McGill University
School of Computer Science
ACAPS Laboratory

Computing ϕ-nodes in Linear Time Using DJ-graphs

Vugranam C. Sreedhar
Guang R. Gao
{sreedhar,gao}@acaps.cs.mcgill.ca

ACAPS Technical Memo 75
January 18, 1994

Advanced Compilers, Architectures and
Parallel Systems Group

A version of this report has been submitted for publication

ACAPS Laboratory • School of Computer Science • 3480 University St. • Montréal • PQ • Canada • H3A 2A7
Correspondence to: V. C. Sreedhar

Contents

Abstract

Contemporary dataflow analysis framework based on sparse evaluation techniques demands fast computation of program points where data flow information must be merged, the so-called **ϕ-nodes**. The original algorithm for computing ϕ-nodes due to Cytron et. al. [CFR$^+$91] has a worst-case quadratic time complexity. More recently, Cytron and Ferrante improved the worst-case behavior of the original algorithm to $O(E \times \alpha(E))$, where E is the number of edges in the flowgraph, and $\alpha()$ is the slowly growing inverse-Ackermann function [CF93]. The problem of finding an algorithm for computing **ϕ-nodes** for arbitrary Sparse Evaluation Graphs in linear time remains open.[1]

In this paper, we present an algorithm for computing ϕ-nodes that runs in *linear* time. This allows us to construct a single Sparse Evaluation Graph (SEG) in time $O(E)$, based on which a wide range of dataflow problems raised in optimizing compilers can be solved. Furthermore, the algorithm is surprisingly simple. We employ a novel program presentation — the DJ graph — by augmenting the dominator tree of a flowgraph with edges which may lead to a potential "merge" of dataflow information. In searching for ϕ-nodes we never visit an edge in the DJ-graph more than once. A prototype implementation of the algorithm has been completed and successfully tested.

1 Introduction

In the recent years, Static Single Assignment (SSA) form [CFR$^+$89, CFR$^+$91], Sparse Evaluation Graphs (SEG) [CCF91], Dependence Flow Graphs (DFG) [JP93], and other related intermediate representations have been successfully used for efficient data flow analysis and program transformations [CLZ86, RWZ88, AWZ88, WZ85, WCES94]. The algorithms for computing these intermediate representations have one common step— computing program points where data flow information must be "merged", the so called **ϕ-nodes**. The original algorithm for computing ϕ-nodes, proposed by Cytron et. al. [CFR$^+$91], consists of first computing the dominance frontiers of the given flowgraph and then using this structure to compute the relevant set of ϕ-nodes. The complexity of the original algorithm depends on the size of the dominance frontier. Although the size of the dominance frontier is linear for many programs (as was noted by Cytron et. al.), there are cases in which the size of the dominance frontier is quadratic in terms of the number of nodes in a flowgraph. This is true even for some cases of structured programs consisting of nested `repeat-until` loops [CFR$^+$91]. Note that, even though the size of the dominance frontier may be quadratic in terms of the number of nodes in the flowgraph, the number of ϕ-nodes that is actually needed is only linear in terms of the number of nodes in the flowgraph (for a particular SEG) [CFR$^+$91]. As Cytron and Ferrante pointed out: "Since one reason for introducing ϕ-nodes is to eliminate potentially quadratic behavior when solving actual data flow problems, such worst case behavior during SEG or SSA construction could be problematic. Clearly, avoiding such behavior necessitates placing ϕ-nodes without computing or using dominance frontiers" [CF93].

[1] More recently, Johnson and Pingali [JP93] have given a linear time algorithm for constructing an SSA like graph, called the Dependence Flow Graph. We compare our work with theirs in Section 7.

To overcome the potential quadratic behavior of computing ϕ-nodes using dominance frontiers, Cytron and Ferrante proposed a new algorithm that has a better complexity than the original algorithm [CF93]. Instead of first computing the dominance frontier and then using this structure for computing ϕ-nodes, Cytron and Ferrante use Tarjan's *balanced path compression* algorithm [Tar79], and combined with other properties that relate dominance relation and depth-first numbering of flowgraph, gave an algorithm that has a time complexity of $O(E \times \alpha(E))$, where $\alpha()$ is the slowly-growing inverse-Ackermann function. (Actually, in the paper, Cytron and Ferrante propose two algorithms— the first one has time complexity of $O(E \times N)$, and the second one has complexity $O(E \times \alpha(E))$.) To the best of our knowledge, the problem of finding an algorithm for computing **ϕ-nodes** for an arbitrary SEG in linear time remains open.

In this paper, we present a *simple* algorithm for computing ϕ-nodes that runs in *linear* time. Given a set of initial nodes N_α, to compute the relevant set of ϕ-nodes, we made one key observation: Consider nodes x and y, where y is an ancestor of x in terms of the dominator tree. If the dominance frontier of x, $DF(x)$, has been computed, then to compute the dominance frontier of y, $DF(y)$, we need not recompute $DF(x)$. However, the reverse may not be true. Therefore, we should order the nodes in the dominator tree such that when the computation of $DF(y)$ is performed, the dominance frontier $DF(x)$ of any its descendent node x, if it is essential for computing the set of desired ϕ-nodes for N_α, has already been computed and is so marked. As a result for any such x, the computation of $DF(y)$ do not require the traversal of the dominator sub-tree rooted at x.

In order to guide the proper node ordering and marking, our algorithm employs a program representation, called the **DJ-graph** (Section 3). The skeleton of the DJ-graph of a program is the dominator tree of its flowgraph (whose edges are called D-edges in this paper— see Section 3). The tree skeleton is augmented with *join edges* (called J-edges in this paper— see Section 3) from the original flowgraph which potentially lead to join nodes where data flow information are merged. The levels of the nodes in the dominator tree are used to order the computation of dominance frontiers of those nodes, x, which are essential to compute the final set of ϕ-nodes— in a bottom-up fashion. Meanwhile, during each computation of $DF(x)$, the descendent nodes of x in the dominator sub-tree rooted at x are visited in a top-down fashion guided by the D-edges, while avoiding nodes which have already been marked. During this top-down visit, the J-edges are used to identify the candidate nodes that should be added into the set of final ϕ-nodes, and recursively explored further. It is important to observe (yet another key observation!) that each new candidate node that is generated on-the-fly always has a level no greater than those nodes currently being processed (assuming that the nodes in the dominator tree are numbered such that the root node has level number 0, and all other nodes in the dominator tree have a level number that is equal to the depth of the node from the root). Therefore, a data structure (called *PiggyBank*— see Section 4) is utilized to keep the candidate nodes in the order of their respective levels, and no nodes are inserted into the *PiggyBank* structure more than once.

We show that our algorithm visits each edge in the DJ-graph at most once, and therefore the complexity is *linear*.[2] The algorithm has been implemented and tested. The linearity of the

[2] As we will show later in the paper (Proposition 3.1), the number of edges in the DJ-graph is no more than

algorithm has been convincingly demonstrated using the ladder graph example (Figure 3) of up to 20,000 nodes.

The rest of the paper is organized as follows: In the next section, we introduce some standard notation and definitions that we will use in the rest of the paper. This section also contains a brief summary of data flow framework and sparse evaluation technique proposed by Choi et. al. [CCF91]. In Section 3, we introduce the DJ-graph. We also discuss some of the properties of this graph that are relevant to our discussion. In Section 4, we give a simple linear algorithm for computing ϕ-nodes. We prove the correctness and the complexity of our algorithm in Section 5. We have implemented a prototype of our linear algorithm. We give some preliminary experimental results in Section 6. In Section 7 we compare our work with related work and finally, in Section 8, we give our conclusion.

2 Background and Notation

In this section, we review standard notation and terminology that are relevant to our discussion. We also briefly describe data flow framework in the context of sparse evaluation methodology [CCF91].

2.1 Terminology and Notation

A Control Flow Graph (CFG) 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 of a CFG. If $x \rightarrow y \in E$, then x is called the *source* node and y is called the *destination* node of the edge. A node in a CFG that has more than one incoming edges is called a *join* node. A *path* of length n is a sequence of edges $(x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_n)$, where each $x_i \rightarrow x_{i+1} \in E$. We use the notation $P : x_0 \xrightarrow{*} x_n$ to represent a path of length zero 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.

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

In a CFG, a node x **dominates** another node y iff all paths from START to y pass through x. We write x **dom** y to indicate that x dominates y, and write x **!dom** y if x does not dominate y. If x **dom** y and $x \neq y$, then x **strictly dominates** y. We write x **sdom** y to indicate x strictly dominates y, and write x **!sdom** y if x does not strictly dominate y. The dominance relation is reflexive and transitive, and can be represented by a tree, called the dominator tree. If x is a parent node of y in the dominator tree, then x **immediately dominates** y, and we write $idom(y)$ to denote the immediate dominator of y. Given a node x in the dominator tree, we define $SubTree(x)$ to be the dominator sub-tree rooted at x. Note that the nodes in $SubTree(x)$ is simply the set of all nodes dominated by x. Figure 1(b) shows the dominator tree for the CFG shown

$|N| + |E|$, where $|N|$ is the number of nodes in the flowgraph and $|E|$ is the number of edges.

in Figure 1(a). For example, for the flowgraph in Figure 1(a), node 4 dominates nodes $\{4, 5, 6\}$, and node 4 strictly dominates $\{5, 6\}$. Also, in the same figure, $idom(5) = 4$, and the nodes in $Subtree(3) = \{3, 9, 10, 11, 12, 13, 14\}$.

For each node in the dominator tree we associate a *level number* that is the depth of the node from the root. We write $x.level$ to indicate the level number of a node x. For the dominator tree shown in Figure 1(b), $START.level = 0$, $1.level = 1$, $2.level = 2$, $5.level = 3$, $12.level = 4$, etc.

A number of algorithms have been proposed for computing dominance relation. Lengauer and Tarjan proposed an algorithm that is almost linear, $O(E \times \alpha(N, E))$, 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 example, consider the control flow graph given in Figure 1(a), $DF(9) = \{3, 15\}$, since 9 dominates 13, which is a predecessor of nodes 3 and 15. Intuitively, one can visualize the dominance frontier as follows: to compute the dominance frontier of x, shine a *light* at the START node, and put a *shade* at the output edges of the node x, so that the light does not pass through them. This process partition the nodes into two groups: light and dark. Consider all edges that go from a dark node to a light node. The set of destination light nodes of these edges form the dominance frontier of x. Note that all the dark nodes are strictly dominated by x.[3]

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

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

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

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

For example, consider again the flowgraph shown in Figure 1(a), $DF^+(9, 8) = \{3, 15, 8, 7, END\}$.

2.2 Data Flow Framework and Sparse Evaluation Graph

There is a general data flow framework in which data flow problems can be expressed [MR90]. Following Choi et. al. [CCF91], this framework is expressed formally as $D =< FG, L, F >$:

[3] We believe the above intuition is due to Ron Cytron and Jeanne Ferrante [Sar93].

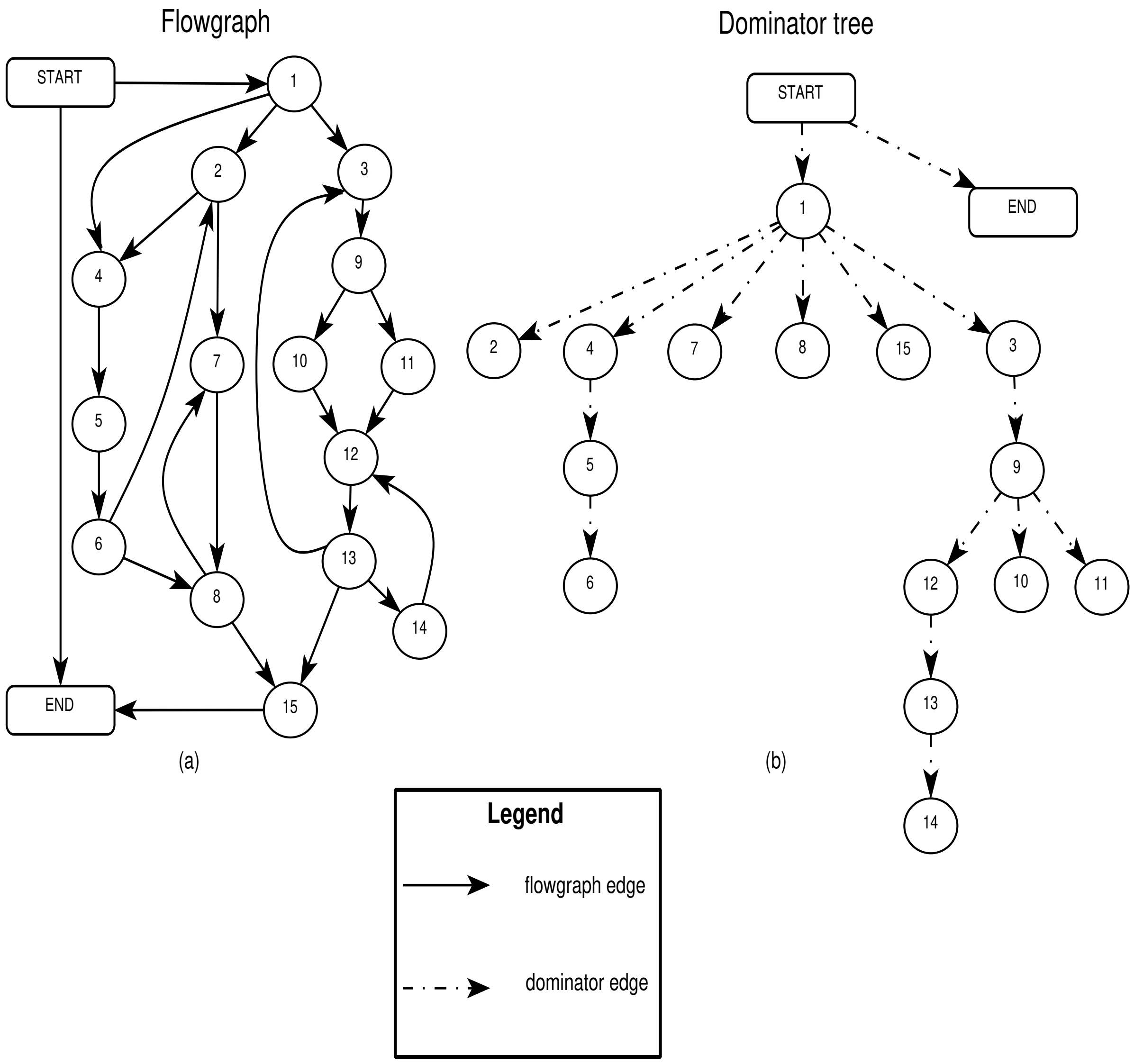


Figure 1: Flowgraph and its dominator tree

Flow Graph $FG =< N_{cfg}, E_{cfg}, root >$ has the same nodes as the CFG, but edges are oriented in the direction of the data flow problem. For forward problems, $FG = CFG$, and $root$ = START, and for backward problems $FG = \overline{CFG}$, where $\overline{CFG}$ is the reverse CFG with all edges reversed in the flow direction, and $root$ = END. In the rest of the paper, we will use the term flowgraph (FG) to refer to the appropriate graph (CFG or $\overline{CFG}$), depending upon the type of data flow problems being solved.

Meet Lattice $L =< A, \top, \bot, \leq, \wedge >$, where

A is a set, whose elements form the domain of the data flow problem,
$\top, \bot$ are 'top' and 'bottom' elements of A,
$\leq$ is a reflexive partial order, and
$\wedge$ is the associative and commutative *meet* operator.

Transfer Functions F is a set of functions on L,

$$F \subseteq \{f : L \rightarrow L\}$$

The local properties of a node y are captured by its transfer function:

$$OUT_y = f_y(IN_y)$$

where $IN_y, OUT_y \in L$, are input and output data flow information at node y, respectively.

The set of functions F can be partitioned into: *constant* transfer function, *identity* transfer function, and *non-identity* transfer functions. The constant transfer function is single-valued and is independent of the input to the transfer function. Identity transfer function at a node cannot affect the data flow information at that node. Sparse evaluation essentially eliminates the flow of information through all identity transfer function nodes. Only nodes that have non-identity transfer function and constant transfer function can affect data flow information at that node.

Choi et. al. gave a procedure for constructing a single SEG for a particular flow analysis problem [CCF91]. The procedure consists of the following steps:

1. Identify all nodes that have either a non-identity transfer function or a constant transfer function. We denote this set of nodes as N_α.
2. Compute the iterated dominance frontier for N_α. The iterated dominance frontier nodes are exactly the nodes where ϕ-functions are inserted [CFR+91]. We denote this set as N_ϕ— the set of ϕ-nodes.
3. Using the nodes $N_\sigma = N_\alpha \cup N_\phi$ construct the sparse graph by directly connecting nodes that *produce* flow information to the nodes that either *use* or *combine* the information. If a node has a constant transfer function then no information propagates to this node, but its output

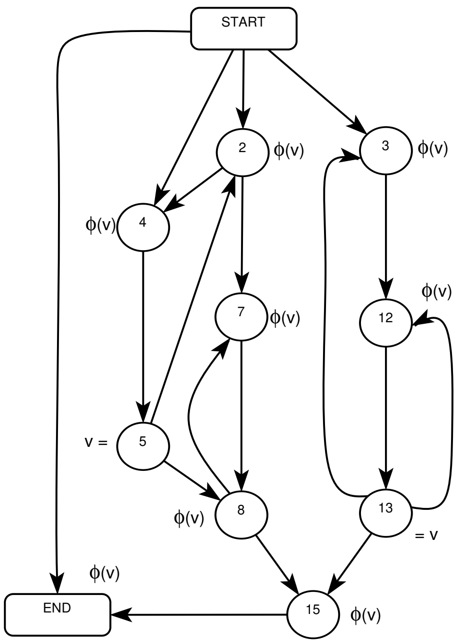


Figure 2: Sparse evaluation graph for v

information is propagated. Nodes that produce or use information are usually the initial set of nodes N_α. Nodes that combine information are the ϕ-nodes. Note that a node in N_α can itself be a ϕ-node.

4. Compute a mapping function that maps identity transfer function nodes to one of the nodes in N_σ. This step is needed for some data flow problems like *live analysis*, but is not needed for other analyses like constructing SSA form.

As an example, consider the flowgraph shown in Figure 1(a). Let us assume N_α equals $\{\text{START}, 5, 13\}$ for some specific data flow problem (say, node 5 defines a variable v and node 13 uses the variable v, and we are interested in computing the SSA graph for v). Note that the transfer function of START node is a constant transfer function, whose output value is always a constant value [CCF91]. Next, we compute $N_\phi = DF^+(\{\text{START}, 5, 13\})$, which is $\{2, 3, 4, 7, 8, 12, 15, \text{END}\}$. Once N_ϕ is computed, we use the set $N_\sigma = N_\alpha \cup N_\phi = \{2, 3, 4, 5, 7, 8, 12, 13, 15, \text{END}\}$ to generate the sparse graph by linking the nodes in N_σ in such a way that any two nodes x and y in N_σ are linked in the sparse graph iff there is a path in the flowgraph from x to y that does not contain any other node in N_σ (other than x and y). For example, node 5 is linked to node 2 in the sparse graph because there is a path from $5 \to 6 \to 2$ in the flowgraph that does not contain any other node in N_σ. The complete sparse graph for the example is shown in Figure 2.

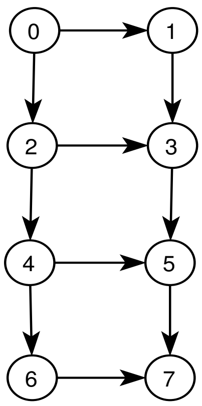
The most time consuming step in the above construction is step 2, that is, computing the set of ϕ-nodes N_ϕ. The original algorithm for computing the relevant set of ϕ-nodes, proposed by

Cytron et. al. [CFR+91], has a worst case time complexity of $O(E + N^2)$. More recently, Cytron and Ferrante [CF93] proposed a more complex algorithm that has a worst case time complexity of $O(E \times \alpha(E))$, where $\alpha()$ is the inverse-Ackerman function [CF93]. In the next section, we will illustrate, using an example, a worst-case scenario of computing ϕ-nodes using the original algorithm.

2.3 The Potential Quadratic Complexity of Computing ϕ-nodes: An Example

Cytron and Ferrante gave an example of a flowgraph (Figure 3(a)) that exhibits worst-case quadratic complexity of the original algorithm for computing ϕ-nodes, based on iterating through the dominance frontier set. Figure 3(b) shows the dominance frontiers for the ladder flowgraph in Figure 3(a). As the ladder size grows, the size of the dominance frontiers of nodes along its left spine increases quadratically, although the size of the sparse graph itself is linear. Suppose one is interested in finding $N_\phi = DF^+(2)$, the original algorithm would iterate through the dominance frontier set, and use the Equation 2 to compute the iterated dominance frontier of node 2.

How can we improve the quadratic complexity of the original algorithm? As will be demonstrated in the rest of this paper, our algorithm avoids the explicit computation of dominance frontiers of all nodes relevant in the definition of iterative dominance frontier $DF^+(2)$. We will come back to this example later in Section 5.3 and Section 6.



(a)

x	DF(x)
0	{}
1	{3}
2	{3,5,7}
3	{5}
4	{5,7}
5	{7}
6	{7}
7	{}

(b)

Figure 3: A ladder graph of size 8, and its dominance frontier

3 The DJ-graph

In this section, we introduce DJ-graphs and discuss some of the properties of DJ-graphs that are relevant to our discussion. This section contains three main results.

1. In Proposition 3.1 we show that the number of edges in a DJ-graph is less than the sum of the number of nodes and the number of edges in the corresponding flowgraph. We will use this result for proving the complexity of our ϕ-node placement algorithm (Section 5.2).
2. In Theorem 3.1 we show that the level number of a node y in $DF(x)$ (the set of dominance frontier of node x) or $DF^{+}(x)$ (the set of iterated dominance frontier of node x), is always less than or equal to the level number of x. This result is one of key points for obtaining our linear time algorithm for placing ϕ-functions. Intuitively, what this result says is that if we want to find what nodes in the flowgraph could be in the dominance frontier (or the iterated dominance frontier) of node x, we only need to look at those nodes whose level number is no greater than that of x. Other nodes (whose level number is strictly greater than the level number of x) can never be in the dominance frontier of x.
3. We give a simple algorithm (Algorithm 3.1) for computing dominance frontier of a node using DJ-graphs and level information. In the next section, we will show how to modify this simple algorithm for computing the relevant set of ϕ-nodes in linear time.

In the rest of this section we formally state and prove these three key results using DJ-graphs. A DJ-graph has 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:

Definition 3.1 (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 given 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) iff $x \to y$ is a **join edge** in the original flowgraph.

Figure 4 shows the DJ-graph for the flowgraph of Figure 1(a). To see how a J-edge is inserted in the dominator tree, consider join node 2 shown in the flowgraph of Figure 1(a). This node consists of $1 \to 2$ and $6 \to 2$ as its two incoming edges. Of these two incoming edges, node 1 strictly dominates join node 2, and so we do not insert an edge from 1 to 2 in the corresponding dominator tree; however 6 does not dominate 2, therefore we do insert an edge from 6 to 2 in the dominator tree. We can easily see that the time complexity for inserting all the J-edges in 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]

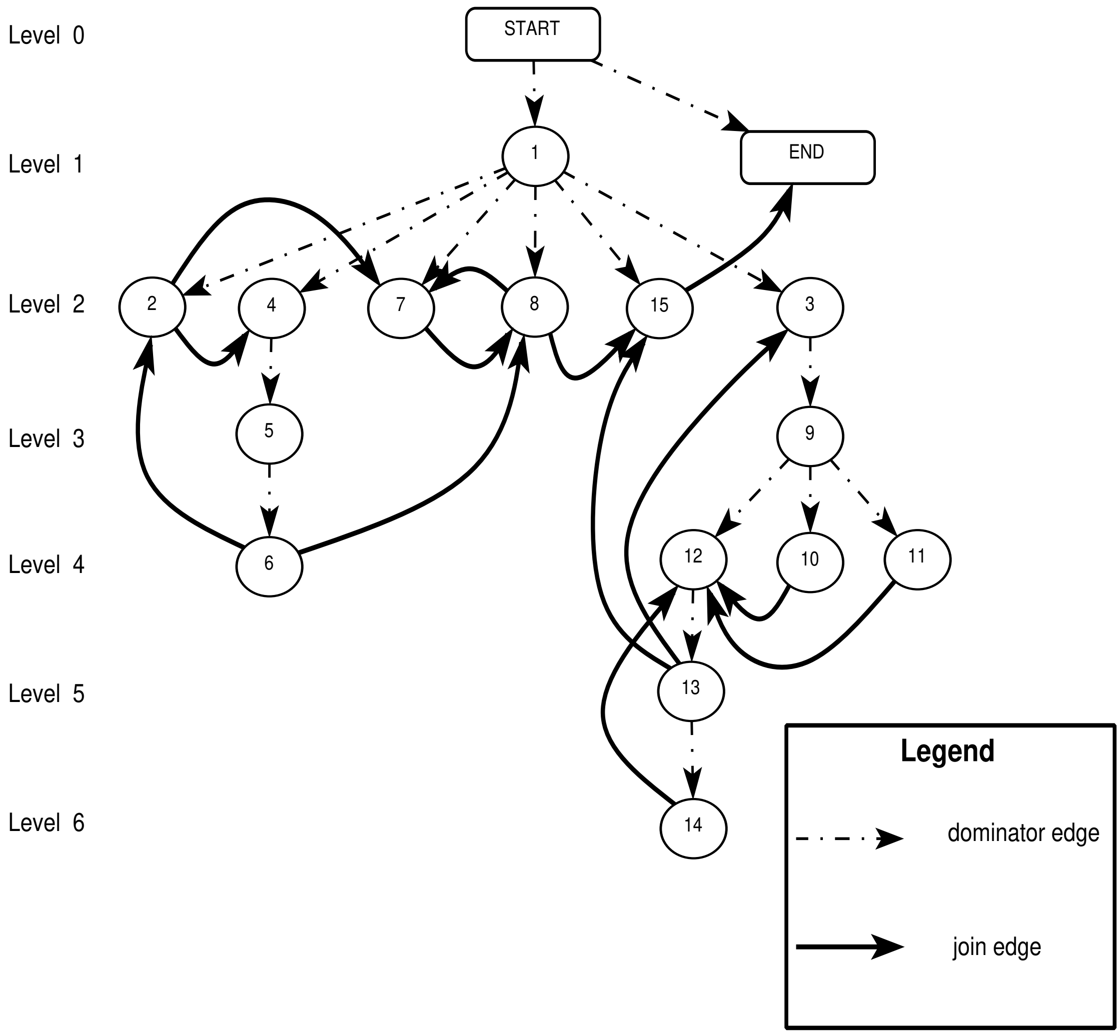


Figure 4: The DJ-graph of the example flowgraph

Notation:
We would like to clarify some terminology that may confuse the reader in later discussion. A DJ-graph is made of the dominator tree and join edges. Whenever we use the term dominator tree, we also mean DJ-graph without J-edges.

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

Since a flowgraph and its DJ-graph consist of the same set of nodes, whenever we use the term *node* x, the node x may belong to either the flowgraph or the DJ-graph, unless explicitly specified using subscript notation, such as f for flowgraph, dj for DJ-graph.

Now that we know how to construct the DJ-graph of a flowgraph, we will next state and prove the three main results mentioned in the beginning of this section. The first result gives an upper bound on the number of edges in the DJ-graph with respect to the size of the flowgraph.

Proposition 3.1

Let $FG = (N_f, E_f)$ be a flowgraph, and let $DJ = (N_{dj}, E_{dj})$ be the corresponding DJ-graph. Then, $|N_{dj}| = |N_f|$ and $|E_{dj}| < (|N_f| + |E_f|)$.

Proof:

The proof is based on the following observation: the DJ-graph has the same set of nodes as the flowgraph, hence $|N_f| = |N_{dj}|$.

The number of edges in the dominator tree of FG is $|N_f| - 1$; thus the number of D-edges in the corresponding DJ-graph is $|N_f| - 1$. The number of J-edges that we introduce in the DJ-graph can be no more than the number of edges in the corresponding flowgraph, hence $|E_{dj}| < (|N_f| + |E_f|)$. ∎

Next we state and prove our second result that relates the level number of a node y in $DF(x)$ with the level number of node x. To prove the result we need the following observation that relates path information in a DJ-graph with respect to the path information in the corresponding flowgraph. For every path $P : x \xrightarrow{*} y$ in a flowgraph there is a corresponding path $P_{dj} : x \xrightarrow{*} y$ in the DJ-graph, and vice versa. This immediately follows from construction of the DJ-graph, and the definition of the dominance relation. Note that the nodes in the two paths may not be exactly the same. Stated more formally, given any two nodes x and y, y is reachable from x in a flowgraph iff y is reachable from x in the corresponding DJ-graph.

Theorem 3.1

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 $\ldots \to x \ldots \to \ldots z \to y$ 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 3.1 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 whose level number is greater than the level number of x in the dominator tree will never be in the dominance frontier of x. The Theorem 3.1 is very important for proving the correctness and complexity of our ϕ-node placement algorithm (Algorithm 4.1).

Finally, we give our third major result of this section— that of computing dominance frontier of a node directly from the DJ-graph. For this we first give a lemma (Lemma 3.1) that establishes the relation between a node z in the dominance frontier of x, $DF(x)$, and the nodes in the dominator sub-tree rooted at x. This relation is captured by a J-edge $y \to z$, where y is a node in the dominator sub-tree rooted at x. We will use this lemma again in Section 5.1 for proving another important lemma (Lemma 5.4), with which we will establish the correctness of our linear algorithm for computing ϕ-nodes (Algorithm 4.1).

Lemma 3.1

A node $z \in DF(x)$ iff there exists a $y \in SubTree(x)$ with $y \to z$ as a J-edge and $z.level \leq x.level$.

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 3.1 , 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 $y \rightarrow z$ 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 $y \rightarrow z$ 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 3.1 we can easily devise a simple algorithm for computing the dominance frontier of a node as follows:

Algorithm 3.1 *The following algorithm computes dominance frontier $DF(x)$ of a node x using DJ-graph.*

♠ **Input:** DJ-graph and a node x whose dominance frontier is to be computed.

♠ **Output:** $DF_x = DF(x)$.

DomFrontier(x)
{
0: $DF_x = \emptyset$
1: **foreach** $y \in SubTree(x)$ **do**
2:   **if**($y \rightarrow z == Jedge$)
3:   **if**($z.level \leq x.level$)
4: $DF_x = DF_x \cup z$
}

For example, consider the DJ-graph shown in Figure 4. Suppose we want to determine the dominance frontier of node 3. The $SubTree(3) = \{3, 9, 10, 11, 12, 13, 14\}$. At step $\boxed{\mathbf{2}}$ we find the following J-edges: $\{10 \rightarrow 12, 11 \rightarrow 12, 13 \rightarrow 3, 13 \rightarrow 15, 14 \rightarrow 12\}$. Of these J-edges, we see that only nodes 3 and 15 satisfy the condition at step $\boxed{\mathbf{3}}$. Therefore $DF(3) = \{3, 15\}$.

We can easily use the Algorithm 3.1 to compute the iterated dominance frontier for the initial set of sparse nodes N_α, by first pre-computing the dominance frontier for all the nodes, and then using the definition of iterated dominance frontier to proceed with the computation. But the time complexity of the resulting algorithm is quadratic. In the next section, we show how to modify Algorithm 3.1 to compute the relevant set of ϕ-nodes in linear time, without actually pre-computing the dominance frontier for all the nodes in the flowgraph.

4 Algorithm for Placing ϕ-nodes

In this section, we give a simple linear algorithm for computing ϕ-nodes using the DJ-graph (Algorithm 4.1). Given a set of initial nodes N_α , the algorithm computes the relevant set of ϕ-nodes by computing the set IDF— the iterated dominance frontier of N_α.

As we indicated in Section 3, a direct application of Algorithm 3.1 based on the inductive definition of iterated dominance frontier can lead to quadratic behavior. Instead, our linear time algorithm is based on two key observations:

1. If $DF(x)$, the dominance frontier of a node x, has already been computed before the computation of $DF(y)$, the dominance frontier of an ancestor node y (in the dominator tree of the program), $DF(x)$ need not be recomputed when computing $DF(y)$. However, the reverse may not be true; therefore the order of the computation is crucial. In Algorithm 4.1 the computation of relevant set of nodes is ordered in such a way that at the time when the computation of $DF(y)$ is performed, $DF(x)$ of any node x within the dominator sub-tree rooted at y has already been computed (and so marked), if $DF(x)$ is essential for computing the set of desired ϕ-nodes for N_α. As a result, the computation of $DF(y)$ need not traverse the dominator sub-tree of x for all such x.
2. When computing $DF(x)$— the dominance frontier of a node x, we only need to examine the J-edges $y \rightarrow z$, where y is a node in the dominator sub-tree rooted at x and z is a node whose level is no greater than the level of x. Recall that we have previously made this observation in the proof of Lemma 3.1. Actually Lemma 3.1 is a stronger version of this key observation.

We employ a data structure called *PiggyBank* to keep the candidate nodes in the order of their respective levels. Based on the above observations, levels of the nodes in the dominator tree will be used in a bottom-up fashion to order the computation of dominator frontiers of those nodes, x, which are essential to compute the final set of ϕ-nodes. Meanwhile, during each computation of $DF(x)$, the successor nodes of x in the dominator sub-tree rooted at x are visited in a top-down fashion guided by the D-edges, while avoiding nodes which have already been marked. During this top-down visit, the J-edges are used to identify the candidate nodes which should be added into the IDF— the set of final ϕ-nodes and those to be recursively explored further. Note that each new candidate generated on-the-fly always has a level number no greater than that of the nodes currently being processed, and we assure that no nodes are inserted into the *PiggyBank* more than once. Therefore, intuitively, we exploit each node (and hence each edge) in the DJ-graph at most once. This, and the structure of the *PiggyBank*, are the basis of the time linearity of our algorithm as will be demonstrated later in Section 5.

PiggyBank is like a piggy-bank, where nodes are temporarily deposited for later withdrawal (See Figure 5). *PiggyBank* is an array of pointers to node structures, with index i storing nodes of level i. Associated with *PiggyBank* are two procedures: **InsertNode**() and **GetNode**().

InsertNode() inserts a node in *PiggyBank* at the index corresponding to the level number of the node. **GetNode**() returns the node whose level number is the maximum of all nodes currently stored in the *PiggyBank*. We first insert the initial set of nodes N_α into the piggy-bank structure. Then, we iteratively compute the dominance frontiers of the nodes in *PiggyBank* in the order that **GetNode**() returns to obtain the iterated dominance frontier of the initial set of nodes N_α. (A node is inserted into *PiggyBank* structure if it is either in N_α or in the iterated dominance frontier of some node in N_α.) We formally prove the correctness the algorithm in Section 5.1, and analyze its complexity in Section 5.2.

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

- A DJ-graph $DJ = (N, E)$, where N is the set of nodes, and E is the set of edges.
- $NumLevel$ is the total number of levels in the dominator tree embedded in the DJ-graph.
- Each node $x \in N$ has the following attributes:

  $struct\ NodeStructure\{$
  $\quad visited = \{Visited, NotVisited\}$
  $\quad alpha = \{Alpha, NotAlpha\}$ `/* in` N_α `or not in` N_α `*/`
  $\quad inphi = \{InPhi, NotInPhi\}$ `/* Checks whether a node is in` N_ϕ `or not */`
  $\quad level = \{0 \dots NumLevel - 1\}$
  $\}$

- Each edge $x \rightarrow y \in E$ has an attribute that specifies the type of the edge: $\{Dedge, Jedge\}$.
- $N_\alpha \subseteq N$ is the initial set of sparse nodes.
- N_ϕ is the set of ϕ-nodes that the algorithm computes, and $IDF = N_\phi = DF^+(N_\alpha)$.
- $Succ(n) = \{m | n \rightarrow m \in E\}$.
- *PiggyBank* is an array of pointers to nodes. Its structure is defined as follows:

  $struct\ PiggyBankStructure\{$
  $\quad NodeStructure\ *node$
  $\quad PiggyBankStructure\ *next$ `/* list of nodes at the same level */`
  $\}\ PiggyBank[NumLevel]$

- **InsertNode**(x) inserts x in $PiggyBank[x.level]$.
- **GetNode**() gets a node stored temporarily in the *PiggyBank*. **GetNode**() always returns a node whose level number (called *CurrentLevel* — see below) is the maximum of all nodes that are currently stored in the *PiggyBank*. **GetNode**() also removes the node from the *PiggyBank* and adjusts the *CurrentLevel* accordingly.
- *CurrentLevel* is initially $NumLevel - 1$, and subsequently has a value that corresponds to the level number of the node that **GetNode**() returns.

- *CurrentRoot* always points to the node that **GetNode**() returns. *CurrentRoot* is equivalent to root of the $SubTree()$ whose dominance frontier is currently being computed.

The first step in the algorithm is to insert all the nodes in N_α into the piggy-bank structure. This is shown below in the **Main** procedure as steps $\boxed{1}$ to $\boxed{4}$. We mark the nodes that are initially inserted into the piggy-bank as *Alpha* to indicate that they belong to the initial set N_α. This is needed to avoid re-inserting these nodes into the piggy-bank again in the future (a condition that we check in the procedure **Visit**(), at step $\boxed{16}$). We then iteratively invoke the procedure **Visit**() on the nodes, x, that **GetNode**() returns to compute the iterated dominance frontier. At step $\boxed{6}$, we assign the variable *CurrentRoot* to point to the node x that **GetNode**() returns in order to keep track of the current root of $SubTree(x)$. Before **Visit**(x) is invoked at step $\boxed{8}$, the node x is marked $Visited$ at step $\boxed{7}$. This marking is crucial because we never visit a node that has been marked $Visited$. We check for this condition in the procedure **Visit**() at step $\boxed{22}$.

Algorithm 4.1 *The following algorithm computes* $N_\phi = DF^+(N_\alpha)$.

♠ **Input:** A DJ graph $DJ = (N, E)$, and the initial set $N_\alpha \subseteq N$ of sparse nodes.

♠ **Output:** The set $IDF = N_\phi = DF^+(N_\alpha)$.

♠ **Initialization:**

- $IDF = \{\}$
- $\forall x \in N$ ($x.visited = NotVisited$;
  $x.inphi = NotInPhi$;
  $x.alpha = NotAlpha$;
  $x.level = Level(x)$) `/* compute level numbers */`
- $CurrentLevel = NumLevel - 1$

♠ **The Algorithm:**

Main()
{
1: **foreach** $x \in N_\alpha$ **do**
2: $x.alpha = Alpha$
3: **InsertNode**(x) `/* Insert the nodes in the` *PiggyBank* `*/`
4: **endfor**
5: **while**((x = **GetNode**()) $! = NULL$)
6: $CurrentRoot = x$
7: $x.visited = Visited$
8: **Visit**(x) `/* Find the dominance frontier of` x `*/`
9: **endwhile**
} `/* EndMain */`

The procedure **Visit**() called with $CurrentRoot$ essentially traverses the dominator sub-tree $SubTree(CurrentRoot)$ in a top-down fashion marking all nodes in the sub-tree as $Visited$ if the nodes are not already marked $Visited$ (a condition that checked at step $\boxed{\textbf{22}}$). Notice that the nodes in the dominator sub-tree are connected through D-edges. As it walks down the sub-tree, the procedure **Visit**() also "peeks" at all nodes that are connected through J-edges, but does not mark them as $Visited$. It only checks the level number of these nodes, and whenever it notices that the level number of a node (that it peeked through a J-edge) is less than or equal to the level number of $CurrentRoot$, it adds the node into the set IDF, if the node is not already in the set (a condition checked at step $\boxed{\textbf{13}}$). It also marks the node as $InPhi$ whenever the node is added to the set IDF. This marking is necessary to avoid adding the node again into IDF whenever it may peek at this node through some other J-edge in the future. It also inserts the node into the piggy-bank structure $PiggyBank$ if the node is not in the set N_α (a condition checked at step $\boxed{\textbf{16}}$).

```
/* visit nodes in SubTree(x) */
Procedure Visit(x)
{
10:  foreach y ∈ Succ(x)
11:     if(x → y == Jedge)
12:        if(y.level ≤ CurrentRoot.level)
13:           if(y.inphi != InPhi) /* Check if y already in N_φ */
14:              y.inphi = InPhi /* y in N_φ */
15:              IDF = IDF ∪ {y} /*Compute the set N_φ */
16:              if(y.alpha != Alpha) /* Check if x is already PiggyBank */
17:                 InsertNode(y) /* Put it in PiggyBank for future search */
18:              endif
19:           endif
20:        endif
21:     else/* x → y is Dedge */
22:        if(y.visited != Visited) /* Avoid redundant visit */
23:           x.visited = Visited
24:           Visit(y)
25:        endif
26:     endif
27:  endfor
} /* EndVisit */
```

The procedures **GetNode**() and **InsertNode**() are given below. **GetNode**() returns a node whose level number is the maximum of all the nodes currently in the $PiggyBank$. **GetNode**() also removes this node from the $PiggyBank$, and adjusts the $CurrentLevel$ accordingly. $CurrentLevel$ keeps track of the level number of the node that **GetNode**() returns. Note that a node will never be inserted in $PiggyBank$ at a level number greater than $CurrentLevel$. As a result, $CurrentLevel$ monotonically decreases through the level numbers. That is, the calls to **Visit**(x) at step $\boxed{\textbf{8}}$ is performed in a bottom-up fashion, in contrast, with each such call, the traversal of the dominator sub-tree rooted at x is performed in a top-down fashion. The marking of the nodes prevents any nodes from being processed more than once in the algorithm. This is essential to ensure the time linearity of the algorithm. We give a proof of correctness and analyze the complexity of the algorithm in the next section.

```
/* This function inserts x in PiggyBank[x.level] */
Procedure InsertNode(x)
{
28:  x.next = PiggyBank[x.level]
29:  PiggyBank[x.level] = x
} /* EndInsertNode */
/* GetNode() returns a node whose level number is the maximum of all
nodes stored in PiggyBank.  It also removes the node from the PiggyBank */
Function GetNode()
{
30:  if(PiggyBank[CurrentLevel] != NULL) /* More nodes in the current level */
31:     x = PiggyBank[CurrentLevel]
32:     PiggyBank[CurrentLevel] = x.next /* delete x from PiggyBank */
33:     return x
34:  endif
     /* Search for nodes up the current level */
35:  for i = CurrentLevel downto 1 do /* level 0 is for START*/
36:     if(PiggyBank[i] != NULL)
37:        CurrentLevel = i /* Update the current level */
38:        x = PiggyBank[i]
39:        PiggyBank[i] = x.next /*Delete x from PiggyBank */
40:        return x
41:     endif
42:  endfor
43:  return NULL /* No more nodes in PiggyBank */
} /* EndGetNode */
```

Next we illustrate Algorithm 4.1 through an example. Consider the DJ-graph shown in Figure 4. Let $N_\alpha = \{13, 5\}$. The first step is to deposit the nodes $\{13, 5\}$ into *PiggyBank*, and to mark these nodes as *Alpha*. After the **for** loop at step $\boxed{1}$, the *PiggyBank* structure would look like Figure 5(a). At step $\boxed{5}$, **GetNode**() returns node 13 (since the level number of node 13 is greater than that of node 5). **GetNode**() also removes 13 from *PiggyBank*. At step $\boxed{6}$, *CurrentRoot* is set to node 13. Node 13 is the root of $SubTree(13)$. In order to find the dominance frontier of node 13 we begin by calling **Visit**(13) at step $\boxed{8}$ (see also step $\boxed{1}$ in Algorithm 3.1). Before that, we also mark node 13 as *Visited* at step $\boxed{7}$.

In the procedure **Visit**(), at step $\boxed{10}$ we find that the successor nodes of 13 to be nodes 3, 15, and 14. Of these the edges $13 \rightarrow 15$ and $13 \rightarrow 3$ are J-edges, and the edge $13 \rightarrow 14$ is a D-edge.

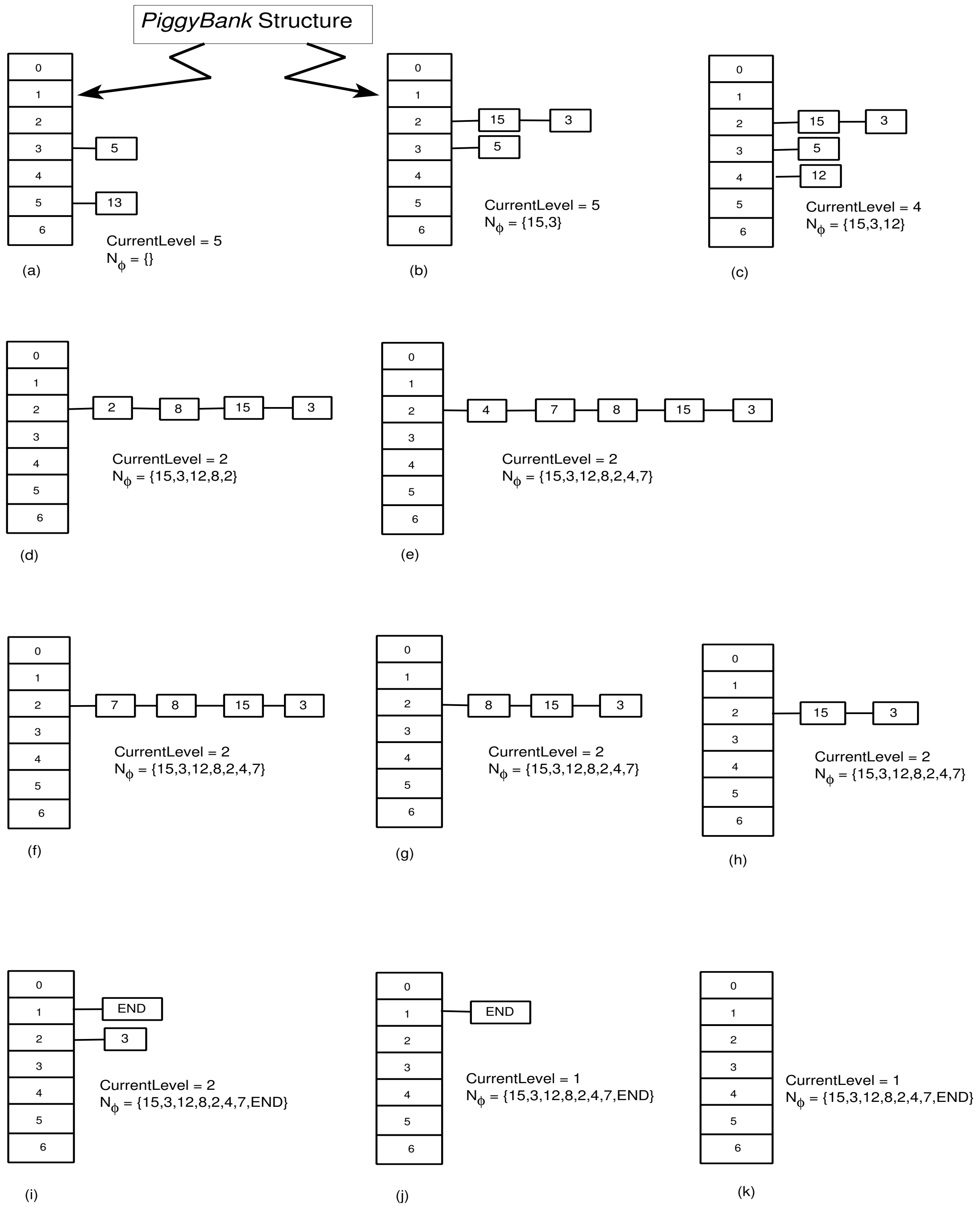


Figure 5: Trace of ϕ-node placement algorithm

Since $15.level = 2$ and $3.level = 2$ are less than $CurrentRoot.level = 13.level = 5$, both nodes 3 and 15 are added to IDF (since the nodes are not already in IDF). Also, neither node 3 nor node 15 is in N_α, and therefore both are inserted into *PiggyBank* (Step $\boxed{\mathbf{17}}$). Figure 5(b) shows the new state of the *PiggyBank*.

Since the edge $13 \rightarrow 14$ is a D-edge, and node 14 is not yet visited, we call **Visit**(14) at step $\boxed{\mathbf{24}}$. Again, before calling **Visit**(14), we mark node 14 as *Visited* (step $\boxed{\mathbf{23}}$). The only successor of 14 is node 12, and $12.level = 4$ is less than $CurrentRoot.level = 13.level = 5$. Also, 12 is neither in IDF nor in N_α, and therefore is added into IDF and inserted into *PiggyBank* (step $\boxed{\mathbf{15}}$ and $\boxed{\mathbf{17}}$). The call to **Visit**(13) terminates and returns at step $\boxed{\mathbf{8}}$.

Now the function **GetNode**() is executed at step $\boxed{\mathbf{5}}$ and it returns node 12. **Visit**(12) is called at step $\boxed{\mathbf{8}}$, and *CurrentRoot* is 12. The only successor of 12 is node 13 ($12 \rightarrow 13$ is a D-edge) and 13 has already been marked *Visited*. So the call to **Visit**(12) terminates and returns at step $\boxed{\mathbf{8}}$.

GetNode() is called again, and this time it returns node 5. **Visit**(5) is called at step $\boxed{\mathbf{8}}$. The only successor of node 5 is node 6, and $5 \rightarrow 6$ is a D-edge, and node 6 is not yet visited. So, **Visit**(6) is called at step $\boxed{\mathbf{24}}$. We can see that node 6 has two successor nodes, 2 and 8, and the edges $6 \rightarrow 2$ and $6 \rightarrow 8$ are J-edges. At step $\boxed{\mathbf{12}}$, the condition is satisfied, as $2.level = 2$ and $8.level = 2$ are less than $CurrentRoot.level = 3$. We can also see that nodes 2 and 8 are not in IDF, and are not in N_α. Therefore, the two nodes are added to IDF, and inserted into *PiggyBank*. After this, *PiggyBank* looks as in Figure 5(d). Again, **Visit**(5) terminates, and **GetNode**() is called at step $\boxed{\mathbf{5}}$, and the process continues. Figure 5 shows the complete trace of *PiggyBank* structure for the example.

5 Correctness and Complexity

In this section, we first give a proof of correctness (Theorem 5.1), and then analyze the complexity of the algorithm (Theorem 5.2).

5.1 Correctness

The main theorem which establishes the correctness of Algorithm 4.1 is Theorem 5.1. The theorem states that the algorithm computes the iterated dominance frontiers of the set N_α. The inductive proof of the theorem is based on a major lemma (Lemma 5.4), which establishes the fact that when the algorithm calls **Visit**(x) at step $\boxed{\mathbf{8}}$ and the call terminates, all nodes in the dominance frontiers $DF(x)$ are already added into the set IDF (a fact used both in the induction basis and induction steps). Let x be the current root of a dominator sub-tree $SubTree(x)$ visited by **Visit**(x) at step $\boxed{\mathbf{8}}$. Let z be in $DF(x)$. Lemma 3.1, introduced earlier in Section 3, guarantees that there must exist a node y in $SubTree(x)$ such that $y \rightarrow z$ is a J-edge and $level.z \leq level.x$. Another lemma, Lemma 5.3, states that y will already have been marked *Visited* when **Visit**(x) returns. There are two cases in the algorithm where a node can be marked $Visited$: 1. at step $\boxed{\mathbf{23}}$, and 2. at step $\boxed{\mathbf{7}}$. The validity of Lemma 5.4 for case 1 is straightforward. For case 2, y must be marked $Visited$ by an earlier call of **Visit**(v) for a node v in $SubTree(x)$. This fact is made possible because of the *PiggyBank* structure and we formalize this in Lemma 5.1 and Lemma 5.2. We then make an inductive argument on the decreasing level of the nodes to demonstrate that all nodes in $DF(v)$ should already be inserted into IDF by this time. The node z should also be in IDF according to Lemma 3.1. From this the validity of Theorem 5.1 is established.

In our chain of proofs, we begin with Lemma 5.1, which states that a node can never be inserted in *PiggyBank* at an index greater than the level number of the current root node $CurrentLevel$. We will use this fact to prove Lemma 5.2.

Lemma 5.1

A node is never inserted in the PiggyBank at an index that is greater than $CurrentLevel$.

Proof:

There are only two places (in the algorithm) that a node can be inserted in the *PiggyBank*: at step $\boxed{\mathbf{3}}$ and at step $\boxed{\mathbf{17}}$. Since the initial value of $CurrentLevel$ is $NumLevel - 1$, the level number of any node that is insert at step $\boxed{\mathbf{3}}$ can never be greater than $NumLevel - 1$.

At step $\boxed{\mathbf{17}}$, a node y is inserted in the *PiggyBank* only if it is visited through a J-edge and if $y.level \leq CurrentLevel$. Therefore y is never inserted in the *PiggyBank* at an index that is greater than $CurrentLevel$. ∎

Lemma 5.2 gives an order (based on the level number of nodes) in which calls to **Visit**(), at step $\boxed{\mathbf{8}}$, can be performed. The ordering of nodes is controlled by calls to **GetNode**() at step $\boxed{\mathbf{5}}$. Recall that **GetNode**() always returns a node whose level number is the maximum of all nodes currently stored in the *PiggyBank* structure. We will use this lemma later in the proof of our main lemma (Lemma 5.4).

Lemma 5.2

Let x and y be any two nodes that are inserted in the PiggyBank and later removed (and returned) from the PiggyBank by **GetNode**() *at step* $\boxed{\mathbf{5}}$. *If $y.level > x.level$, then* **Visit**(y) *will be called earlier than* **Visit**(x) *at step* $\boxed{\mathbf{8}}$.

Proof:

First of all observe that **Visit**(), at step $\boxed{\mathbf{8}}$, is always called on the node that **GetNode**() returns at step $\boxed{\mathbf{5}}$. Also, we know that **GetNode**() always returns a node whose level number is the maximum of all the nodes currently in the *PiggyBank*. Also, we know from Lemma 5.1 that a node will never be inserted in *PiggyBank* at an index greater than $CurrentLevel$. From this we prove validity of the lemma as follows: There are two cases:

Case 1 Before **GetNode**() returns either of the two nodes, both nodes x and y are in the $PiggyBank$. Naturally y will be returned earlier to x (since **GetNode**() always returns a node whose level number is the maximum).

Case 2 Before **GetNode**() returns either of the two nodes, only one of the two nodes is in the $PiggyBank$. Let x be in the $PiggyBank$. This means that either y was already inserted and removed from the $PiggyBank$, even before x was inserted into the $PiggyBank$, in which case the validity of the lemma is true, or y will be inserted in future. The latter situation is impossible since, from Lemma 5.1, y should be inserted into the $PiggyBank$ at a level number that is less than or equal to the level number of x. (Recall that we have assumed $y.level > x.level$). We can make a similar argument by assuming that y is in the $PiggyBank$.

∎

The next lemma establishes an important fact that when a node x is visited by a call of **Visit**(x) from step $\boxed{\mathbf{8}}$ and returned, that all nodes in the dominator sub-tree of x $SubTree(x)$ have been marked $Visited$. Intuitively, this means that when such a visit returns, none of the nodes in the $SubTree(x)$ have been overlooked.

Lemma 5.3

When **Visit**(x) *returns at step* $\boxed{\mathbf{8}}$*, all nodes in* $SubTree(x)$ *are marked* $Visited$*.*

Proof:

Using Lemma 5.2 and induction on levels of the nodes which can be x in the call to **Visit**(x) at step $\boxed{\mathbf{8}}$, we can easily prove the lemma. The base case is when the node x has the maximum level; the validity of the lemma is straightforward. Assume that the lemma is true for all **Visit**(x) returned at step $\boxed{\mathbf{8}}$ with $x.level \geq k$ for some k. Now, assume we examine **Visit**(x) with $x.level = k - 1$. From our observation above, a top down traversal of nodes in dominator sub-tree $SubTree(x)$ will be performed during the execution of **Visit**(x). Assume y is the next node to be probed at step $\boxed{\mathbf{22}}$. One of the following two cases will be encountered:

Case 1: The next node y is not marked $Visited$. Then the program will continue to mark it $Visited$ via a recursive call to **Visit**() at step $\boxed{\mathbf{24}}$.

Case 2: y is already marked $Visited$. y could only have been marked $Visited$ by some earlier call to **Visit**(y), and this call must have been invoked at step $\boxed{\mathbf{8}}$, and not at step $\boxed{\mathbf{24}}$ (since the nodes are visited in a top-down fashion and there can be only one D-edge). But since $y.level \geq x.level$ (i.e. $y.level \geq k$), by induction on k, we know **Visit**(y) has marked y and all descendents of y. (This is because **Visit**(y) rooted at y was called at step $\boxed{\mathbf{8}}$ prior to the call of **Visit**(x)).

∎

It is easy to see from Lemma 5.2 and Lemma 5.3, that calls to **Visit**() at step $\boxed{\textbf{8}}$ are made in a bottom-up fashion and while each recursive call at step $\boxed{\textbf{24}}$, the recursive procedure **Visit**() visits the nodes in the dominator tree in a top-down fashion.

Lemma 5.4 is the main lemma which shows how the procedure **Visit**() captures the dominance frontier of a node in the set IDF. Intuitively, the lemma states that when **Visit**(x) is called and terminated at step $\boxed{\textbf{8}}$ all the nodes in the dominance frontier of x are added to the set IDF. We will use this property in the main theorem (Theorem 5.1) to inductively argue the correctness of Algorithm 4.1.

Lemma 5.4

> *When* **Visit**(x) *is called with* x *as the* $CurrentRoot$ *and returned at step* $\boxed{\textbf{8}}$, *all the nodes in* $DF(x)$ *are also in the set* IDF.

Proof:

Let x be the current root of the $SubTree(x)$ visited by a call to **Visit**(x) at step $\boxed{\textbf{8}}$, and terminated. Also let z be in $DF(x)$. From Lemma 3.1 there must exist a node y in $SubTree(x)$ such that $y \rightarrow z$ is a J-edge and $z.level \leq x.level$. Since y is in $SubTree(x)$, from Lemma 3.1, y is marked $Visited$. As in the proof of Lemma 5.3 , there are two cases:

Case 1: y is marked $Visited$ by the current **Visit**(x) invoked at step $\boxed{\textbf{8}}$. Then, a recursive call at step $\boxed{\textbf{24}}$ will cause its children (in the dominator tree) to be explored subsequently at step $\boxed{\textbf{10}}$. Since $y \rightarrow z$ is a J-edge, z will be included in IDF at step $\boxed{\textbf{15}}$.

Case 2: y is not marked $Visited$ by the current **Visit**(x) invoked at step $\boxed{\textbf{8}}$. Since nodes in $SubTree(x)$ are visited in a top-down fashion, there must be a node u such that x **sdom** u and u **dom** y, and u is not marked $Visited$ by the current call to **Visit**(x) (invoked at step $\boxed{\textbf{8}}$). That is, u is marked $Visited$ by a prior call of **Visit**(u) also invoked at step $\boxed{\textbf{8}}$. If $u = y$, z will be added to the set IDF, since $y \rightarrow z$ is a J-edge and $z.level < u.level$. If $u \neq y$, then y must be visited at step $\boxed{\textbf{22}}$ via a D-edge. A subsequent call of **Visit**(y) will add z to the set IDF, since $y \rightarrow z$ is a J-edge and $z.level < u.level$.

∎

Notice that the above lemma only says that **Visit**(x), when it returns at step $\boxed{\textbf{8}}$, will have added the entire dominance frontier of x to IDF. It does not specify which of the nodes in the set IDF belong to $DF(x)$. Notice that the set IDF can contain nodes that are not in the set $DF(x)$. In the other words, **Visit**() does not *explicitly* compute the dominance frontier of a node.

There is an important subtle point in the proof of Case 2 of Lemma 5.4. Note that we have argued that "u is marked $Visited$ by a call of **Visit**(u) at step $\boxed{\mathbf{8}}$.", the reader may wonder how we can be sure such a u does exist. Is it possible for all the nodes from y up to x in the $SubTree(x)$ to have been marked $Visited$ by some previous call of **Visit**() via a D-edge at step $\boxed{\mathbf{24}}$? The answer is no! This is because we visit nodes in the $Subtree(x)$ in a top-down fashion and pass through a node by an explicit check at step $\boxed{\mathbf{22}}$ to see if it is not yet marked $Visited$. This fact is important, as u is now assured to have been called earlier from step $\boxed{\mathbf{8}}$ afresh from $PiggyBank$. With u at the root of such an earlier call **Visit**(u), all the nodes in $DF(u)$ must have been examined and put into the set IDF. And our Piggybank structure ensures **Visit**(u) happens before **Visit**(x) (Lemma 5.1). Otherwise, the algorithm may fail– we will come back to this issue again in Section 5.3.

Finally, we prove that when the algorithm terminates $IDF = DF^+(N_\alpha)$.

Theorem 5.1

Algorithm 4.1 correctly computes $N_\phi = DF^+(N_\alpha)$.

Proof:

We will show that, when Algorithm 4.1 terminates, $IDF = DF^+(N_\alpha)$. From now on let $S = N_\alpha$. First of all it is obvious that $DF(S) \subseteq IDF$ (Lemma 5.4). Now we need to show that if $DF_i(S) \subseteq IDF$ then $DF_{i+1}(S) \subseteq IDF$, where

$$DF_{i+1}(S) = DF(S \cup DF_i(S))$$

Rewrite the above equation as

$$DF_{i+1}(S) = DF(S) \cup DF(DF_i(S))$$

We know $DF(S) \subseteq IDF$. So we are left to show $DF(S')$ is in IDF, where $S' = DF_i(S)$. Let $S' = \{x_1, \ldots x_i, \ldots\}$. Since S' is in IDF (assumption), x_i is in IDF for all i. But the only way the node x_i can be added to IDF is at step $\boxed{\mathbf{15}}$ of the algorithm. Therefore, x_i must also be inserted into the $PiggyBank$, at step $\boxed{\mathbf{17}}$. Since the algorithm eventually terminates, x_i must be processed as the current root at step $\boxed{\mathbf{8}}$ by a call to **Visit**(x_i). By Lemma 5.4, $DF(x_i)$ is in IDF when **Visit**(x_i) returns at step $\boxed{\mathbf{8}}$. And this is true for all nodes $x_i \in S'$. Therefore $DF(S')$ is in IDF. As a result $DF_{i+1}(S)$ is in IDF, and

$$DF^+(N_\alpha) \quad \subseteq IDF \tag{3}$$

We can easily see from Lemma 5.4 that a node is inserted in IDF if it is in the dominance frontier of some node that was previously inserted and retrieved from $PiggyBank$. Recall that a node is inserted into $PiggyBank$ only at two places: step $\boxed{\mathbf{3}}$ and step $\boxed{\mathbf{17}}$. A node is inserted in $PiggyBank$ at step $\boxed{\mathbf{3}}$ if it is in N_α, and a node is inserted in $PiggyBank$ at step $\boxed{\mathbf{17}}$ if it is in the iterated dominance frontier of N_α. Therefore,

$$IDF \subseteq DF^+(N_\alpha) \tag{4}$$

From Equation 3 and 4, we get

$$DF^{+}(N_{\alpha}) \quad = \quad IDF \tag{5}$$

∎

5.2 Complexity

Next we will show that the time complexity of Algorithm 4.1 is $O(|E|)$, where $|E|$ is the number of edges in the DJ-graph. Recall that the number of edges in the DJ-graph is less than $|N_f| + |E_f|$ (Proposition 3.1). Therefore, the time complexity of Algorithm 4.1 is $O(|N_f| + |E_f|)$. Since $|E_f| \geq |N_f| - 1$, the time complexity of the algorithm is $O(|E_f|)$, which is linear with respect to the number of edges in the flowgraph.

From the proof of the correctness of Algorithm 4.1, readers may have already observed that for any node x in the DJ-graph, the node may be processed by a call of **Visit**(x) (which may happen at step $\boxed{\mathbf{8}}$ or $\boxed{\mathbf{24}}$) at most once. This observation is a key to the proof of linearity of the algorithm, and is stated as the following lemma.

Lemma 5.5

When Algorithm 4.1 terminates, a node $x \in N$ may be processed by a call to **Visit***(x) at most once.*

Proof:

There are only two places a node can be processed by a call to **Visit**(): **case 1.** at step $\boxed{\mathbf{24}}$ and **case 2.** at $\boxed{\mathbf{8}}$. It is obvious that step $\boxed{\mathbf{24}}$ can only be reached by the traversal of an incoming D-edge of x. Since there is only one such D-edge, x cannot not be processed more than once in case 1. Furthermore, it cannot be processed through case 2 more than once either (since a node can be inserted into and deleted from Piggybank only once).

Now we prove that it is not possible for x to be processed in both case 1 and case 2. Suppose the contrary is true. Then this is only possible if case 2 happens after case 1. (The condition at step $\boxed{\mathbf{22}}$ prevents the opposite.) That means node x is already in the *PiggyBank* before the current execution of **Visit**(v) for some v at step $\boxed{\mathbf{8}}$. Thus, $x.level \geq v.level$, since $SubTree(v)$ is explored in a top-down fashion. On the other hand, $x.level \leq v.level$, as any node in the *PiggyBank* must have a level number no greater than that of the node currently being processed (from Lemma 5.1). This implies $x = v$. But this is impossible since x would been marked *Visited* twice from step $\boxed{\mathbf{8}}$. Hence the lemma is true by contradiction. ∎

From the above proof, it also true that a node can never be marked $Visited$ more than once. We will use the above lemma in proving the complexity of the algorithm.

Theorem 5.2

The time complexity of Algorithm 4.1 is $O(|E|)$.

Proof:

According to Lemma 5.5, a node can be marked $Visited$ at most once, and there can be at most $|N|$ calls to **Visit**(). Also, at each node in the procedure **Visit**(), we either visit (through a D-edge) or "peek" (through a J-edge) all the successor nodes (step $\boxed{\mathbf{10}}$) only once. This means that we have effectively probed all the edges in the DJ-graph at most once. Hence the complexity of the algorithm is $O(|E|)$. ∎

An acute reader may ask the following question: What is the complexity of inserting and deleting nodes into/from the $PiggyBank$ structure? It is easy to see that the complexity of inserting a node in the $PiggyBank$ is $O(1)$. As for the complexity of getting a node from the $PiggyBank$, it is again easy to see that a node will never be inserted in $PiggyBank$ at the index greater than the $CurrentLevel$ (from Lemma 5.1). Each call of **GetNode**() will execute the **for** loop with a monotonically decreasing $CurrentLevel$ from $NumLevel - 1$ **down-to** 1 during successive calls for the entire duration of the algorithm (follows from Lemma 5.1). Hence the overall complexity of deleting nodes from the $PiggyBank$ is, in the worst case, $O(|N|)$.

5.3 Discussion

There are a number of key issues in our algorithm that we would like to summarize in this section.

- The key point that makes our new algorithm linear is the piggy-bank structure $PiggyBank$. If one were to use other structures such as a linked-list, a stack or a queue, either the proof of correctness would fail (if we still wish to continue to mark nodes as $Visited$ using one color), or the complexity of the algorithm would not be linear (we will need to mark nodes as $Visited$ using more than one color). The second situation similar to finding the iterated dominance frontier by iteratively applying Algorithm 3.1. We can easily show that the complexity of this method will be quadratic.

  To fully understand the above discussion, the readers are encouraged to apply the algorithm, with $N_\alpha = \{0, 2\}$, to the "ladder graph" example shown in Figure 3 whose DJ-graph is shown in Figure 6. Try to use a linked-list structure to replace the $PiggyBank$ structure, and assume node 0 is visited before node 2.

- Let us recall Theorem 3.1. Theorem 3.1 states that if $y \in DF(x)$, then the level number of y will never be greater than the level number of x (i.e., $y.level \leq x.level$). This property is very important in proving the correctness and complexity of our algorithm. Recall that Lemma 3.1 establishes the relation for computing the dominance frontier of a node from level information and J-edges, and Lemma 5.1 guarantees that a node will never be inserted at an index (in $PiggyBank$) greater than the value of $CurrentLevel$. Again, an acute reader will immediately notice the relation between these two lemmas and Theorem 3.1.

6 Implementation

We implemented a prototype of our algorithm and tested it on many graphs. Our test program suite consisted of a set of randomly generated flowgraphs. Our objective was to test the validity of the algorithm and check its asymptotic behavior. We are currently integrating the prototype implementation into an experimental compiler system. We intend to perform sparse evaluation for solving data flow problems.

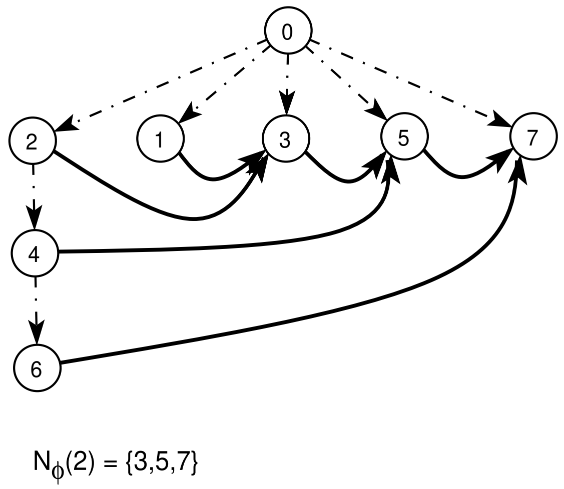


Figure 6: DJ-graph for the ladder example

To illustrate the linear behavior of our algorithm, we ran our algorithm on increasingly taller ladder graphs of the form shown in Figure 3, and whose DJ-graph is shown in Figure 6. We chose the ladder graph because previous algorithms for this graph have running times that are not linear in the size of the flowgraph. We tested the implementation on SPARC 10/14 work-station, and compiled the prototype implementation using GCC version 2.4.5, with -O2 compiler optimization option. Figure 7 shows the running time of our algorithm. The times shown are the average over 10 consecutive runs. We can see that the asymptotic behavior of our algorithm is *linear* with respect to the size of the ladder graph. For the graph shown in Figure 3, we chose $N_\alpha = \{2\}$.

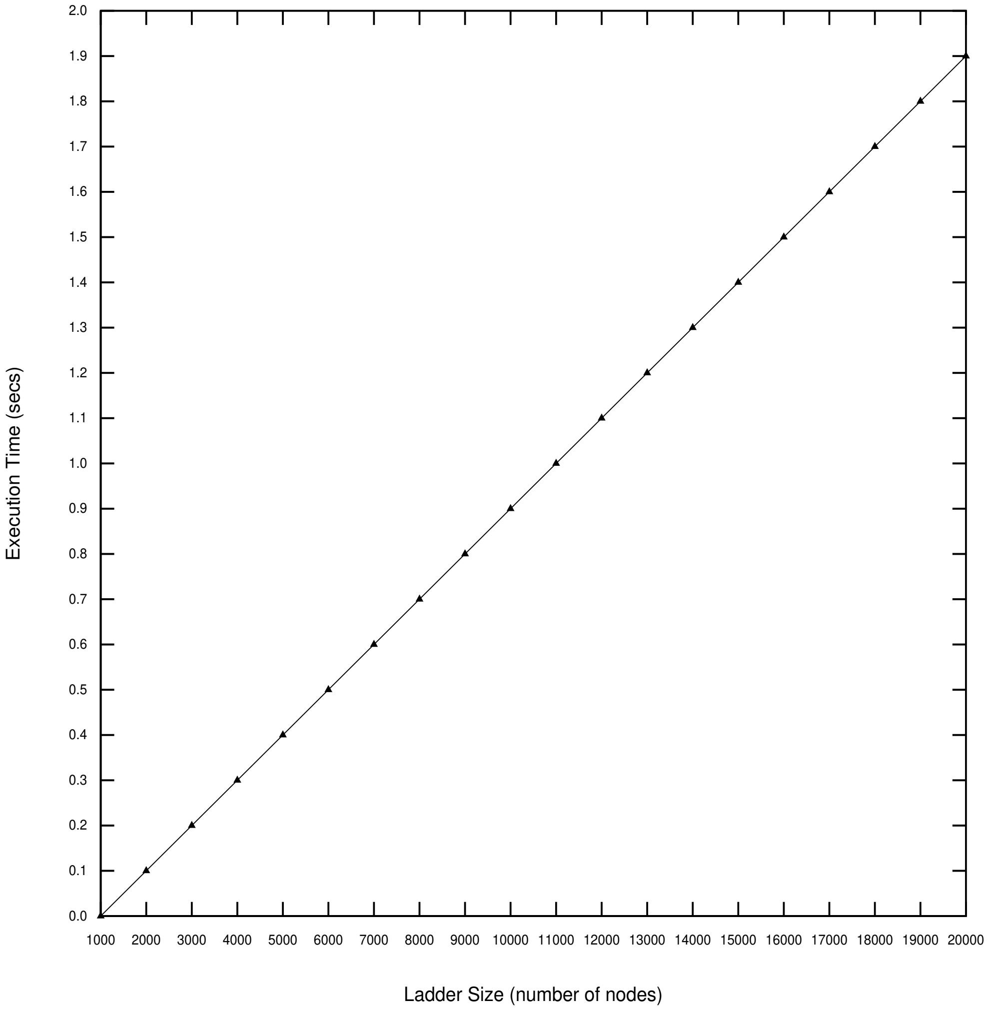


Figure 7: Performance of our algorithm on ladder graph

7 Related Work

The sparse evaluation technique is becoming popular, especially for analyzing large programs. To this end, many intermediate representations have been proposed in the literature for performing sparse evaluation [CFR+91, CCF91, JP93, WCES94]. The algorithms for constructing these intermediate representations have one common step— determining program points where data flow information must be merged (the so called ϕ-nodes). The notion of ϕ-nodes dates back to the work of Shapiro and Saint [SS70] (as noted in [CFR+91]). Subsequently, others have proposed sparse evaluation in one form or another that is related to work of Shapiro and Saint [RT82, CF87]. Cytron et. al. [CFR+89] gave the first algorithm for computing ϕ-nodes for arbitrary flowgraphs. The time complexity of the algorithm depended on the size of the dominance frontier, which is $O(N^2)$. Recently, Cytron and Ferrante improved the quadratic behavior of computing ϕ-nodes to be almost linear time. The time complexity of the new algorithm is $O(E \times \alpha(E))$, where $\alpha()$ is the inverse-Ackermann function [CF93]. It seems that the algorithm has not been implemented [CF93], and, since the algorithm is not exactly linear, more experimental studies are needed to evaluate the performance of this algorithm when applied to real programs.

Johnson and Pingali recently proposed an algorithm for constructing SSA-like representation called the Dependence Flow Graph (DFG) [JP93]. To construct DFG they first compute regions of control dependence. Using this information they determine single-entry-single-exit regions. Then they perform, for each variable, an inside-out traversal of these regions, computing dependence information and inserting `switch` and `merge` nodes, whenever dependences cross regions of control dependence. The authors have shown that the running time of the algorithm for constructing DFG is $O(E)$. One can easily construct the SSA form from the DFG by simply eliminating `switch` nodes in the DFG. Although, the method of Johnson and Pingali can be used for constructing the SSA form in time $O(E \times V)$ (where V is the number of program variables) [JP93], it has the same problem as the SSA form, i.e. the DFG and the SSA form cannot be used for solving arbitrary data flow problems (for example, *liveness* analysis), as noted in [CF93].

Our algorithm can be used to construct arbitrary SEG's. Compared to any of the previous work, our algorithm reduces the time complexity of constructing a single SEG to $O(E)$. Also, we can use our algorithm to construct SSA form or DFG in time $O(E \times V)$.

We are not aware of any other algorithm for computing ϕ-nodes. There is much related work that uses SSA like representation, for example, the Program Dependence Web [BMO90] and the Value Dependence Graph [WCES94], and our algorithm could improve the complexity of constructing these related intermediate representations. Also there are many optimizations that use SSA form for efficient implementation, for example, constant propagation [WZ85], value numbering [RWZ88], register allocation [Bri92], code motion [CLZ86], etc. Our algorithm could improve the overall running time of these optimizations.

In this paper we have employed a new program representation— the DJ-graph. Derived from a flowgraph, the DJ-graph can be viewed as a refinement representing explicitly and precisely both the dominator relation between nodes (via D-edges) and the potential program points where the dataflow information may be merged (via J-edges). As demonstrated in this paper, DJ-graphs have facilitated the development of our algorithm. Furthermore, some properties of DJ-graphs make much easier the proofs of the correctness and linearity of our algorithm.

The DJ-graphs can also be applied to program analysis other than computing ϕ-nodes, but that is beyond the scope of the present paper. We direct interested readers to our companion paper for more details [SGL94].

8 Conclusion

In this paper, we have provided a positive answer to the open problem posted in the introduction; i.e., it is indeed possible to design an algorithm for computing ϕ-nodes in linear time. This is a good news for work which depends on efficient dataflow analysis — as computing ϕ-nodes is a key step in constructing a sparse dataflow evaluation framework. Furthermore, the algorithm presented in this paper is very simple.

We proved the correctness of our algorithm and its complexity. Our algorithm uses the properties of a new program representation called the DJ-graph which facilities its design and analysis. We benefit from the simplicity of the algorithm in its implementation. We presently have constructed a prototype implementation of the algorithm and performed experiments on a number of random generated flowgraphs. We are currently integrating the implementation into a real compiler. A comparison with other related work is presented. We also plan to explore properties and applications of that DJ-graph that would be useful for other program analyses.

Acknowledgement

We thank Russell Olsen for commenting on an earlier draft of the paper. We thank Erik Altman who simplified one of the proofs and suggested many improvements in the writing of the paper. We owe much thanks to Yong-fong Lee for his insightful technical discussions and also for critically commenting on drafts of the paper. His comments greatly improved the presentation of the paper. Finally, we thank the National Sciences and Engineering Research Council (NSERC) for their continued support of this research.

References

[ACM88] ACM SIGACT and SIGPLAN. *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, California, January 13–15, 1988.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages* [ACM88], pages 1–11.

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The program dependence web: A representation supporting control-, and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, page 257, White Plains, New York, June 20–22, 1990. ACM SIGPLAN. Also in *SIGPLAN Notices*, 25(6), June 1990.

[Bri92] Preston Briggs. *Register Allocation via Graph Coloring.* PhD thesis, Rice University, Houston, Texas, April 1992.

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

[CF87] Ron Cytron and Jeanne Ferrante. What's in a name? or the value of renaming for parallelism detection and storage allocation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, St. Charles, Illinois, August 17–21, 1987.

[CF93] Ron Cytron and Jeanne Ferrante. Efficiently computing ϕ-nodes on-the-fly. In *Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, August 1993.

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

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

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

[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.

[JP93] Richard Johnson and Keshav Pingali. Dependence-based program ananlysis. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 78–89, 1993.

[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.

[MR90] Thomas Marlowe and Barbara Ryder. Properites of data flow frameworks: A unified model. *Acta Informatica*, 28:121–163, 1990.

[RT82] J. H. Reif and Robert Tarjan. Symbolic program analysis in almost linear time. *SIAM Journal of Computing*, 11(1):81–93, February 1982.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages* [ACM88], pages 12–27.

[Sar93] Vivek Sarkar, 1993. Personal Communication.

[SGL94] Vugranam Sreedhar, Guang Gao, and Yong-fong Lee. DJ-graph and its applications to flowgraph analysis. Technical report, McGill University, January 1994. In preparation.

[SS70] R. M. Shapiro and H. Saint. The representation of algorithm. Technical Report CA-7002-1432, MCA, 1970.

[Tar79] Robert Tarjan. Applications of path compression on balanced trees. *JACM*, 26(4):690–715, October 1979.

[WCES94] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, 1994.

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