flow [Bur90].

[2] In some analyses, a CFG node represents a basic block which is a single-entry, single-exit chunk of code.

Computing generalized dominators. Recently Gupta introduced a relation called multiple-node immediate dominators of a node [Gup92]. In a multiple-node immediate dominance relation a subset of the predecessor nodes of a node together dominates the node. This relation has been used for helping propagate loop invariants and check array bounds. Gupta gave a simple algorithm for computing the multiple-node dominance relation that has a worst-case time complexity of $O(|N|^p)$, where $|N|$ is the number of CFG nodes and p is bounded by the maximum number of predecessors of any node. We have devised a simple algorithm that has a better worst-case time complexity of $O(|E| \times |N| \times p)$, where $|E|$ is the number of edges in the flowgraph. We show how the properties of DJ-graphs can be used to reduce the search space when computing multiple-node immediate dominators.

Identifying multiple-entry irreducible loops. Certain dataflow analysis algorithms require that the flowgraph be reducible; therefore, it is important to verify whether the flowgraph is reducible or not. Furthermore, some compiler analyses and transformations may require explicit identification of irreducible loops [Ste93]. Using DJ-graphs we present an algorithm to identify both reducible and irreducible loops by extending Tarjan's approach [Tar74]. The algorithm has a time complexity close to linear. The advantage of our method is to be able to identify the bodies of arbitrarily nested (reducible and irreducible) loops, with the only restriction that a sequence of consecutively nested irreducible loops will be identified as a single irreducible loop (region). It can expose maximally, in a loop nest, the nesting structure of each portion which is reducible separated by irreducible regions.

Computing dominance frontiers. Dominance frontiers [CFR+91] and the dual control dependence relation CONDS [CFS90] have been successfully used in many optimizations and vectorizations. CONDS relation is nothing but dominance frontiers on the reverse flowgraph. We will give a simple algorithm for computing related sets in Section 5 which is linear both in space and time. Our algorithm is comparable to the algorithm proposed by Cytron et. al. for computing the CONDS relation [CFS90]. Another feature of our method is that it can be easily extended to compute iterated dominance frontiers of a node (or a set of nodes).

The rest of the paper is organized as follows. Section 2 introduces the DJ-graph and discusses some of its properties. In Section 3 we discuss how to compute multiple-node immediate dominance relation using DJ-graphs. In Section 4 we demonstrates the application of DJ-graphs for identifying loops. In Section 5 we give a simple linear algorithm for computing dominance frontiers of a node. In Section 6 we discuss some related work. Finally, Section 7 gives our concluding remarks and future work.

2 The DJ-graph

In this section, we introduce the DJ-graph and discuss some of its properties. We begin by defining notation and terminology that will be used later in our discussions.

2.1 Background and Notation

A Control Flow Graph is a connected, directed graph $G = (N, E, \text{START}, \text{END})$, where N is the set of nodes, E is the set of edges, START $\in N$ is a distinguished start node, and END $\in N$ is a distinguished end node. Figure 1(a) shows an example CFG. If $x \to y \in E$, then x and y are called the *source* and the *destination* node of the edge, respectively. A *path* of length n is a sequence of edges $(x_0 \to x_1 \to \ldots \to x_n)$, where each $x_i \to x_{i+1} \in E$. We use the notation $P : x_0 \xrightarrow{*} x_n$ to represent a path of length zero or more, and $P : x_0 \xrightarrow{+} x_n$ to represent a path of length one or more. In our definition of CFG, we will assume that for every node $x \in N$ there is a path from START to x and a path from x to END.[3]

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

Node x **dominates** y if and only if 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 when x does not dominate y. Node x **strictly dominates** y if and only if x **dom** y and $x \neq y$. We write x **sdom** y to indicate x strictly dominates y, and write x !**sdom** y when 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 the parent of y in the dominator tree, then x **immediately dominates** y. 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 subtree rooted at x. Note that the nodes in $SubTree(x)$ are simply the set of all nodes dominated by x. Figure 1(b) shows the dominator tree (ignore the thick solid edges for the moment) for the flowgraph in Figure 1(a).

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 node x. A number of algorithms have been proposed for computing dominance relation. Lengauer and Tarjan proposed an algorithm that is almost linear, $O(E\alpha(N, E))$, where $\alpha()$ is the slowly-growing inverse Ackermann function [LT79]. More recently, Harel has given a *linear* algorithm for computing the dominator tree of a flowgraph [Har85].

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]. For the control flow graph given in Figure 1, $DF(3) = \{5, 7\}$.

We can extend the definition of dominance frontier to a set S of nodes:

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

We define **iterated dominance frontier** DF^+ for a set of nodes as the limit of the increasing sequence:

$$\begin{aligned} DF_1 &= DF(S), && (1) \\ DF_{i+1} &= DF(S \cup DF_i) && (2) \end{aligned}$$

2.2 Properties of DJ-graphs

The DJ-graph of a flowgraph consists of the same set of nodes as in the flowgraph, and two types of edges called D-edges and J-edges. D-edges are dominator tree edges, and we define J-edges as follows:

[3] We will use the terms flowgraph and control flow graph interchangeably.

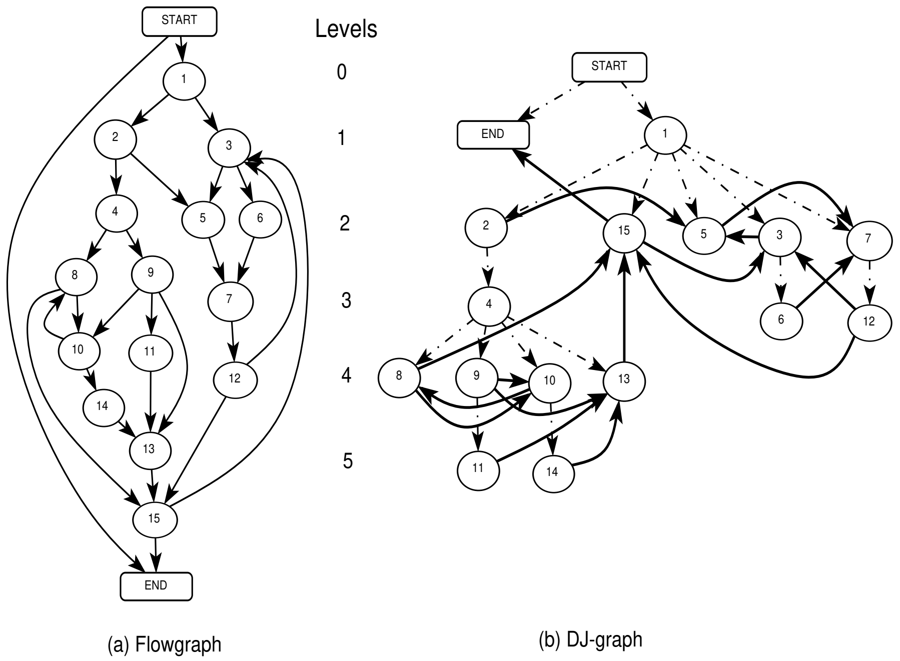


Figure 1: An example: (a) Flowgraph; (b) DJ-graph (dashed edges are D-edges, and thick and solid edges are J-edges)

Definition 2.1 (J-edge)

An edge $x \to y$ in a flowgraph is named a **join edge** *(or* **J-edge***) if x* **!sdom** *y. Furthermore, y is named a* **join node***.*

Therefore, in order to construct the DJ-graph of a flowgraph, we first construct the dominator tree of the flowgraph. Then, we insert the J-edges into the dominator tree as follows:

For each **join node** y in the dominator tree connect x to y (in the dominator tree) if and only if $x \to y$ is a **join edge** in the original flowgraph.

Figure 1(b) shows the DJ-graph for our example CFG. To see how a J-edge is inserted into the dominator tree, consider join node 3 in the example flowgraph. This node has three incoming edges, $1 \to 3$, $12 \to 3$, and $15 \to 3$. Of these three incoming edges, node 1 strictly dominates join node 3, so we do not insert a J-edge from 1 to 3 in the corresponding dominator tree. However, neither node 12 nor 15 dominates node 3, and thus we insert J-edges from nodes 12 and 15 to node 3.

Given the DJ-graph of a flowgraph, we can classify its J-edges into *back J-edges* (BJ-edges) and *cross J-edges* (CJ-edges). A J-edge $x \to y$ is a BJ-edge if y **dom** x, otherwise it is a CJ-edge.

We can easily see that the time complexity for inserting all the J-edges into the dominator tree is $O(|E|)$. Therefore the time complexity of constructing a DJ-graph is linear with respect to the size of the flowgraph.[4] Also, the size of the DJ-graph is bounded by a small constant with respect to the size of the flowgraph (Proposition 2.1).

Proposition 2.1

Let $FG = (N_f, E_f)$ be a flowgraph, and let $DJ = (N_d, E_d)$ be the corresponding DJ-graph. Then, $|N_d| = |N_f|$ and $|E_d| < (|N_f| + |E_f|)$.

In the following we will discuss some structural properties of the DJ-graph. First we investigate path relationship between a flowgraph and its DJ-graph. Then we discuss how J-edges are related to levels of nodes.

For every edge $x \to y$ in a flowgraph, there is a corresponding edge $x \to y$ in the DJ-graph. But the reverse is not true; that is, the DJ-graph can have more edges than its flowgraph does. In addition, for every path $P_f : x \xrightarrow{*} y$ in a flowgraph there is a corresponding path $P_d : x \xrightarrow{*} y$ in the DJ-graph, and vice versa. This immediately follows from construction of the DJ-graph. Note that the nodes in the two paths may not be exactly the same. In other words, given any two nodes x and y, y is reachable from x in a flowgraph if and only if y is reachable from x in the corresponding DJ-graph.[5]

[4] Note that we can construct the dominator tree of a flowgraph in linear time [Har85].

[5] Another view of the DJ-graph may give a better intuition behind these observations: A DJ-graph can also be constructed by adding every missing *immediate dominance edge* $x \to y$ into its flowgraph if the edge is not yet in the flowgraph and x is the immediate dominator of y.

Also recall that there is exactly one path from START to any other node in the dominator tree of a flowgraph. Due to the presence of J-edges, there may be more than one path from START to any other node in the DJ-graph. How are J-edges related to D-edges? Or, is there a flowgraph that corresponds to any arbitrary DJ-graph? We will show that there may not exist a flowgraph for a given graph that "looks like a DJ-graph". To see this consider the graph shown in Figure 2. Although this graph structurally looks like a DJ-graph, it is impossible to construct a flowgraph that corresponds to this DJ-graph.

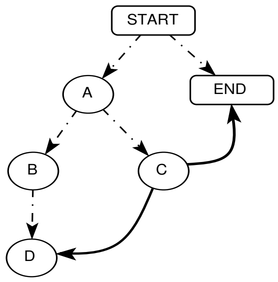


Figure 2: An invalid DJ-graph

Lemma 2.1

Let x and y be any two nodes in a DJ-graph. Let there be a path P from x to y such that all edges in P are D-edges (this means x **dom** *y in the corresponding flowgraph). If there exists a path Q (which can contain D-edges as well as J-edges) in the DJ-graph from the* START *to y that does not go through x, then there is no flowgraph that can correspond to this DJ-graph.*

Proof:

If such a path Q exist then x will not dominate y, contradicting our initial assumption. ∎

Consider the graph in Figure 2, there are two paths to the node D from the START: P : START $\rightarrow A \rightarrow B \rightarrow D$, and Q : START $\rightarrow A \rightarrow C \rightarrow D$. In the figure B **dom** D, but the path Q does not pass through B, contradicting the assumption that B dominates D. An immediate corollary to the above proposition is as follows:

Corollary 2.1

If x **dom** *y in a flowgraph, then all paths from* START *to y in the corresponding DJ-graph must pass through x.*

Intuitively, Lemma 2.1 and Corollary 2.1 implies that J-edges *preserve* the dominance relation. In other words, if we compute dominance relation directly from the DJ-graph, rather from the flowgraph, we get the same dominator tree.

From the above discussion we can easily derive the following lemma.

Lemma 2.2

*Let $x \rightarrow y$ be a J-edge, then $idom(y)$ **sdom** x.*

Proof:

Let $z = idom(y)$, and assume that z !**sdom** x. We can immediately see that there is a path from START to y that does not go through z, contradicting our initial assumption that z immediately dominates y. ∎

From Lemma 2.2 we can see that if $x \rightarrow y$ is a J-edge then the level number of x is always greater than or equal to that of y. This observation leads to the following lemma.

Lemma 2.3

Let $x \rightarrow y$ be a J-edge in a DJ-graph, then $x.level \geq y.level$.

Proof:

Suppose $x.level < y.level$. There are two cases:

1. There is a path from x to y in the dominator tree. This is impossible by definition (because x will dominate y).
2. There is no path from x to y in the dominator tree. Let z be a node such that $z.level = x.level$, and z **dom** y (actually, z strictly dominates y). Since $x \rightarrow y$ is in the DJ-graph, there is a path from START to x to y in the CFG that does not pass through z (a contradiction, because we have assumed z **dom** y).

∎

Now, if $x \rightarrow y$ is a J-edge, then $y \in DF(x)$. Our next question is, in general, how the nodes in $DF(x)$ are related to x in terms of the level of nodes. The following theorem formalizes this concept, which is a stronger version of Lemma 2.3.

Theorem 2.1

Let x be a node in the DJ-graph, and let $DF(x)$ be the dominance frontier of x. Then, $x.level \geq y.level$ for each $y \in DF(x)$, and $x.level \geq y.level$ for each $y \in DF^+(x)$.

Proof:

Let $y \in DF(x)$ and let $u = idom(y)$. We will first show that u will strictly dominate x. Then using this result we will prove the validity of the theorem. Assume that u does not strictly dominate x. Then there is a path from START to x that does not pass through u. Since $y \in DF(x)$, there must be a predecessor node z strictly dominated by x. From this we can immediately see that there is a path from START $\ldots \rightarrow x \ldots \rightarrow \ldots z \rightarrow y$ in the DJ-graph that does not pass through u, contradicting the assumption that $u = idom(y)$. Therefore u **sdom** x. From this we can easily see that

$$u.level < x.level$$

Again since $u = idom(y)$, $u.level = y.level - 1$. Substituting this in the above inequality, we get

$$y.level - 1 < x.level$$

or

$$y.level \leq x.level$$

Hence the result.

The property $x.level \geq y.level$ for each $y \in DF^+(x)$, inductively follows from the definition of iterated dominance frontier and the property $x.level \geq y.level$ for each $y \in DF(x)$.

∎

Intuitively, Theorem 2.1 implies that given a node x, to determine its dominance frontier (or iterated dominance frontier) we need to look at only those nodes at the same level as x or above it, in the corresponding DJ-graph. Nodes with level number greater than $x.level$ will never be in the dominance frontier of x. In Section 5 we will give a simple algorithm for computing the dominance frontier of a node in linear time using level information and J-edges.

3 Computing Multiple-node Immediate Dominance Relation

The traditional definition of dominance relation is called single-node dominance relation, meaning every node except for the START node has exactly one immediate dominator. In a recent paper [Gup92], Gupta introduced the notion of multiple-node dominance relation, where a node can have more than one immediate dominator. Generalized dominance relation captures both single-node and multiple-node dominance relation in a unified way. His definition of generalized dominance relation is as follows:

Definition 3.1

A set of nodes S dominates node x if and only if:
(1) *all paths from* START *to x contain some node $y \in S$; and*
(2) *for each $y \in S$, there is at least one path from* START *to x which contains y but does not contain any other node in S.*

Gupta presented a two-step process for computing multiple-node dominance relation. First multiple-node *immediate* dominator set is computed, with which multiple-node dominance relation is computed. The notion of multiple-node immediate dominance relation is similar to single-node immediate dominance relation. Gupta defines multiple-node immediate dominance relation as follows. (Note that $Pred_f(x)$ represents the predecessor nodes of x in the flowgraph.)

Definition 3.2

A multiple-node immediate dominator set $midom(x)$ of a node x is defined to be a subset of $Pred_f(x)$ which dominates x.

We distinguish between single node immediate dominator ($sidom$) and multiple-node immediate dominator ($midom$) set. Every node has exactly one and only one $sidom$ (except the START node which has none). The $midom$-set of a node, although unique, can be empty. The $midom$-set of a node is empty whenever the node satisfies the following property:

Lemma 3.1

The $midom$-set of a node x ($x \neq$ START) is empty if and only if there is a flowgraph edge from $sidom(x)$ to x.

Therefore, we will compute the $midom$-set only for those nodes x such that $sidom(x) \rightarrow x$ is not a flowgraph edge. We will use $\mathcal{R}$ to denote the *relevant* set of nodes whose $midom$-sets are *not empty*.

Gupta gave an algorithm for computing multiple-node immediate dominators with $O(|N|^p)$ time, where p is bounded by the maximum number of predecessors of any node. Here we give a simple algorithm for computing multiple-node immediate dominator with a better worst-case time complexity of $O(|E| \times |N| \times p)$. A key observation we made in our approach is to notice that if a node y is in $midom(x)$, then there is at least one path from $sidom(x)$ to y that does not contain any other nodes in $Pred_f(x)$ (excluding y itself). Therefore, in order to compute $midom(x)$ we check for each $y \in Pred_f(x)$ whether there is a path from $sidom(x)$ to y that does not contain any other nodes in $Pred_f(x)$. If such a path exists then we add y to $midom(x)$. Thus, in our approach, we start off with an empty $midom(x)$, and add nodes from $Pred_f(x)$ to $midom(x)$ if they satisfy the aforementioned property. This is in contrast to Gupta's method, where the $midom(x)$ set is initially assumed to be $Pred_f(x)$, and then nodes are removed from $midom(x)$ until the set satisfies certain properties (Lemma 2 in [Gup92]).

In order to check if the above mentioned path exists, we do the following: For each node $y \in Pred_f(x)$ (assuming that we are computing $midom(x)$), we 'outblock' all the nodes in $Pred_f(x)$ except y. By 'outblock' we mean that all the outgoing edges from the node are conceptually cut. Next we check if x is reachable from $sidom(x)$. If so we have found a path from $sidom(x)$ to x that does not pass through any other node than y in $Pred_f(x)$. This means that $y \in midom(x)$. We do the above process for all nodes $Pred_f(x)$. The complete algorithm for computing $midom(x)$ for every node $x \in \mathcal{R}$ is given below.

To simplify the presentation of the algorithm and proof of correctness, we use the following notation and data structures:

- $NumLevel$ is the total number of levels in the dominator tree embedded in the DJ-graph.
- Each node $x \in N$ has the following attributes:

$$
\begin{array}{l}
struct\ NodeStructure\{ \\
\quad proc = \{Processed, NotProcessed\} \\
\quad instack = \{InStack, NotInStack\} \\
\quad outblocked = \{OutBlocked, NotOutBlocked\} \\
\quad level = \{0 \ldots NumLevel - 1\} \\
\}
\end{array}
$$

- **PushNode**(x) inserts x into a temporary *stack*. **PopNode**() retrieves a node from the stack. **ClearStack**() clears the stack.
- $Succ(x) = \{y|x \rightarrow y \text{ is an edge}\}$ and $Pred(x) = \{y|y \rightarrow x \text{ is an edge}\}$. We will use subcripts f or d to distinguish between edges in flowgraph or DJ-graph, respectively. For example, $Succ_f(x)$ means successor nodes of x in the flowgraph.

The first step in the algorithm is to mark all the predecessor nodes $Pred_f(x)$ of x as $OutBlocked$ in the DJ-graph (**for** loop at step $\boxed{2}$). Then select one of the 'un-processed' node that is marked $OutBlocked$ and un-block it (step $\boxed{7}$). Next call the function **VisitedX**($sidom(x), x$) (step $\boxed{8}$). This function returns $True$ if a path exists from some node in $Succ_f(sidom(x))$ to x that does not pass through any of the outblocked nodes, otherwise it returns $False$ (step $\boxed{8}$). Add y to M_x (step $\boxed{10}$) if the function **VisitedX**() returned $True$. At step $\boxed{13}$ we again outblock the current node. This process is repeated for all un-processed nodes in $Pred_f(x)$. When the procedure terminates the $midom$-set for node x is stored in M_x. The **for** loop at step $\boxed{16}$ un-blocks all outblocked nodes.

Algorithm 3.1 *The following algorithm computes the $midom$-set M_x for any $x \in \mathcal{R}$.*

Procedure MidomSet(x)
{

1: $M_x = \emptyset$
2: **foreach** $y \in Pred_f(x)$
3: $y.outblocked = OutBlocked$
4: **endfor**
5: **foreach** $y \in Pred_f(x)$
6: **if**($y.proc! = Processed$) `/* get the next un-processed node */`
7: $y.outblocked = NotOutBlocked$
8: $status$ = **VisitedX**($sidom(x), x$)
9: **if**($status == True$) `/* found a path */`
10: $M_x = M_x \cup y$ `/* insert in` M_x `*/`
11: **endif**
12: $y.proc = Processed$`/* mark processed*/`
13: $y.outblock = OutBlocked$
14: **endif**
15: **endfor**
16: **foreach** $y \in Pred_f(x)$
17: $y.proc = NotProcessed$ `/* recover the DJ-graph*/`
18: $y.outblocked = NotOutBlocked$
19: **endfor**

}

The function **VisitedX**() in conjunction with another function **Visit**() searches through the nodes below $sidom(x)$ to see if it can reach x from some node in $Succ_f(sidom(x))$ without passing through any of the outblocked nodes. **VisitedX**() returns $True$ if such a path exists.

The first step in function **VisitedX**() is to insert into a temporary *stack* all the nodes in $Succ_f(sidom(x))$ that are at the same level as x and are not marked $OutBlocked$ (**for** loop at step $\boxed{20}$). Then it picks a node z from the stack (step $\boxed{26}$) and calls another function **Visit**(z, x) (step $\boxed{27}$). The function **Visit**() returns $True$ if the node x was visited during the invocation of **Visit**(z, x), otherwise it returns $False$ (step $\boxed{27}$). When the $True$ value is returned, further search through any remaining nodes in stack is aborted and the $True$ value is propagated back to the main procedure **MidomSet**() (step $\boxed{30}$). Otherwise it picks another node from the stack and continues with the search until there are no more nodes in the stack. Whenever **Visit**() returns $True$, the function **VisitedX**() clears the stack (by calling the procedure **ClearStack**() at step $\boxed{29}$) before propagating the $True$ value back to the main procedure.

Function VisitedX(w, x)

```
{
20:   foreach z ∈ Succ_f(w)
21:     if((z.level == x.level) and
                (z.outblocked! = OutBlocked))
22:       PushNode(z) /* Push onto stack */
23:       z.instack = InStack
24:     endif
25:   endfor
26:   while((z = PopNode())! = NULL)
27:     status = Visit(z, x)
28:     if(status == True)
29:       ClearStack()
30:       return True /* found a path */
31:     endif
32:   endwhile
33:   return False /* not found a path */
}
```

The function **Visit**(z, x) basically walks down $SubTree(z)$ through D-edges (step $\boxed{36}$) and checks if there is a J-edge from some node $v \in SubTree(z)$ to x (where z was previously put on the stack). If such a J-edge is found (step $\boxed{42}$), further search down the subtree is aborted and the function returns $True$ back to function **VisitedX**() (step $\boxed{43}$). Otherwise the function visits all the nodes in $SubTree(z)$ through D-edges. Note that the function **Visit**() never walks down an outblocked node (step $\boxed{35}$). Whenever it finds a successor node to be outblocked, it avoids searching further down the tree through the outblocked node.

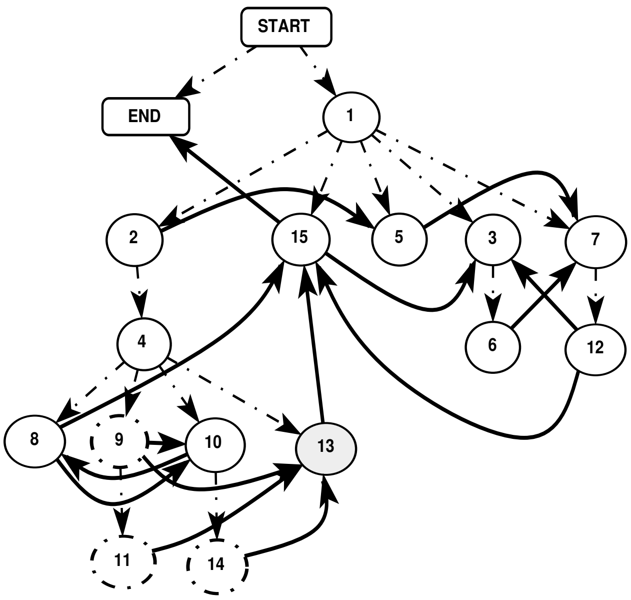
As it walks down the subtree, if it finds a J-edge whose destination node, say u, is at the same level as x, and u was not previously put on the stack, it pushes u onto the stack (step $\boxed{47}$). Note that it never pushes a node that is outblocked onto the stack.

```
Visit(z, x)
{
34:   foreach u ∈ Succ_d(z)
35:     if(u.outblocked != OutBlocked)
36:       if(z → u == Dedge) /* walk down D-edge */
37:         status = Visit(u, x)
38:         if(status == True)
39:           return True /* found a path, so stop searching further */
40:         endif
41:       else/* Jedge */
42:         if(u == x) /* found path to x */
43:           return True
44:         endif
45:         if(u.level == x.level) /* same level as x? */
46:           if(u.instack != InStack)
47:             PushNode (u)
48:             u.instack = InStack
49:           endif
50:         endif
51:       endif
52:     endif
53:   endfor
54:   return False
}
```

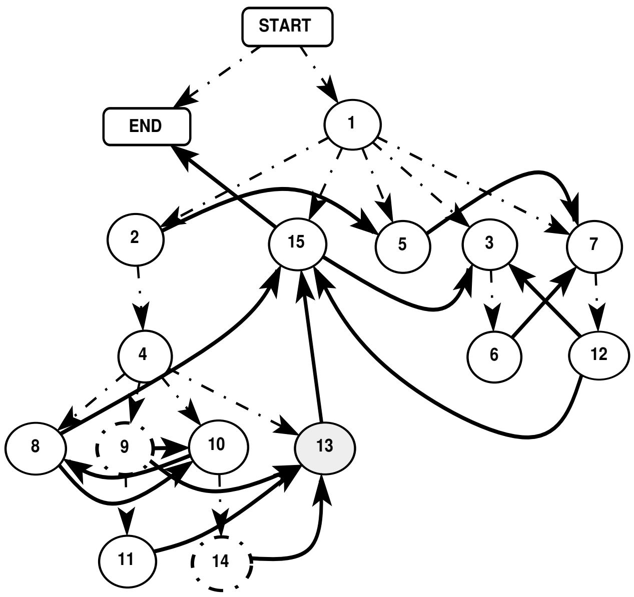
3.1 An Example

Let us illustrate the working of Algorithm 3.1 for the flowgraph given in Figure 1(a). Its corresponding DJ-graph is shown in Figure 1(b). Assume that we want to compute $midom(13)$, the multiple-node immediate dominator set for node 13. The first step is to outblock all the nodes in $Pred_f(13)$ in the DJ-graph (**for** loop at step $\boxed{2}$). The resulting DJ-graph is shown in Figure 3(a). Next we unblock one of the previously blocked node, say 11. The resulting DJ-graph is shown in Figure 3(b). Then we call the function **VisitedX**$(4, 13)$ (step $\boxed{7}$).

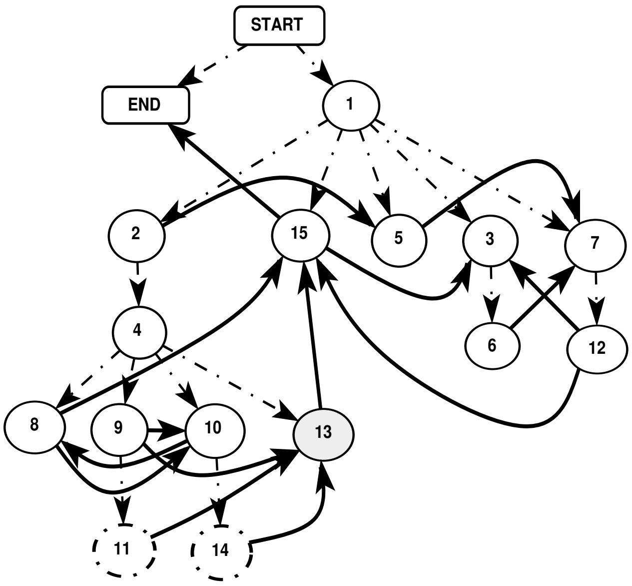
VisitedX$(4, 13)$ first inserts into a stack all the nodes in $Succ_f(4)$ that are at the same level as node 13 and are not marked $OutBlocked$ (**for** loop at step $\boxed{20}$). In this case only node 8 is pushed onto the stack, since node 9 is marked $OutBlocked$. Then we call **Visit**$(8, 13)$. This function will in turn visit all the nodes $SubTree(8)$ (step $\boxed{34}$), which contains only node 8. There are two J-edges from 8, $8 \to 10$ and $8 \to 15$. The level number of node 15 is less than that of 13, so we do nothing. Node 10 is not outblocked, is at the same level as node 13, and 10 was not previously put on the stack. So we push node 10 on the stack (step $\boxed{47}$). Since we have not yet reached our goal (that of reaching node 13), **Visit**() returns $False$ back to its calling function **VisitedX**() (step $\boxed{27}$). Since node 10 is on the stack, at step $\boxed{26}$ we pop the



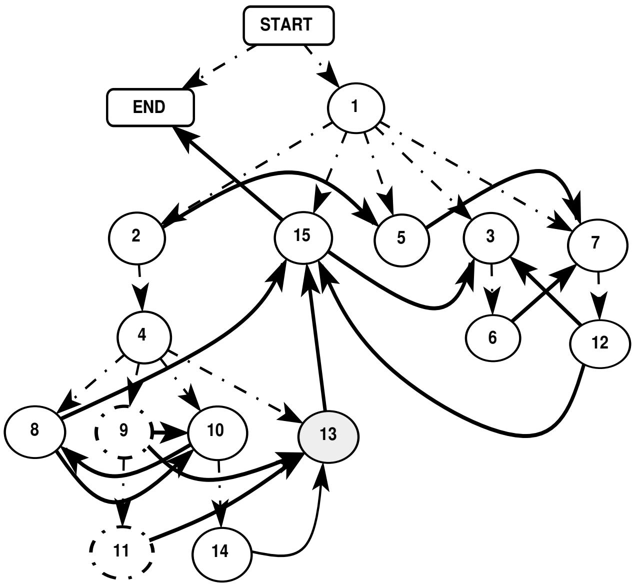
(a) All predecessor nodes are outblocked



(b) Only 9 and 14 are outblocked



(c) Only 14 and 11 are outblocked



(d) Only 9 and 11 are outblocked

Figure 3: Various stages of DJ-graph while computing $midom(13)$

node and call **Visit**$(10, 13)$ at step $\boxed{\textbf{27}}$. The only successor of node 10 is 14, and 14 is marked outblocked (step $\boxed{\textbf{35}}$). So **Visit**$(10, 13)$ returns $False$, and so does **VisitedX**$(4, 13)$ to the main procedure (since there are no more nodes on the stack). Consequently, we do not add node 11 to M_{13}.

Next, in the main procedure **MidomSet**(), we outblock node 11 and un-block node 9 (see Figure 3(c)). We again call the function **VisitedX**$(4, 13)$. As before, **VisitedX**$(4, 13)$ first pushed nodes 8 and 9 onto the stack (note that this time we have un-blocked node 9). Assume that node 9 was pushed earlier node 8. At step $\boxed{\textbf{27}}$ we call **Visit**$(8, 13)$. The only successors of 8 are node 10 and node 14. The scenario is same as before– from node 10 we visit node 14 which is outblocked (the level number of node 14 is less than level of 13.) Therefore, we return with $False$ back to **Visited**(). Next the function **VisitedX**() picks node 9 from the stack (step $\boxed{\textbf{26}}$) and invokes **Visit**$(9, 13)$. There are three successor nodes to node 9, nodes 11, 10, and 13. Node 11 is outblocked and node 10 was previously put on the stack. So we visit node 13, which is our goal state. Therefore we immediately return with $True$ (step $\boxed{\textbf{43}}$). This means we have found a path from 9, a node in $Succ_f(sidom(13))$, to node 13 that avoids any of the outblocked nodes. Consequently, we add node 9 to M_{13} (step $\boxed{\textbf{10}}$).

The only remaining un-processed node is 14. We un-block it and call the function **VisitedX**$(4, 13)$ (see Figure 3(d)). Performing the computation process as before, we will see that this function returns $True$; therefore, we include node 14 in M_{13}. So, finally we have $M_{13} = \{9, 14\}$.

3.2 Correctness

In order to prove that the Algorithm 3.1 correctly computes the $midom$-set for a node $x \in \mathcal{R}$ we have to show that, when the algorithm terminates, the set M_x (at step $\boxed{\textbf{10}}$) satisfies the definition of $midom$-set (Definition 3.2). In other words, we have to show that M_x satisfies the following conditions:

1. M_x is a subset of $Pred_f(x)$.
2. All paths from START to x contain some node $y \in M_x$, and for each $y \in M_x$, there is at least one path from START to x which contains y but does not contain any other node in M_x.

First of all notice that all paths from START to x must pass through $sidom(x)$. Also, all paths from START to any node y in $Pred_f(x)$ must again pass through $sidom(x)$ [PM72]. Therefore, it is sufficient to consider paths from $sidom(x)$ (rather than from START) x when arguing that M_x does indeed satisfies Definition 3.2.

Now, in order to show that M_x indeed satisfies the definition of $midom$-set we will first examine the what nodes are added to M_x when algorithm terminates. First of all observe that a node y is added to M_x at step $\boxed{\textbf{10}}$ only if **VisitedX**() returns $True$. When does **VisitedX**() return $True$? In Lemma 3.5 we will show that **Visited**() returns $True$ iff there exists a path from some node $y \in S$ to x in the DJ-graph that does not pass through any outblocked node (where S is a subset of $Succ_f(sidom(x))$ such that all the nodes in S are at the same level as node x). What this means is that for a node y to be in M_x there must exist a path from $sidom(x)$ to x that does not pass through any other node in $Pred_f(x)$. The proof of Lemma 3.5 relies on the result of the following three lemmas.

Lemma 3.2

Let x and y be any two nodes such that $y.level < x.level$. Then every path from y to x must pass through $sidom(x)$.

Proof:

Easily follows from properties of the DJ-graph. ∎

In the following discussions, we assume that x is in the relevant set $\mathcal{R}$. We also define a subset S of $Succ_f(x)$ such that for all $y \in S$, $y.level == x.level$. In the next lemma (Lemma 3.3) we will show that, in order to search for a outblock-free path from $sidom(x)$ to x, it is sufficient to start the search from nodes in S, thus reducing the search space.

Lemma 3.3

All simple paths (in the flowgraph) from $sidom(x)$ to x must contain some node $y \in S$, where S is as defined previously.

Proof:

Assume P is a simple path (in the flowgraph) from $sidom(x)$ to x that does not contain any node in S. Notice that all paths from $sidom(x)$ to x must contain some node in $Succ_f(x)$. Let $p \in Succ_f(sidom(x))$ be some node such that $p.level \neq x.level$ and is in the path P. First of all observe that $p.level$ can never be greater than $x.level$. So $p.level$ must be strictly less than $x.level$. Therefore, from Lemma 3.2, all paths from p to x must pass through $sidom(x)$, and so does P. If this is the case then P is not a simple path any more. The only way to reach x from p without going through $sidom(x)$ again is if level of p is greater than the level of $sidom(x)$. But p being a successor node, its level should be one greater than the level of $sidom(x)$, and hence at the same level as node x. But p should be in S in that case. Hence the result. ∎

In an attempt to search for a path from $sidom(x)$ to x, we use Lemma 3.2 to guarantee that it is safe not to search any nodes whose levels are less than $x.level$. In other words, we can limit the search for such a path to only nodes below $sidom(x)$. Lemma 3.3 specifies exactly which subset of nodes of $Succ_f(idom(x))$ one needs to consider when determining if node x can be reached from $sidom(x)$. The needed nodes in $Succ_f(idom(x))$ are those at the same level as x. The next lemma specifies which nodes need to be pushed onto the stack (at step $\boxed{\textbf{22}}$ and step $\boxed{\textbf{47}}$). Those that are at the same level as x and are reachable through a outblock-free path from some node in $Succ_f(idom(x))$.

Lemma 3.4

A node z is pushed onto the stack (at step $\boxed{\textbf{22}}$ and step $\boxed{\textbf{47}}$) only if there is a outblock-free path from some node in S, z is not outblocked, and $z.level = x.level$.

Proof:

Nodes are put on the stack only at two places: step $\boxed{\textbf{22}}$ and step $\boxed{\textbf{47}}$. At step $\boxed{\textbf{22}}$ only nodes in S that are not outblocked are pushed onto the stack. At step $\boxed{\textbf{47}}$ nodes are put on the stack when they are not outblocked, are siblings of x, and are reachable from some node $u \in SubTree(w)$, where w was previously put on the stack and is reachable from some node in S. ∎

Lemma 3.5 specifies that the function **VisitedX**($sidom(x), x$) does indeed find a path from some node in S to x that does not pass through any outblocked nodes, whenever such a path exists (where the set S is as defined previously).

Lemma 3.5

The function **VisitedX**($sidom(x), x$) *returns* $True$ *if and only if there exists a path from some node* $y \in S$ *to* x *in the DJ-graph that does not pass through any outblocked node.*

Proof:

⇒ We first show that if **VisitedX**($sidom(x), x$) returns $True$, then there is outblock-free path from some node $y \in S$ to x. Note that we never visit any successor of an outblocked node in the process (step $\boxed{\mathbf{35}}$). From Lemma 3.4 we know that a node is put on the stack only if it is not outblocked and there is a outblock-free path from some node in S to this node. At step $\boxed{\mathbf{43}}$ we return $True$ only if x is visited through some J-edge $z \to x$ (step $\boxed{\mathbf{42}}$). Node z was previously put on the stack and is reachable from y. Therefore, if **VisitedX**() returns $True$, then there is a outblocked-free path from y to x.

⇐ We next show that if there is a outblock-free path P from $y\ inS$ to x, then **VisitedX**($sidom(x), x$) returns $True$. Note that our algorithm outblocks all the nodes in $Pred_f(x)$ except one node, say node z. So if P is a outblock-free path, then P must contain z. Let P consists of a sequence of edges $y \xrightarrow{+} y_j \xrightarrow{+} z \to x$ such that none of the nodes in P are outblocked (where y is some node in S). We can see that as soon as node z is reached, **Visit**() will $True$ when it probes x through the J-edge $z \to x$. How do we know that **Visit**() will eventually reach z? The proof is based on induction on sequence of edges leading to z from y.

Base case Since $z \in Pred_f(x)$ the path length from some node $y \in S$ to z should be at least one (otherwise z will be same as $sidom(x)$). It is trivial to that z will eventually be reached.

Induction Let $y \xrightarrow{+} y_j \xrightarrow{+} u$ be a path that leads from y to u such that none of the nodes in the path is outblocked, and $u \in Pred_f(z)$. Since z is not marked outblocked **Visit**() visits z through the edge $u \to z$ (which can be either a D-edge or a J-edge). Once it reaches z it will eventually probe the node x through the J-edge $z \to x$, and so will return $True$. We know that **VisitedX**() returns $True$ only if $Visit$ returns $True$. Hence the result.

∎

Finally, we prove our main result in Theorem 3.1. In Lemma 3.5 we have established that **VisitedX**() returns $True$ iff there is an outblocked-free path from some node in S to x. In the Algorithm 3.1 a node y is added to the set M_x only if **VisitedX**() returns $True$, meaning only if there is an outblock-free path from some node in S to x in the DJ-graph. What we will show in Theorem 3.1 that the set M_x, when the algorithm terminates, does indeed satisfies the Definition 3.2.

Theorem 3.1

Algorithm 3.1 correctly computes $midom(x)$ for any $x \in \mathcal{R}$.

Proof:

First of all it is straightforward to see that M_x is a subset of $Pred_f(x)$. This is because each node y that is added to M_x belongs to $Pred_f(x)$ (step $\boxed{\mathbf{5}}$, and also M_x is initialized to empty set at step $\boxed{\mathbf{1}}$).

Next we will show that all paths from $sidom(x)$ to x (in the flowgraph) contains some node in M_x. The proof is again easy. A node is added to M_x only if **VisitedX**() returns $True$. From Lemma 3.5 it is clear that **VisitedX**() returns $True$ if x is reachable from some node in S. Therefore all paths from $sidom(x)$ to x (in the flowgraph) contains some node in M_x.

Next we will show that for each $y \in M_x$ there is at least one path from $sidom(x)$ (in the flowgraph) that contains y and does not contain any other nodes in $Pred_f(x)$. Again the proof follows from Lemma 3.5. What we do in the Algorithm 3.1 is to outblock all the nodes in $Pred_f(x)$ except node y, and then check if there is a outblock-free from some node in S. From Lemma 3.5 we know that **VisitedX**() returns $True$ if such a path is found. Also this path should contain y (since all other nodes in $Pred_f(x)$ are outblocked). Hence the result.

∎

3.3 Complexity

Here we analyze the worst-case time complexity of computing the $midom$-sets for all the nodes in $\mathcal{R}$. We first give the time complexity of Algorithm 3.1, which computes $midom(x)$ for any $x \in \mathcal{R}$.

Theorem 3.2

The worst-case time complexity of Algorithm 3.1 is $O(|E| \times p)$, where $|E|$ is the number of edges in the DJ-graph, and p is bounded by the maximum number of predecessor nodes of a node in the corresponding flowgraph.

Proof:

For each node y in $Pred_f(x)$ we essentially traverse the DJ-graph below $sidom(x)$ looking for a path to y that does not pass through any other nodes in $Pred_f(x) - y$. Therefore, in the worst case we may visit all the edges in the graph. Also, we perform the above process for each node in $Pred_f(x)$. With p being the maximum number of predecessors a node can have, the result easily follows. ∎

With the above result, we have $O(|E| \times |N| \times p)$ as the time complexity for computing the $midom$-sets for all nodes in $\mathcal{R}$ (in the worst case, $\mathcal{R}$ is N).

3.4 Discussion

- Why DJ-graph? We could have searched for a outblock-free path from $sidom(x)$ to x by simply using an oblivious depth-first (or breadth-first) search. This also gives us the same time bound. However, properties of the DJ-graph allow us to focus on only "promising" paths. To illustrate this consider the previous example, and assume that we are checking whether node 9 is in $midom(13)$. We start the search with some successor node of node 4 (the $sidom(13)$). Assume we pick node 8. From node 8 we could visit any of its successor nodes. The successor nodes of node 8 are node 10 and node 15. Supposing we pick 15, and from this node we could easily wander-off to other irrelevant nodes of the flowgraph (like node 3, etc.). Using DJ-graph this never happens. This is because node 15 is above node 13, and we know that the only way to visit 13 from node 14 is through $sidom(13)$ (which is node 4). So it is safe not to visit any nodes whose level is strictly less than the level of node 13.

  We speculate that, in many cases, the saving due to narrowing of search space can be significant. We are currently working on extending our experimental platform for DJ-graphs (see [SG94]) for the $midom$-set calculation, and will report some preliminary results in the final paper.

- Relation to cut set. We can easily notice that generalized dominators is related to vertex cut-set problem. But computing $midom$-sets is different from general cut-set problem in the following ways. In vertex (node) cut set problem we are given a directed graph, and the problem is to find a subset (usually a minimal subset) of nodes whose removal (along with all the edges incident on these nodes) will split the graph into two disconnected subgraphs. One could use any of the network flow algorithms to compute what is termed as min-cut set that will split the graph into two disconnected components (by assuming all edges have a flow capacity of 1). But in $midom$-set problem for a flowgraph, our objective is to find a minimum subset of nodes from $Pred_f(x)$ whose removal (along with all the edges incident on these nodes) will separate $sidom(x)$ and x (and not necessarily separate the whole graph into disconnected components).

- Open question. Lastly we pose an open question: Can we compute $midom$-set for all nodes in $\mathcal{R}$ in linear time? We currently have no answer, but we conjecture it may be possible to obtain a linear time algorithm to this problem.

4 Identifying Irreducible Multiple-Entry Loops

Testing reducibility of a flowgraph has been a classical exercise in the arena of flowgraph analysis. A variety of definitions have been proposed to capture the notion of "reducible" flow graphs. Formally, a flowgraph G is *reducible* iff the edges of the flowgraph can be partitioned into two disjoint groups, called the forward edges and back edges, with the following two properties [ASU86]:

1. The forward edges form an acyclic graph which every node can be reached from the START.

2. The back edges consists of only edges whose destination node dominates the source node.[6]

[6] These edge classification, into forward and back edges, are done with respect to a depth-first spanning tree. Thus, an edge can still be a back edge without the destination node dominating the source node.

Any flowgraph that does not satisfy the above definition is called an "irreducible" flowgraph [ASU86]. Given the notion of reducible graphs, we can then talk about "natural loops". Given a back edge $n \rightarrow d$ (so that d dominates n), a natural loop is defined as a set of nodes that can reach n without going through d. Node d is called the loop header. Now coming back to reducible graphs, we can think of a flowgraph to be reducible if there are no "jumps" into the middle of loops from outside; the only entry to a loop is through its header [ASU86].

Loop identification is a necessary step in loop transformations for high-performance architectures. The *Tarjan's intervals* are single-entry, strongly connected subgraphs, so they closely reflect the loop structure of a program [Tar74]. They have been used for loop identification. The basic idea behind Tarjan's method is to repeatedly collapse each natural loop into a single node inside-out until the whole graph reduces to one node. This idea will work if the flowgraph is reducible. Recall that a node h is a loop header if $n \rightarrow h$ is a back edge and h dominates n. Now as the reduction process proceeds in Tarjan's method, if we come across a back edge $n \rightarrow g$ such that g does not dominate n, further reduction of the graph cannot be continued. This is because g is not a unique loop header node. A number of actions can be take at this point; one of which is to split the header node, transforming the graph to a reducible graph; another action is to abandon the reduction process and warn the programmer that the graph is not reducible.

What we propose in this paper is to be able to continue the reduction process without node splitting and/or warning the programmer. We will use DJ-graphs to accomplish this. Translating the notion of reducibility on DJ-graph, we can easily observe the following property.

Lemma 4.1

A flowgraph is irreducible if and only if there exists a simple cycle in its DJ-graph that does not contain a BJ-edge (that is, the cycle is made of only D-edges and CJ-edges).

Consider our example flowgraph in Figure 1(a) and its DJ-graph in Figure 1(b). The flowgraph is irreducible because of a multiple-entry loop that has two entry nodes 8 and 9. The DJ-graph contains a simple cycle $3 \rightarrow 6 \rightarrow 7 \rightarrow 12 \rightarrow 3$, in which there are two CJ-edges, $6 \rightarrow 7$ and $12 \rightarrow 3$. The intuition behind this lemma is that if we perform a depth-first search on such a DJ-graph, we can always find a potential back edge that is also a CJ-edge. This observation is not new. Hecht and Ullman have defined reducibility using dominance relation [HU72].

Following Tarjan [Tar74], after performing a depth-first search on a DJ-graph, we can find all the back edges with respect to the depth-first spanning tree. We say $x \rightarrow y$ is an **sp-back** edge iff y is x or is an ancestor of x in the spanning tree.

The order of our reduction process proceeds quit differently from that of Tarjan's method, but the basic philosophy is still the same. One key observation we made in our approach is the following lemma:

Lemma 4.2

All the entry nodes of an irreducible multiple-entry loop have the same immediate dominator.

Proof:

Let $x_1, x_2, \ldots x_n$ be the set of loop entry nodes of an irreducible loop. By definition of loop, there exists a cycle C, such that $x_1, x_2, \ldots x_n$ are in C. Let $y = idom(x_i)$ and let $z = idom(x_j)$, for some $i \neq j \in \{1, 2, \ldots n\}$. We want to show $y = z$. Suppose that $y \neq z$. Then there is a path from START $\xrightarrow{*} z \xrightarrow{+} x_j \xrightarrow{+} x_i$ that does not pass through y, contradicting $y = idom(x_i)$. Therefore, y must be the same as z. ∎

What the above lemma provides is that when looking for candidate nodes that belong to a loop, look only at nodes that are at the same level as loop header and below it (i.e., whose level numbers are greater than the loop header's). This is obvious for a reducible loop, since its unique entry node, the loop header, strictly dominates (or has a smaller level number than) any other nodes in the loop. For an irreducible loop, all its entry nodes collectively dominate any other nodes in the loop. Furthermore, all such entry nodes for an irreducible loop are on the same level. Therefore, we can identify an irreducible loop with entry nodes at a certain level by determining the Strongly Connected Component(SCC) while considering only nodes whose level number is equal to or greater than the current level. Once we find such a SCC, collapse the whole component into one node. Figure 4 gives the complete algorithm for identifying loops.

```
MainLoop()
{
    Perform a depth-first search on the DJ-graph and identify sp-back edges;
    for(i = NumLevel - 1 downto 0) /* visit nodes in a bottom-up fashion */
      Irreducible = False;
      for every node n with n.level = i
        if an edge m → n is both a CJ-edge and an sp-back edge
          Irreducible = True; /* n is part of an irreducible loop */
  /* handle the reducible loop whose header is node n if it exists */
        if n is the destination node of any BJ-edge
          Find ReachUnder(n) for all the BJ-edges m_1 → n, ..., m_k → n
          Collapse the loop consisting of nodes {n} ∪ ReachUnder(n);
        endif
      endfor
      if(Irreducible) /* there exists an irreducible loop */
        Identify SCCs for the subgraphs induced by nodes at level i and below;
        Collapse every irreducible loop which corresponds to a multiple-node SCC;
      endif
    endfor
}
```

Figure 4: Algorithm for identifying loops

The algorithm visits all the nodes level by level. At every level, when there is a reducible loop whose header is at this level, we identify it by checking to see if that node is the destination node of any back edge. If yes, we know a potential reducible loop is found! The algorithm goes to find the loop body with that node being the loop header, and collapse the loop – between step $\boxed{\textbf{7}}$ and step $\boxed{\textbf{10}}$.[7] Note that all the back edges with the same destination node are back edges for the same loop. The algorithm also checks to see if that node could be an entry node of an irreducible loop. If yes, turns on the $Irreducible$ flag at step $\boxed{\textbf{6}}$ — indicating there are some irreducible loops that need to be handled later at step $\boxed{\textbf{12}}$ to step $\boxed{\textbf{15}}$. Once we are done with the above process for every node at the level, we check to see if there is any irreducible loop with its entry nodes at this level at step $\boxed{\textbf{12}}$. If yes, we use SCC's to identify the loop body for an irreducible loop and collapse it. Also note that at this point any reducible loop whose header is at this level or below, has been "collapsed", so has any irreducible loop whose entry nodes are below this level.

For a loop nest consisting of L_1, L_2, L_3 and L_4, where L_1 and L_3 are reducible while L_2 and L_4 are irreducible, we can identify the loop body for all four loops. However, when L_1 and L_4 are reducible while L_2 and L_3 are irreducible, we will only identify three loops by merging L_2 and L_3 into one larger irreducible loop. That is, *immediately nested* irreducible loops will not be distinguished. The advantage of our method is to be able to identify the bodies of nested (reducible and irreducible) loops, with the restriction that a sequence of consecutively nested irreducible loops will be collapsed in a single SCC region. It can expose maximally, in a loop nest, the nesting structure of each portion which is reducible separated by irreducible regions.

Finally we analyze the complexity of the algorithm. When a flow graph is reducible, our algorithm has the same time complexity $T_1 = O(|E| \times \alpha(|E|, |N|))$ as Tarjan's approach [Tar81]. The worst-case time complexity of the algorithm occurs when it needs to find irreducible loops at every level. Assume k is the number of levels in the DJ-graph. Then the time complexity of the algorithm is $T_1 + O(k \times |E|)$, since finding strongly connected components needs $O(|E|)$ time. We anticipate in practice, k is a constant, so the algorithm has T_1 time, which is almost linear.

5 Computing Dominance Frontiers and CONDS Sets

Dominance frontier and control dependence are related to each other; a node $y \in DF(x)$ in the original flowgraph iff x is control dependent on y in the reverse flowgraph. The dual to DF is the $CONDS$ set introduced by Cytron et al [CFS90]. Computing DF is dual to computing $CONDS$ on the reverse flowgraph. In this section we give a simple algorithm for computing the dominance frontier of a node using the DJ-graph that runs in $O(|E|)$ time. For this we need the following lemma which establishes the relation between a node z in $DF(x)$, and the nodes in the dominator subtree rooted at x. This relation is captured by a J-edge $y \to z$, where y is a node in the dominator subtree rooted at x.

Lemma 5.1

A node $z \in DF(x)$ if and only if there exists a node $y \in SubTree(x)$ such that $y \to z$ is a J-edge and $z.level \le x.level$.

[7] By "collapse" we mean that a loop body is condensed and becomes a single node. Any edge incoming into the loop from outside the loop will become an edge incoming into the representative node. Any edge outgoing from the loop to outside the loop will become an edge outgoing from the representative node.

Proof:

For the "only if" part, since z is in $DF(x)$, by definition of dominance frontier, x does not strictly dominate z, and also there must be a node y that is a predecessor of z in the corresponding flowgraph such that x **dom** y. Since x **dom** y, we have $y \in SubTree(x)$. Also, by definition, $y \to z$ is a J-edge. Now, using Theorem 2.1 , it is easy to see that $z.level \leq x.level$.

For the "if" part we have to show that if $y \in SubTree(x)$ and $y \to z$ is a J-edge such that $z.level \leq x.level$, then z is in the set $DF(x)$. There are two cases:

Case 1: z is in $SubTree(x)$. Since $z.level \leq x.level$, z must be x itself. Also, since $y \to z$ is a J-edge, y must be a predecessor of z in the corresponding flow graph. Furthermore, y is in $SubTree(x)$, hence z must be in $DF(x)$ (from the definition of dominance frontier).

Case 2: z is not in $SubTree(x)$. In this case x does not dominate (and hence does not strictly dominate) z. Now since $y \to z$ is a J-edge, y is a predecessor of z in the corresponding flowgraph, from the definition of dominance frontier z is in $DF(x)$.

∎

Using Lemma 5.1 we can easily devise a simple algorithm for computing the dominance frontier of a node as follows:

Algorithm 5.1 *The following algorithm computes dominance frontier $DF(x)$ of a node x using the DJ-graph.*

DomFrontier(x)
{

1: $DF_x = \emptyset$
2: **foreach** $y \in SubTree(x)$ **do**
3: **if**$(y \to z == Jedge)$
4: **if**$(z.level \leq x.level)$
5: $DF_x = DF_x \cup z$

}

For example, consider the DJ-graph shown in Figure 1(b). Suppose we want to determine the dominance frontier of node 3. Nodes in $SubTree(4)$ are 4, 8, 9, 10, 11, 13, and 14. At step $\boxed{\mathbf{3}}$ we find the following J-edges: 8 $\to$ 10, 8 $\to$ 14, 9 $\to$ 10, 9 $\to$ 13, 10 $\to$ 8, 14 $\to$ 13, 11 $\to$ 13 and 13 $\to$ 15. Of these J-edges, we see that only node 15 satisfy the condition at step $\boxed{\mathbf{4}}$. Therefore $DF(4) = \{15\}$.

Next we will show that the space complexity of the above algorithm is $O(|E|)$

Theorem 5.1

The space complexity of Algorithm 5.1 is $O(|E|)$.

Proof sketch:

This is because the size of DJ-graph is linear with respect to the size of flowgraph, and we merely walk on the DJ-graph to compute $DF(x)$, starting from x and noting all the J-edges that come out of the $SubTree(x)$. ∎

Our algorithm is comparable to the algorithm of Cytron et. al. [CFR$^+$91] for computing CONDS relation. CONDS relation is simply dominance frontiers on the reverse flowgraph. Cytron et. al. [CFS90] proposed a simple algorithm for computing the CONDS relation. By contrast, our algorithm can be easily modified to compute the iterated dominance frontier for a node (or a set of nodes) in linear time. We direct interested readers to our companion paper [SG94]. Iterated fork nodes are dual to iterated dominance frontiers on the reverse flowgraph. Iterated fork nodes have been used for computing guarded commands [Wei92]. We can also easily compute iterated fork nodes for a node in linear time using the method proposed in our companion paper.

One point worth noting is that join edges are equivalent to the DF_{local} relation of Cytron et al. [CFR$^+$91]. An edge $x \rightarrow y$ is a join edge iff $y \in DF_{local}(x)$. In the DJ-graph we explicitly represent the DF_{local} relation with join edges.

6 Related Work

DJ-graphs captures uniformly both the dominance relation and the join relation of a flowgraph in a single representation. This paper is inspired by our experience with using DJ-graphs to derive a linear time algorithm for computing ϕ-nodes for an arbitrary sparse evaluation graph [SG94]. The surprising simplicity of that algorithm lead us to believe that DJ-graphs may be useful to solve other problems in traditional flowgraph analysis.

Previously DJ-graphs have been used indirectly for capturing control flow properties of a flowgraph. DF_{local} relation of Cytron et al. [CFR$^+$91] are equivalent to J-edges. An edge $x \rightarrow y$ is a join edge iff $y \in DF_{local}(x)$. In the DJ-graph we explicitly represent the DF_{local} relation with join edges. CD.START and CD.END relations in [CFS90] are again related to J-edges. Recall from Lemma 2.2 that if $x \rightarrow y$ is a J-edge then $idom(y)$ $\mathbf{sdom}$ x. By relating this to CD.START and CD.END, x is CD.START and $idom(x)$ is CD.END.[8]

We have also shown how to use the DJ-graph to facilitate computing the immediate multiple-node dominator of a node. Compared to Gupta's algorithm ours has a better worst-case time complexity for computing the same set. We believe this should make some of the optimizations that uses $midom$-set more practical. We have also shown how one could limit the search space using DJ-graph. We believe this is important. In other cases, although DJ-graphs may not improve on the asymptotic complexity of control flow algorithms, it may reduce the average complexity of certain control flow algorithms. We are not aware of any other work on generalized dominators. Another interesting observation to note is that $midom$-set is related to J-edges in the following sense: For each node y in $midom(x)$ link y to x (do this on

[8] A minor point to note is that Cytron et al. defined CD.START and CD.END on post-dominator tree since they were interested in computing control dependence relation, which is dual to computing dominance frontiers on reverse flowgraph.

the dominator tree of the flowgraph). The resulting graph is a subgraph of DJ-graph. Since we can build DJ-graphs in linear time, we believe it is possible to improve the asymptotic complexity of computing $midom$-set

Identifying loops has been a classical exercise in control flow analysis. We have shown how to use DJ-graphs for identifying both reducible and irreducible loops by extending Tarjan's approach [Tar74]. One feature of our algorithm for identifying irreducible loops is that it utilizes level information in DJ-graphs to discover the body of irreducible loops. In addition, DJ-graphs allow us to better capture the notion of natural loops than Steensgard's approach does [Ste93]. Our algorithm also has a better time complexity than his.

An interesting open problem that was first posed by Cytron et al. [CFS90] is as follows: "Is there a linear space data structure for factoring the dominance frontiers of a flowgraph such that one can determine the set of dominance frontiers of a node in time proportional to the size of the set?" Their conjecture was that it requires a data structure of quadratic space to enumerate the dominance frontier of a node in time proportional to the size of the set. We too believe that the conjecture is true. Our method for determining the dominance frontier of a node also requires only $O(|E|)$ space, but to answer this question again takes $O(|E|)$ time.

7 Conclusion

We have presented a new program representation, the DJ-graph. This representation enables us to design simpler and/or more efficient algorithms for performing three control flow analyses applications. These preliminary results are very encouraging.

Our other future work includes: (1) to apply the DJ-graph to a wide range of program flowgraph analyses and develop it into a robust framework of program representation in program analysis and compiler optimization; and (2) to continue our implementation of the DJ-graph based analysis and validate the algorithms and their performance.

Acknowledgement

We thank Rajiv Gupta for some insightful discussion and clarification on generalized dominators. We also thank Roy Ju for fruitful discussions. The first and the last author thank the National Sciences and Engineering Research Council (NSERC) and the Canadian Centers of Excellence (IRIS) for their continued support of this research.

References

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

[Bur90] Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

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

[CFS90] Ron Cytron, Jeanne Ferrante, and Vivek Sarkar. Compact representations for control dependence. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 337–351, White Plains, New York, June 20–22, 1990. ACM SIGPLAN. Also in *SIGPLAN Notices*, 25(6), June 1990.

[Gup92] Rajiv Gupta. Generalized dominators and post-dominators. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 246–257, 1992.

[Har85] Dov Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Symposium on Theory of Computing*. ACM, May 1985.

[HU72] Matthew S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. *SIAM Journal of Computing*, 1(2):188–202, June 1972.

[LT79] T. Lengauer and Robert Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[PM72] Paul W. Purdom, Jr. and Edward F. Moore. Algorithm 430: Immediate predominators in a directed graph. *Communications of the ACM*, 15(8):777–778, August 1972.

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

[Ste93] Bjarne Steengard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Microsoft Research, October 1993.

[Tar74] Robert E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[Tar81] Robert E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.

[Wei92] Michael Weiss. The transitive closure of control dependence: the iterated join. *ACM Letters on Programming Languages and Systems*, 1(2), June 1992.