A Linear Time Algorithm for Placing ϕ-Nodes

Vugranam C. Sreedhar **Guang R. Gao**
School of Computer Science
McGill University
Montreal H3A 2A7
Canada
{sreedhar,gao}@acaps.cs.mcgill.ca

Dataflow analysis framework based on Static Single Assignment (SSA) form and Sparse Evaluation Graphs (SEGs) demand fast computation of program points where data flow information must be merged, the so-called **ϕ-nodes**. In this paper, we present a surprisingly simple algorithm for computing ϕ-nodes for arbitrary flowgraphs (reducible or irreducible) that runs in *linear* time. We employ a novel program representation — 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 by guiding the search of nodes by their levels in the dominator tree.

The algorithm has been implemented and the results are compared with the well known algorithm due to Cytron et al. [CFR+91]. A consistent and significant speedup has been observed over a range of 46 Fortran procedures taken from a number of benchmark programs. We also ran experiments on increasingly taller ladder graphs and confirmed the linear time complexity of our algorithm.

1 Introduction

Static Single Assignment (SSA) form [CFR+91], Sparse Evaluation Graphs (SEGs) [CCF91], and other related intermediate representations have been successfully used for efficient data flow analysis and program transformations [RWZ88, AWZ88, WZ85, Wol92, 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**. Given a flowgraph, the original algorithm for computing ϕ-nodes for an SEG consists of the following steps [CFR+91, CCF91]:

1. Precompute the dominance frontier $DF(x)$ for each node x. A node y is in $DF(x)$ if x dominates a predecessor of y without strictly dominating y.
2. Determine the initial set of 'sparse' nodes N_α that represent non-identity transference in a data flow framework. For SSA, such nodes contain *definitions* of variables [CFR+91].
3. Compute the iterated dominance frontier $IDF(N_\alpha)$ for the initial set N_α. Cytron et al. have shown that the desired set of ϕ-nodes for an SEG is same as the iterated dominance frontier $IDF(N_\alpha)$ of the initial set [CFR+91].

The time 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, for example, 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 needed remains linear (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]. To overcome the potential quadratic behavior of computing ϕ-nodes using dominance frontiers, Cytron and Ferrante proposed a $O(E \times \alpha(E))$ algorithm that does not use dominance frontiers.[1] 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.[2]

In this paper, we present a linear time algorithm for computing the desired set of ϕ-nodes for N_α without precomputing the dominance frontiers for all the nodes. One key feature of our linear time algorithm is to order the nodes in the dominator tree in such a way that when the computation of dominance frontier $DF(y)$ is performed, the dominance frontier $DF(x)$ of any its descendant node x, if it is essential for computing the desired set of ϕ-nodes for N_α, has already

POPL '95 1/ 95 San Francisco CA USA

[1] $\alpha()$ is the slowly-growing inverse-Ackermann function.

[2] 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.

been computed and is so marked. As a result for any such x, the computation of $DF(y)$ does not require the traversal of the dominator sub-tree rooted at x.

To perform the proper node ordering and marking, our algorithm uses a novel 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 which are essential to compute the final set of ϕ-nodes in a bottom-up fashion. We show that our algorithm visits each edge in the DJ-graph at most once, and therefore the complexity is *linear* in the size of the input flowgraph.[3]

The algorithm has been implemented on the top of Parafrase2 compiler [Har85b]. To compare our results, we also implemented the original algorithm based on iterating through dominance frontiers [CFR+91]. We experimented on a number of FORTRAN procedures taken from Perfect, Eispack, and other programs. With our algorithm we were able to obtain, on average, more than five-fold speedup over the original algorithm. We also tested our algorithm against the standard ladder graph example [CF93]. Again our algorithm exhibited a linear behavior compared to the quadratic behavior of the original algorithm.[4]

The significance of the algorithm presented in this paper goes beyond to merely computing ϕ-nodes for SEGs or SSA form. Our framework can be used for the computation of guards [Wei92]. More recently we have used iterated dominance frontiers to incrementally update dominator trees [SGL94b]. Finally, our framework is robust enough to support incremental computation of ϕ-nodes [SGL94b].

Organization. In the next section, we introduce some standard notation and definitions that we will use in the rest of the paper. 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 time algorithm for computing ϕ-nodes. We show the correctness and the complexity of our algorithm in Section 5. In Section 6, we present an implementation of our algorithm and report results for a number of programs. We also report results for the ladder graph example. In Section 7, we compare our work with related work and finally, in Section 8, we give our conclusion.

2 Background and Notation

A **flowgraph** 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 flowgraph. If $x \rightarrow y \in E$, then x is called the *source* node and y is called the *destination* node of the edge. We will assume that every node in N is on some path from START to END.

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

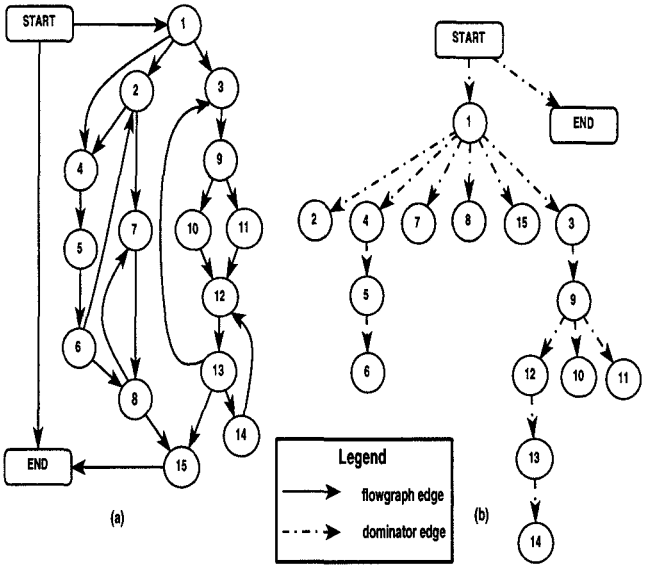

Figure 1: Flowgraph and its dominator tree

In a flowgraph, 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 flowgraph shown 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 of the tree. We write $x.level$ to indicate the level number of a node x. For example, for the dominator tree shown in Figure 1(b), START$.level = 0, 8.level = 2, 9.level = 3$, etc.

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

$$DF(S) = \bigcup_{x \in S} DF(x) \quad (1)$$

We define the **iterated dominance frontier** $IDF(S)$ for a

[3] As we will show later in the paper, the number of edges in the DJ-graph is no more than $|N| + |E|$, where $|N|$ is the number of nodes in the flowgraph and $|E|$ is the number of edges.

[4] Due to the complex nature and partial description of Cytron and Ferrante's almost linear time algorithm we did not implement that algorithm.

set of nodes S as the limit of the increasing sequence:

$$IDF_1(S) = DF(S), \quad (2)$$
$$IDF_{i+1}(S) = DF(S \cup IDF_i(S)) \quad (3)$$

Let $N_\alpha \subseteq N$ be the initial set of "sparse" nodes [CF93]. Cytron et al. have shown that the desired set of ϕ-nodes for N_α is exactly same as $IDF(N_\alpha)$ [CFR+91]. Therefore in the rest this paper we present a linear time algorithm for computing the iterated dominance frontiers for $N_\alpha \subseteq N$.

3 DJ-Graphs and Their Properties

In this section we briefly introduce DJ-graphs and state some of the properties of DJ-graphs relevant to our discussion. We also give a simple algorithm for computing the dominance frontiers for a node using DJ-graphs. We will then show how to compute the dominance frontiers for a set of nodes without precomputing the dominance frontiers for all nodes. In the next section we will show how to extend this simple algorithm to compute the relevant set of ϕ-nodes in linear time.

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 $x \rightarrow y$ 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. [5] 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 $x \rightarrow y$ is a **join edge** in the original flowgraph.

Figure 2 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 $1 \rightarrow 2$ and $6 \rightarrow 2$ 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 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.[6]

In the rest of the section we discuss some of the properties of DJ-graphs. Due to space reason we only outline the basic properties and direct interested readers to the full paper [SG94] for a more thorough discussion on this.

[5] Another view of D-edges and J-edges may give a better intuition: We *mark* each edge $x \rightarrow y$ in the flowgraph as an immediate dominance edge if $x = idom(y)$. The edges that are *not* marked are join edges.

[6] Note that we can construct the dominator tree of a flowgraph in linear time [Har85a].

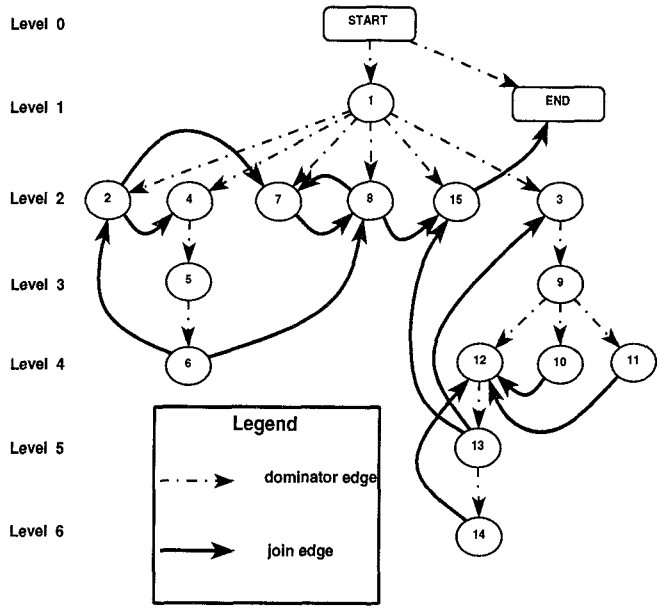

Figure 2: The DJ-graph of the example flowgraph

1. 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 [SG94]. We will use this result in the complexity analysis of our ϕ-node placement algorithm (Section 5.2).

2. Let $y \in DF(x)$ (and thus is also in $IDF(x)$). Then the level number of y is always less than or equal to the level number of x [SG94]. This result is one of the 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 a 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. A node y is in $DF(x)$ iff there is a node $z \in SubTree(x)$ and a J-edge $z \rightarrow y$ such that the level of y is less than or equal to the level of x (Lemma 3.1). Using this property we next give a simple algorithm (Algorithm 3.1) for computing dominance frontiers.

Computing the Dominance Frontier for a Single Node

Our algorithm for computing the dominance frontier of a node is based on Lemma 3.1 which establishes a relation between a node $z \in DF(x)$, and the nodes in the dominator sub-tree rooted at x. This relation is captured by a J-edge $y \rightarrow z$, where y is a node in the $SubTree(x)$.

Lemma 3.1 *A node $z \in DF(x)$ iff there exists a $y \in SubTree(x)$ with $y \rightarrow z$ as a J-edge and $z.level \leq x.level$.*

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 the dominance frontier $DF(x)$ of a node x using DJ-graphs.*

```
DomFrontier(x)
{
0:  DF_x = ∅
1:  foreach y ∈ SubTree(x) do
2:    if((y → z == Jedge) and
3:            (z.level ≤ x.level))
4:       DF_x = DF_x ∪ z
}
```

For example, consider the DJ-graph shown in Figure 2. Suppose we want to determine the dominance frontier of node 3. The $SubTree(3) = \{3, 9, 10, 11, 12, 13, 14\}$. At step [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 [3]. Therefore $DF(3) = \{3, 15\}$.

Computing the Dominance Frontiers for a Set of Nodes

We can easily use the Algorithm 3.1 to compute the dominance frontier for a set of nodes S, by first pre-computing the dominance frontiers for all the nodes, and then using Equation 1 to proceed with the computation To illustrate this consider the computation of $DF(\{9, 12\})$. By Equation 1, we know $DF(\{9, 12\}) = DF(9) \cup DF(12)$. Let us therefore precompute $DF(9)$ and $DF(12)$. Using Algorithm 3.1 we get $DF(9) = \{3, 15\}$, and $DF(12) = \{3, 12, 15\}$. Therefore $DF(\{9, 12\}) = \{3, 12, 15\}$.

Notice in the above example that we visit the nodes in the $SubTree(12)$ twice— once during the computation of $DF(9)$ and once again during the computation of $DF(12)$. How can we avoid this redundant visitation of the nodes in the $SubTree(12)$? We can avoid this by first computing $DF(12)$ and marking the node 12 as being processed. Now during the computation of $DF(9)$ we avoid visiting any nodes in the $SubTree(12)$ (since node 12 is already processed, and is so marked) thereby avoiding redundant visitation. Notice here that we never need to precompute $DF(9)$ and $DF(12)$ in order to compute $DF(\{9, 12\})$. Therefore in order to compute $DF(\{9, 12\})$, we first compute the $DF(12)$ using Algorithm 3.1, and also mark node 12 as being processed. Any candidate nodes that is generated on-the-fly is then added to the set $DF(\{9, 12\})$. Now during the computation of $DF(9)$ we avoid visiting the nodes in the $SubTree(12)$. Again we add any candidate nodes that is generates on-the-fly to $DF(\{9, 12\})$. Based on this observation we can see that the ordering of the nodes in the dominator tree is important to avoid redundant visitation of nodes during the computation of dominance frontiers.

In the next section, we will show how to extend the above key observation to compute the relevant set of ϕ-nodes in linear time without pre-computing the dominance frontier for all the nodes in the flowgraph. Notice that one can still use Algorithm 3.1 for computing the relevant set of ϕ-nodes by precomputing the dominance frontiers for all the nodes. But the time complexity of the resulting algorithm will be quadratic.[7]

4 Algorithm for Placing ϕ-Nodes

In this section, we give a simple linear time algorithm for computing ϕ-nodes using DJ-graphs (Algorithm 4.1). Given a set of initial nodes N_α, the algorithm computes the relevant set of ϕ-nodes by computing the set $IDF(N_\alpha)$, 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. Let y be an ancestor node of a node x on the dominator tree. If $DF(x)$ has already been computed before the computation of $DF(y)$, $DF(x)$ 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 is 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 $DF(x)$ 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 Lemma 3.1.

We employ a data structure called the $PiggyBank$ 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 dominance frontiers of those nodes, x, which are essential to compute the final set of ϕ-nodes. Meanwhile, during each computation of $DF(x)$, the descendant 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 node currently being processed, and we assure that no nodes are inserted into the $PiggyBank$ more than 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.

[7] An example of a flowgraph which exhibit quadratic behavior is the ladder graph. We will discuss more on this later in Section 6.

The $PiggyBank$ is like a piggy-bank, where nodes are temporarily deposited for later withdrawal (See Figure 3). The $PiggyBank$ is an array of list of nodes, with index i storing nodes of level i. Associated with the $PiggyBank$ are two procedures: **InsertNode**() and **GetNode**(). **InsertNode**() inserts a node in the $PiggyBank$ 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 $PiggyBank$. Then, we iteratively compute the dominance frontiers of the nodes in the $PiggyBank$ 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 the $PiggyBank$ 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:

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

$$
\begin{array}{l}
struct\ NodeStructure\{ \\
\quad visited = \{Visited, NotVisited\} \\
\quad alpha = \{Alpha, NotAlpha\} \\
\quad inphi = \{InPhi, NotInPhi\} \\
\quad level = \{0 \ldots NumLevel - 1\} \\
\}
\end{array}
$$

- Each edge $x \rightarrow y \in E$ has an attribute that specifies the type of the edge: $\{Dedge, Jedge\}$.
- The $PiggyBank$ is an array of pointers to nodes. Its structure is defined as follows:

$$
\begin{array}{l}
struct\ PiggyBankStructure\{ \\
\quad NodeStructure\ *node \\
\quad PiggyBankStructure\ *next \\
\}\ PiggyBank[NumLevel]
\end{array}
$$

- $CurrentLevel$ 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 $PiggyBank$ structure. This is shown below in the **Main** procedure as steps 1 to 4. We mark the nodes that are initially inserted into the $PiggyBank$ as $Alpha$ to indicate that they belong to the initial set N_α. This is needed to avoid re-inserting these nodes into the $PiggyBank$ again in the future (a condition that we check in the procedure **Visit**(), at step 16). We then iteratively invoke the procedure **Visit**() on the nodes that **GetNode**() returns to compute the iterated dominance frontier. At step 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 8, the node x is marked $Visited$ at step 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 22.

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

♠ **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:**

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

♠ **The Algorithm:**

```
Main()
{
/* Insert N_α into the PiggyBank */
1:  foreach x ∈ N_α do
2:     x.alpha = Alpha
3:     InsertNode(x)
4:  endfor
    /* repeat until no more nodes
         in the PiggyBank */
5:  while((x = GetNode()) != NULL)
6:     CurrentRoot = x
7:     x.visited = Visited
8:     Visit(x)
9:  endwhile
} /* EndMain */
```

The procedure **Visit**(), called with the current root $CurrentRoot$, 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 checked at step 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 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 $PiggyBank$ if the node is not in the set N_α (a condition checked at step 16).

```
Procedure Visit(x)
{
10:  foreach y ∈ Succ(x)
11:    if(x → y == Jedge)
12:      if(y.level ≤ CurrentRoot.level)
         /* Check if y already in N_φ */
13:        if(y.inphi != InPhi)
14:          y.inphi = InPhi /* y in N_φ */
         /* Compute the set N_φ */
15:          IDF = IDF ∪ {y}
16:          if(y.alpha != Alpha)
       /* Do not reinsert if y ∈ N_α */
17:            InsertNode(y)
18:          endif
19:        endif
20:      endif
21:    else/* x → y is Dedge */
       /* Avoid redundant visit */
22:      if(y.visited != Visited)
23:        y.visited = Visited
24:        Visit(y)
25:      endif
26:    endif
27:  endfor
} /* EndVisit */
```

InsertNode() inserts a node into the $PiggyBank$ at an index equal to the level number of the node.

```
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 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 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.

```
Function GetNode()
{ /* More nodes in the current level */
30:  if(PiggyBank[CurrentLevel] != NULL)
31:    x = PiggyBank[CurrentLevel]
     /* delete x from PiggyBank */
32:    PiggyBank[CurrentLevel] = x.next
33:    return x
34:  endif
35:  for i = CurrentLevel downto 1 do
36:    if(PiggyBank[i] != NULL)
       /* Update the current level */
37:      CurrentLevel = i
38:      x = PiggyBank[i]
       /* Delete x from PiggyBank */
39:      PiggyBank[i] = x.next
40:      return x
41:    endif
42:  endfor
       /* No more nodes in PiggyBank */
43:  return NULL
} /* EndGetNode */
```

Example: Next we illustrate Algorithm 4.1 through an example. Consider the DJ-graph shown in Figure 2. Let $N_\alpha = \{5, 13\}$. The first step is to deposit the nodes 5 and 13 into the $PiggyBank$, and also mark them as $Alpha$. After the **for** loop at step 1, the $PiggyBank$ would look like Figure 3(a). At step 5, the function **GetNode**() returns node 13. (**GetNode**() also removes 13 from the $PiggyBank$.) At step 6, $CurrentRoot$ is set to node 13. To find the dominance frontier of node 13 we call **Visit**(13) at step 8. Prior to this, we also mark node 13 as $Visited$ at step 7.

In the procedure **Visit**(), at step 10 we find that the successor nodes of 13 to be nodes 3, 15, and 14. Of these, $13 \rightarrow 15$ and $13 \rightarrow 3$ are J-edges, and $13 \rightarrow 14$ is a D-edge. Since $15.level = 2$ and $3.level = 2$ are less than $CurrentRoot.level = 13.level = 5$, nodes 3 and 15 are added to IDF (since they are not already in IDF). Also, neither 3 nor 15 is marked $Alpha$ (and hence not in N_α), both the nodes are inserted into the $PiggyBank$ (Step 17). Figure 3(b) shows the new state of the $PiggyBank$.

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

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

GetNode() is called again, and this time it returns node 5. **Visit**(5) is called at step 8 and the process continues.

Figure 3 shows a partial trace of the $PiggyBank$ for the example.

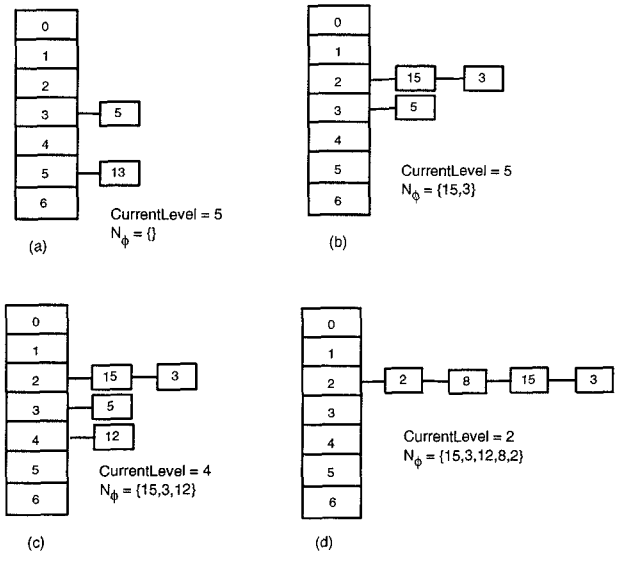

Figure 3: Partial trace of ϕ-node placement algorithm

5 Correctness and Complexity

In this section we establish the correctness of Algorithm 4.1 and analyze its complexity. Due to space reasons, we state the main theorems (Theorem 5.1 and Theorem 5.2) and only give intuitive sketch of their proof structures. We also state all the supporting lemmas needed and state their intuitive meaning. All proofs can be found in [SG94].

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 [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 $SubTree(x)$ visited by **Visit**(x) at step [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 [23], and (2) at step [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 the $PiggyBank$ at an index greater than the level number of the current root node $CurrentLevel$. We 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 $CurrentLevel$.*

Lemma 5.2 gives an order (based on the level number of nodes) in which calls to **Visit**(), at step [8], can be performed. The ordering of nodes is controlled by calls to **GetNode**() at step [5]. Recall that **GetNode**() always returns a node whose level number is the maximum of all nodes currently stored in the $PiggyBank$ structure.

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 [5]. If $y.level > x.level$, then* **Visit**(y) *will be called earlier than* **Visit**(x) *at step [8].*

The next lemma establishes an important fact that when a node x is visited by a call of **Visit**(x) from step [8] and returned, that all nodes in the dominator $SubTree(x)$ 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 [8], all nodes in $SubTree(x)$ are marked $Visited$.*

It is easy to see from Lemma 5.2 and Lemma 5.3, that calls to **Visit**() at step [8] are made in a bottom-up fashion and while each recursive call at step [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 [8] all the nodes in the dominance frontier of x are added to the set IDF.

Lemma 5.4 *When* **Visit**(x) *is called with x as the $CurrentRoot$ and returned at step [8], all the nodes in $DF(x)$ are also in the set IDF.*

Notice that the above lemma only says that **Visit**(x), when it returns at step [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 $DF(x)$.

Theorem 5.1 *Algorithm 4.1 correctly computes the set of ϕ-nodes $N_\phi = IDF(N_\alpha)$.*

The proof of the theorem easily follows from the above lemmas.

5.2 Complexity

Next we will show that the time complexity of Algorithm 4.1 is $O(|E|)$. Recall that the number of edges in the DJ-graph is less than $|N_f| + |E_f|$. 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 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 8 or 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.*

From the above lemma, one can see that a node can never be marked $Visited$ more than once, and there can be at most $|N|$ calls to **Visit**(). Recall that 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 10) only once. This means that we have effectively probed all the edges in the DJ-graph at most once. Hence one can see that the complexity of the algorithm is $O(|E|)$.

Theorem 5.2 *The time complexity of Algorithm 4.1 is $O(|E|)$.*

An acute reader may ask the following question: What about the complexity of inserting and deleting nodes into/from the $PiggyBank$? 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 the $PiggyBank$ 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

Recall that one of the key point that makes our algorithm linear is the $PiggyBank$ structure. 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 the nodes as $Visited$ using one color), or the complexity of the algorithm would not be linear (we will need to mark the nodes as $Visited$ using more than one color). The second situation is 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 4(a) whose DJ-graph is shown in Figure 4(c). Try to use a linked-list structure to replace the $PiggyBank$ structure, and assume node 0 is visited before node 2.

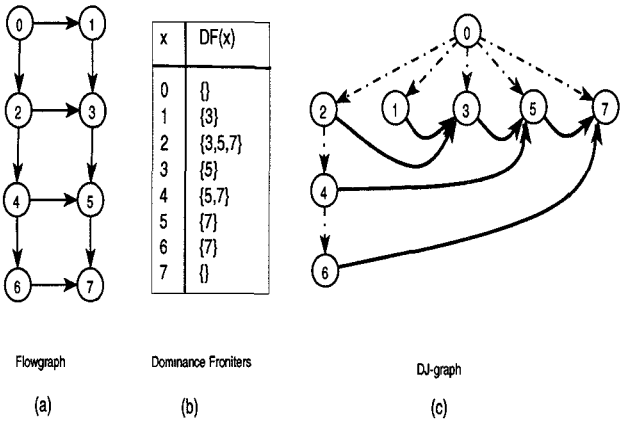

Figure 4: A ladder flowgraph

6 Implementation and Experimental Results

In this section we present our experimental results and their analysis. We implemented our linear time algorithm on top of the Parafrase compiler [Har85b] and executed on 46 FORTRAN routines taken from Perfect, Eispack, Lapack, Ode, Opt, and Gator.[8]. We particular chose those procedures from these suites that are large and have unstructured control flows. We carried out our experiments on a SPARC-10 workstation. To compare the performance of our algorithm with the original algorithm, we also implemented the ϕ-node placement algorithm based on iterating through the dominance frontiers [CFR+91]. This implementation also allowed us to doubly verify the correctness of our algorithm by matching the results of ϕ-nodes of the two algorithms. To be fair, the time measurement shown for the original algorithm does not include the time for pre-computing the dominance frontiers.

For convenience, we will denote $IDF(df)$ for iterated dominance frontier algorithm based on dominance frontiers, and $IDF(new)$ for our new linear time algorithm. Figures 5 and 6 shows the results for some typical procedures taken from Perfect and Lapack programs, respectively. The X-axis gives the names of the procedure we experimented on. In Figure 7 we give the performance for all the 46 FORTRAN procedures we tested. The second column in the table gives the number of flowgraph nodes for the corresponding procedure. The time measurements (averaged over 25 runs) shown for $IDF(df)$ and $IDF(new)$ are for computing ϕ-nodes for a *single* SEG. We randomly chose 15 to 30% of the nodes to be N_α, the initial set of sparse nodes; and we chose the same N_α for both $IDF(new)$ and $IDF(df)$ algorithms.

Originally we implemented the $IDF(df)$ algorithm using bit-vectors for encoding the dominance frontiers for each node. With this implementation our algorithm exhibited a

[8] Eispack, Lapack, Ode, and Opt are available from netlib.att.com. Gator is a Gas, Aerosol, Transport and Radiation model, and is available from ftp.cs.berkeley.edu

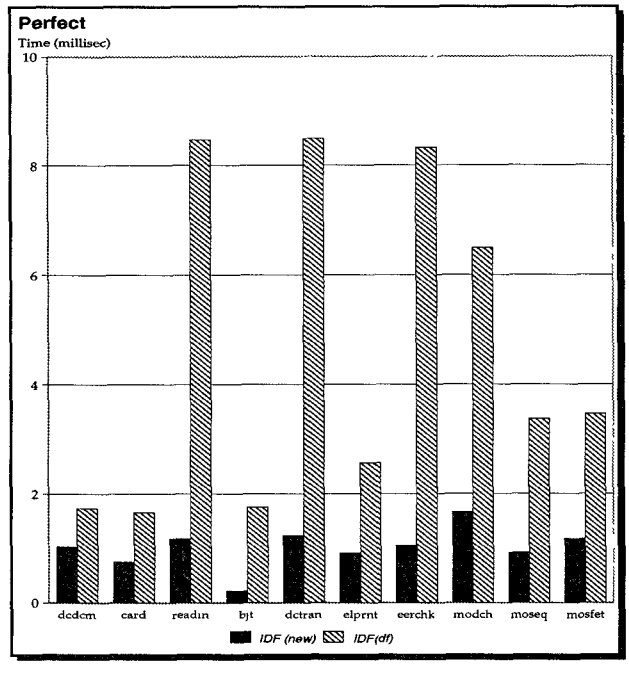

Figure 5: Performance of $IDF(new)$ and $IDF(df)$ algorithms on Perfect

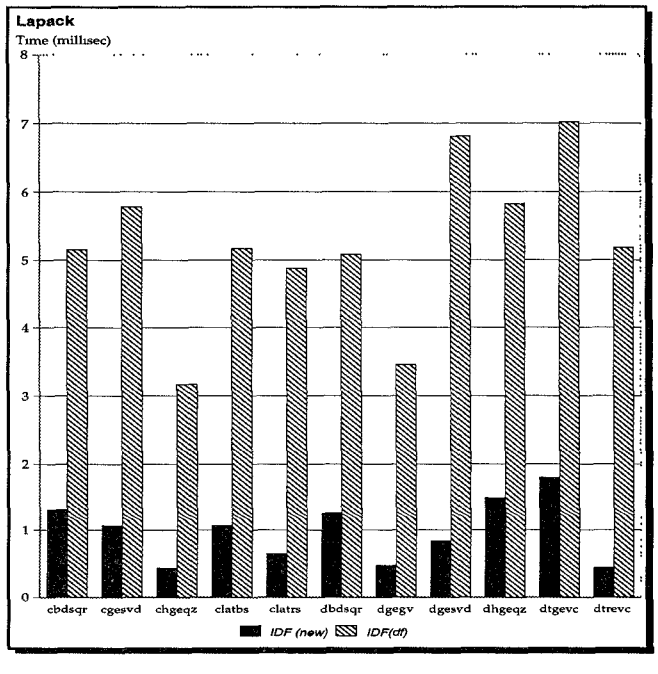

Figure 6: Performance of $IDF(new)$ and $IDF(df)$ algorithms on Lapack

speedup ranging from 4.2 to 19.8 over the $IDF(df)$ algorithm, with average speedup being around 9.0. We then translated the bit-vector representation to a linked-list representation. With this representation our algorithm shows a speedup ranging from 2.2 to 8.9 over the $IDF(df)$ algorithm, with average speedup being around 5.5. This suggest that bit-vectors may not be the best representation for encoding dominance frontiers.

Figure 7 shows the performance of our algorithm over the $IDF(df)$ algorithm (using linked-list structure for representing the dominance frontiers). In particular, the speedup for routines that we tested from the Perfect suite is ranging from 2.2 to 8.0, with average speedup being 5.0. For routines from Lapack speedup range is from 3.9 to 8.1, with the average speedup being 5.7. For smaller programs (less than 75 nodes) we found that both the algorithms take scant time. Clearly, from these plots, we can see that our algorithm performs consistently and significantly faster even for real programs we tested. Our observation here is somewhat different from [CFR+91]: the linear complexity of the algorithm has demonstrated significant benefit in terms of speedup on real programs.

We also measured the execution time for both the algorithms on increasingly taller ladder graphs of the form shown in Figure 4. Recall that for this graph, previous algorithms exhibit non-linear running time because the size of the dominance frontiers of the left-spine increases quadratically as the size of the ladder is increased.

For the ladder graphs, we tested our implementation on a SPARC 20 workstation. We measured the running time of $IDF(df)$ and $IDF(new)$ algorithms as the size of the graph is increased. Figure 8 show the performance curve for both the algorithms on increasingly taller ladder graph. As expected, $IDF(df)$ exhibit quadratic running time, while our new algorithm shows a linear behavior. Notice that the measurement shown for the $IDF(df)$ algorithm is in *seconds*, while for our algorithm it is in milliseconds. This shows that our algorithm is not only linear, but is also significantly faster even for increasingly taller ladder graphs. We observed similar trend for the nested `repeat-until` flowgraph.

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 com-

Proc	$\|N\|$	$IDF(new)^{\dagger}$	$IDF(df)^{\dagger}$	*Speedup*
Perfect				
dcdcmp	138	0.71	1.74	2.4
card	151	0.75	1.66	2.2
readin	407	1.18	8.47	7.2
bjt	136	0.22	1.76	8.0
dctran	310	1.23	8.49	6.9
elprnt	163	0.91	2.56	2.8
errchk	347	1.05	8.32	7.9
modchk	307	1.67	6.49	3.9
moseq2	162	0.92	3.47	3.8
mosfet	215	1.17	3.46	2.8
Eispack				
bandv	126	0.15	0.94	6.3
hqr2	177	0.27	1.67	3.6
invit	189	0.66	2.76	4.1
minfit	120	0.11	1.06	9.6
qzit	125	0.17	1.34	7.9
svd	139	0.49	2.39	4.9
tsturm	151	0.25	2.04	8.2
Lapack				
cbdsqr	238	1.33	5.15	3.9
cgesvd	314	1.07	5.78	5.4
chgeqz	183	0.44	3.17	7.2
clatbs	228	1.07	5.17	4.8
clatrs	218	0.65	4.88	7.5
dbdsqr	238	1.28	5.08	3.9
dgegv	173	0.48	3.46	7.2
dgesvd	324	0.84	6.81	8.1
dhgeqz	295	1.5	5.82	3.8
dtgevc	332	1.8	7.02	3.9
dtrevc	250	0.44	5.18	6.5
Ode				
ddassi	235	1.07	3.50	3.3
svodpk	245	0.85	4.82	5.7
cntrl	228	0.38	3.13	8.2
ddastp	184	1.0	2.60	2.6
newmsh	158	1.14	2.88	2.5
vodpk	245	1.05	4.34	4.1
Opt				
dbocls	169	0.16	1.43	8.9
dbols	129	0.4	1.19	2.9
dbolsm	318	1.81	7.82	4.3
Gator				
aerset	353	1.73	8.73	5.0
aqset	207	0.67	3.79	5.7
chemset	242	1.27	4.71	3.9
equilset	350	2.06	11.37	5.5
initgas	203	1.07	4.86	4.5
jsparse	289	1.36	8.84	5.6
out	405	1.54	9.48	6.2
reader	217	0.93	5.08	5.5
smvgear	224	0.77	4.88	6.3

† in milliseconds

Figure 7: Performance of $IDF(new)$ and $IDF(df)$ on some typical programs.

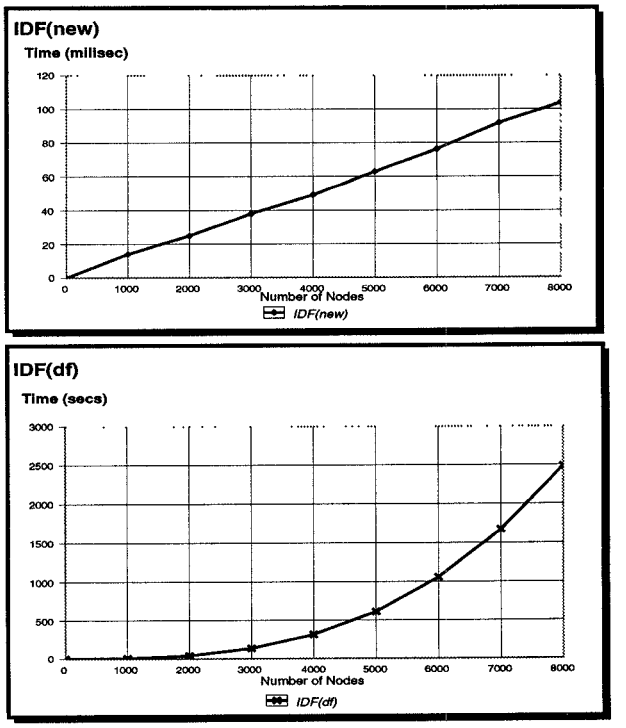

Figure 8: Performance of both the algorithms on ladder graph

plexity 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 that algorithm when applied to real programs. 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 $O(E \times V)$, where V is the number of variables.

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]. Also, their algorithm can not be used for computing the iterated dominance frontiers for a

set of nodes. Iterated dominance frontiers can also used for applications other than for placing ϕ-nodes [SGL94b, Wei92].

Recently Johnson et al. use Quick Propagation Graphs (QPGs) for performing sparse evaluation for arbitrary data flow problem [JPP94]. They give an algorithm for constructing QPGs that runs in linear time. Construction of QPG is based on first constructing regions of control dependences. A disadvantage of QPGs is that it is more denser than SEGs. This means that solving data flow analysis may take more time on QPGs than on SEGs.

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], induction variable recognition [Wol92], 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). 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. The Algorithm 3.1 for computing dominance frontier is similar to the algorithm given by Cytron et al., with one difference, we use level information for capturing the dominance frontiers of a node, while Cytron et al. use CD.START and CD.END relations to do the job.

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.

In a recent work, we have used the algorithmic framework described in this paper to solve the problem of incrementally maintaining dominator trees for an arbitrary flowgraphs [SGL94b]. In the same paper we also propose a method to incrementally update the set of ϕ-nodes of a SEG when the flowgraph is subjected to incremental changes. We believe the framework presented here is robust to accommodate incremental program analysis based on SEGs. We will further explore on this in a future paper.

8 Conclusion

In this paper, we have provided a positive answer to the open problem posted in the introduction: 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.

Our algorithm uses the properties of a new program representation called the DJ-graph which facilities its design and analysis. We also benefit from the simplicity of the algorithm in its implementation. We have constructed a prototype implementation of the algorithm on the top of Parafrase2 compiler. Our experimental results indicate consistent and significant speedup even on real benchmark programs. On increasingly taller ladder graphs our algorithm exhibit linear behavior.

We have been using DJ-graphs and the algorithmic framework presented here to solve a number of other related flowgraph problems. We direct interested readers to our companion papers [SGL94a, SGL94b].

Acknowledgement

We would like to thank many people for their support and encouragement during the course of our work. We owe much thanks to Yong-fong Lee for his insightful technical discussions and also for critically commenting on drafts of the paper. Erik Altman simplified one of the proofs and also suggested many improvements. The comments of Russell Olsen, Bjarne Steengard, Berry Rosen, and the POPL reviewers were very useful and improved the quality of the presentation. Rajiv Gupta and Ron Cytron's constant encouragements and support are greatly appreciated. We thank the National Sciences and Engineering Research Council (NSERC) and the Canadian Centers of Excellence (IRIS) for their continued support of this research. Lastly, the first author would like to dedicate this paper in memory of his late father.

References

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

[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*, pages 257–271, 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 *Languages and Compilers for Parallel Computing*, 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. *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.

[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.

[Har85a] Dov Harel. A linear time algorithm for finding dominators in flow graphs and related problems. In *Symposium on Theory of Computing*. ACM, May 1985.

[Har85b] William Harrison. An overview of the structure of Parafrase. Technical Report 501, PR-85-2 UILU-ENG-85-8002, UIUC, July 1985.

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

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

[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*, pages 12–27, San Diego, California, January 13–15, 1988. ACM SIGACT and SIGPLAN.

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

[SGL94a] Vugranam C. Sreedhar, Guang R. Gao, and Yong-fong Lee. DJ-graphs and their applications to flowgraph analyses. Technical Report ACAPS Memo 70, McGill University, May 1994. Submitted for publication.

[SGL94b] Vugranam C. Sreedhar, Guang R. Gao, and Yong-fong Lee. An efficient incremental algorithm for maintaining dominator trees and its application to ϕ-nodes update. Technical Report ACAPS Memo 77, McGill University, July 1994. Submitted for publication.

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

[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.

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

[Wol92] Michael Wolfe. Beyond induction variables. In *Proceedings of the SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 161–174, 1992.

[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.