1993-01-01

Efficiently Computing $\phi$-Nodes On-The-Fly

Authors: Ron K. Cytron and Jeanne Ferrante

Recently, Static Single Assignment Form and Sparse Evaluation Graphs have been advanced for the efficient solution of program optimization problems. Each method is provided with an initial set of flow graph nodes that inherently affect a problem's solution. Other relevant nodes are those where potentially disparate solutions must combine. Previously, these so-called $\phi$-nodes were found by computing the iterated dominance frontiers of the initial set of nodes, a process that could take worst case quadratic time with respect to the input flow graph. In this paper we present an almost-linear algorithm for determining exactly the same set of $\phi$-nodes.
Efficiently Computing \( \Phi \)-Nodes On-The-Fly

Ron K. Cytron and Jeanne Ferrante

WUCS-93-24

October 1993

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130-4899
Efficiently Computing $\phi$-Nodes On-The-Fly

Ron K. Cytron$^1$ and Jeanne Ferrante$^2$

$^1$ Department of Computer Science  
Washington University Box 1045  
St. Louis, MO 63130-4899  
(Email: cytron@cs.wustl.edu )  
$^2$ T. J. Watson Research Center  
IBM Research PO Box 704  
Yorktown Heights, NY 10598  
(Email: ferrant@watson.ibm.com )

Abstract. Recently, Static Single Assignment Form and Sparse Evaluation Graphs have been advanced for the efficient solution of program optimization problems. Each method is provided with an initial set of flow graph nodes that inherently affect a problem's solution. Other relevant nodes are those where potentially disparate solutions must combine. Previously, these so-called $\phi$-nodes were found by computing the iterated dominance frontiers of the initial set of nodes, a process that could take worst case quadratic time with respect to the input flow graph. In this paper we present an almost-linear algorithm for determining exactly the same set of $\phi$-nodes.

1 Motivation and Background

Static Single Assignment (SSA) form [9] and the more general Sparse Evaluation Graphs (SEG) [6] have emerged as an efficient mechanism for solving compile-time optimization problems via data flow analysis [18, 10, 3, 15, 7, 5]. Given an input data flow framework, the SEG construction algorithm distills the flow graph into a set of relevant nodes. These nodes are then interconnected with edges suitable for evaluating data flow solutions. Similarly, SSA form identifies and appropriately interconnects those variable references that are relevant to solving certain data flow problems.

The SEG and SSA algorithms process and compute the following information:

- **Inputs:**
  - Flowgraph $G_f$ with nodes $N_f$, edges $E_f$, and root node Entry. If $(X, Y) \in E_f$, then we write $X \in Pred(Y)$ and $Y \in Succ(X)$. We assume $G_f$ is connected.
  - Depth-first numbering where $dfn(X)$ is the number associated with $X$
    
    \[ 1 \leq dfn(X) \leq |N_f| \]
in a depth-first search of $G_f$. We call the associated depth-first spanning tree $DFST$. Similarly, we define the inverse mapping

$$\text{vertex}(k) = X | \text{dfn}(X) = k$$

**Dominator tree** where $\text{idom}(X)$ is the immediate dominator of flow graph node $X$. We say that $X$ dominates $Y$, written $X \supseteq Y$, if $X$ appears on every path from flow graph entry to $Y$; domination is both reflexive and transitive. We say that $X$ strictly dominates $Y$, written $X \succ Y$, if $X \supseteq Y$ and $X \neq Y$. Each node $X$ has a unique immediate dominator $\text{idom}(X)$ such that

$$\text{idom}(X) \succ X \text{ and } \forall W \succ X, W \supseteq \text{idom}(X)$$

Node $\text{idom}(X)$ serves as the parent of $X$ in a flow graph's dominator tree. (An example flow graph and its dominator tree are shown in Figure 1.)

**Initial nodes:** $N_\alpha \subseteq N_f$ is an initial subset of those nodes that must appear in the sparse representation. For an SEG, such nodes represent non-identity transference in a data flow framework. For SSA, such nodes contain definitions of variables.

**Outputs:**

Sparse nodes $N_\sigma$, which we compute as a property of each node:

$$T(X) \iff X \in N_\sigma$$

$\phi$-function nodes $N_\phi \subset N_\sigma$, which we compute as a property of each node:

$$\Phi(X) \iff X \in N_\phi$$

Previous SEG and SSA construction algorithms operate as follows:

1. The algorithm precomputes the dominance frontier $DF(X)$ for each node $X$:

$$DF(X) = \{ Z | (\exists Y, Z) \in E_f (X \supseteq Y \text{ and } X \not\supseteq Z) \}$$

In other words, $X$ dominates a predecessor of $Z$ without strictly dominating $Z$. The dominance frontiers for the flow graph in Figure 1 are shown in Figure 2.

2. The algorithm accepts $N_\alpha \subseteq N_f$.

3. The algorithm then computes the set of nodes deserving $\phi$-functions, $N_\phi$, as the iterated dominance frontier of the initial set of nodes:

$$N_\phi = DF^+(N_\alpha)$$

In our example in Figures 1 and 2, if $N_\alpha = \{ D, W \}$, then $N_\phi = \{ W, X, Y, Z \}$.  

4. Steps 2 and 3 can be repeated to create a forest of (related) sparse evaluation graphs. In SSA form, these graphs are usually combined, with the appropriate increase in detail and size with respect to $N_\phi$.

---

3 Here, $\text{dfn}(X)$ is assigned in order of nodes visited, starting with 1; in [2], depth first numbers are assigned starting from $|N_f|$ down to 1.
5. In SEG, appropriate edges are then placed between nodes in

\[ N_\phi = N_\alpha \cup N_\phi \]

In SSA, variables are appropriately renamed, such that each used is reached by a single definition.

These two methods have one component in common, in name as well as function: the determination of \( N_\phi \), where potentially disparate information combines. Consider a flow graph \( G_f \) with \( N \) nodes (set \( N_f \)) and \( E \) edges (set \( E_f \)) for a program with \( V \) variables. While computing the so-called \( \phi \)-nodes is efficient in practice [9],

1. Constructing a single SEG (i.e., one data flow framework) by the usual algorithm [6] takes \( O(E + N^2) \) time;
2. Where a data flow problem can be partitioned into \( V \) disjoint frameworks [14], constructing the associated \( V \) SEGs takes \( O(EV + N^2) \) and \( \Omega(EV) \) time.
3. If we bound the number of variable references per node by some constant, then construction of SSA form takes \( O(EV + N^2) \) and \( \Omega(E + V + N) \) time.

In comparing (2) to (3), note that SEG provides a "solution" for each edge in the flow graph, while SSA form provides a "solution" only at a program's variable references.

Our algorithm for placing \( \phi \)-functions avoids the computation of dominance frontiers. In doing so, we reduce the time bound for (1) to \( O(E \alpha(E)) \), where \( \alpha() \) is the slowly growing inverse Ackermann function [8]. The time bound for (2) is reduced to \( O(V \times E \alpha(E)) \). If our algorithm places \( \phi \)-functions for SSA form, then the time bound for (3) becomes \( O(V \times E \alpha(E)) \) but \( \Omega(EV) \).

To summarize the above discussion, computation of dominance frontiers and their use in placing \( \phi \)-functions can take \( O(E + N^2) \) time [9], although such behavior is neither expected in general nor even possible for programs of certain structure. An example flow graph that exhibits the aforementioned worst case behavior is shown in Figure 1. The flow graph's dominance frontiers are shown in Figure 2. As this graph structure grows\(^4\), the size of dominance frontiers of nodes along its left spine increases quadratically, while the size of the sparse data flow graph or SSA form is certainly linear in size. It is this worst case behavior brought on by precomputing the dominance frontiers that we wish to avoid.

Since one reason for introducing \( \phi \)-functions 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 \( \phi \)-functions without computing or using dominance frontiers.

In this paper we present an algorithm that computes \( N_\phi \) and \( N_\phi \subseteq N_\phi \) from the initially specified \( N_\alpha \). Where previous algorithms begin with a set of nodes \( N_\alpha \) and use dominance frontiers (iteratively) to induct other nodes into \( N_\alpha \), our

\(^4\) by repeating the ladder structure; the back edge is unnecessary and was added to illustrate our algorithm
algorithm does the reverse: we visit nodes in an order that allows us to determine conditions under which a given node must be in \( \mathcal{N}_f \). Using a similar approach, we can then determine \( \mathcal{N}_f \subseteq \mathcal{N}_o \).

In Section 2, we discuss a simple version of our algorithm and illustrate its application to the flow graph in Figure 1. The algorithm's correctness is shown in Section 3. In Section 4, we discuss how balanced path-compression can be used to make the algorithm more efficient; it is these techniques that allow us to achieve our almost-linear time bound. Section 5 gives some preliminary experiments and section 6 suggests future work.

In related work, Johnson and Pingali describe an algorithm to construct an SSA-like representation [13] that takes \( \Theta(EV) \) time. While their upper bound is slightly better than ours, our approach is more general: we construct sparse evaluation graphs for arbitrary data flow problems, while Johnson and Pingali construct def-use structures specific to the solution of SSA-based optimization problems such as constant propagation.

As we discuss further in Section 5, \( O() \) asymptotic bounds in this area are deceptive, and one must take into account lower and expected bounds. The usual dominance-frontier-based algorithm [9] is biased toward the average case, in which linear behavior for constructing or consulting dominance frontiers is expected. The algorithm we present in this paper, and the algorithm due to Johnson and Pingali [13], are \( \Omega(EV) \), since each edge in the flow graph is examined for each variable.

We actually present two variations of the same algorithm in this paper. The first "simple" algorithm is presented for expository reasons; the second algorithm uses balanced path-compression to achieve our improved time bound. Preliminary experiments presented in Section 5 compare the simple algorithm presented in Section 2 with the usual dominance frontier-based algorithm [9]. In fact, the usual algorithm is often faster, and so these experiments do not suggest blindly switching to the asymptotically faster algorithm. However, our algorithm does exhibit the same linear behavior as the usual algorithm. Moreover, we have not implemented the balanced path-compression presented in Section 4 which yields our better bound. These experiments give some evidence that our algorithm can yield comparable performance to the usual algorithm, while avoiding asymptotically poor efficiency.

## 2 Algorithm

**Definition 1.** The equidominates of a node \( X \) are those nodes with the same immediate dominator as \( X \):

\[
equidom(X) = \{ Y \mid idom(Y) = idom(X) \}
\]

For example, in Figure 1,

\[
equidom(A) = \{ A, V, W, X, Y, Z \}
\]

More generally, the noun *equidominates* refers to any such set of nodes.
Our algorithm essentially partitions equidominates into blocks of nodes that are in each other's iterated dominance frontiers, but without the expense of explicitly computing dominance frontiers. The relevant portions of our algorithm refer to $\text{Cnum}(X)$, $\text{LL}(X)$, or $\text{Map}(X)$; these are paraphrases of a strongly-connected component (SCC) algorithm [1], which we use to find (some) strongly-connected components of the dominance frontier graph (in which an edge from $Y$ to $Z$ implies $Z \in DF(Y)$). While the details of $\text{Cnum}(X)$ and $\text{LL}(X)$ are beyond the scope of this paper [1], we note that $\text{Map}(X)$ is the representative node for
Algorithm 1 Sparse graph node determination (simple)

\texttt{NodeCount} \leftarrow 0
\textbf{foreach} (\textbf{X} \in N_f) \textbf{do}
\hspace{1em} \texttt{T}(\textbf{X}) \leftarrow \texttt{false}
\hspace{1em} \Phi(\textbf{X}) \leftarrow \texttt{false}
\hspace{1em} \texttt{Map}(\textbf{X}) \leftarrow \textbf{X}
\textbf{od}
\texttt{Cnum} \leftarrow 0
\textbf{for} \texttt{n} = N \textbf{downto} 1 \textbf{do}
\hspace{1em} \textbf{foreach} \{ \texttt{Z} \mid \texttt{dfn}(\texttt{idom}(\texttt{Z})) = \texttt{n} \}\textbf{do} \hspace{1em} \textit{\textbf{1}}
\hspace{2em} \textbf{if} (\texttt{Cnum}(\texttt{Z}) = 0) \textbf{then} \hspace{1em} \textit{\textbf{2}}
\hspace{3em} \textbf{call} \texttt{Visit}(\texttt{Z})
\hspace{2em} \textbf{fi}
\hspace{1em} \textbf{od}
\textbf{od}
\textbf{for} \texttt{n} = N \textbf{downto} 1 \textbf{do}
\hspace{1em} \textbf{foreach} \{ \texttt{Z} \mid \texttt{dfn}(\texttt{idom}(\texttt{Z})) = \texttt{n} \}\textbf{do} \hspace{1em} \textit{\textbf{3}}
\hspace{2em} \textbf{call} \texttt{Finish}(\texttt{Z})
\hspace{1em} \textbf{od}
\textbf{od}
\textbf{end}

\textbf{Function} \texttt{FindNode}(Y, P) : node
\textbf{for} \texttt{X} = Y \textbf{repeat} \texttt{X} = \texttt{idom}(\texttt{X}) \textbf{do}
\hspace{1em} \textbf{if} (\texttt{T}(\texttt{Map}(\texttt{X})) \textbf{or} (\texttt{idom}(\texttt{X}) = P)) \textbf{then} \hspace{1em} \textit{\textbf{4}}
\hspace{2em} \textbf{return} \texttt{Map}(\texttt{X})
\hspace{2em} \textbf{fi}
\hspace{1em} \textbf{od}
\textbf{end}

\textbf{Fig. 3.}

the SCC containing node \texttt{X}. Throughout this paper, “SCC” or “component” refers to nodes associated in this manner.

Central to this algorithm is the function \texttt{FindNode}(Y, P), which ascends the dominator tree from node \texttt{Y}, searching for a sparse graph node below \texttt{P}. Our proof of algorithmic correctness (Section 3) relies on the nature of \texttt{FindNode}() rather than its actual implementation: the function always returns

\[
\texttt{X} \mid \texttt{Z} \in \texttt{DF}^+(\texttt{X}), \texttt{Z} \in \texttt{Succ}(\texttt{Y}), \texttt{P} \Rightarrow \texttt{X} \gg \texttt{Y}
\]

We present a straightforward, albeit inefficient, version of \texttt{FindNode}(Y, P) in Figure 3. The correctness of our algorithm as presented in Section 3 is based
Procedure \textit{Visit}(Z)\\
\text{\scriptsize LL}(Z) \leftarrow \text{Cnum}(Z) \leftarrow + + \text{NodeCount}\\
call \text{push}(Z)\\
\textbf{foreach} ((Y, Z) \in E \mid Y \neq \text{idom}(Z)) \textbf{do}\\
\quad s \leftarrow \text{FindSnode}(Y, \text{idom}(Z))\\
\quad \text{if} (\text{idom}(s) \neq \text{idom}(Z)) \textbf{then}\\
\quad \quad \text{call} \text{IncludeNode}(Z)\\
\quad \text{else}\\
\quad \quad \text{if} (\text{Cnum}(s) = 0) \textbf{then}\\
\quad \quad \quad \text{call Visit}(s)\\
\quad \quad \quad \text{\text{LL}(Z) \leftarrow \text{min}(\text{LL}(Z), \text{LL}(s))}\\
\quad \quad \text{else}\\
\quad \quad \quad \text{if} ((\text{Cnum}(s) < \text{Cnum}(Z)) \text{ and } \text{OnStack}(s)) \textbf{then}\\
\quad \quad \quad \quad \text{\text{LL}(Z) \leftarrow \text{min}(\text{LL}(Z), \text{Cnum}(s))}\\
\quad \quad \quad \text{fi}\\
\quad \quad \quad \text{fi}\\
\quad \quad \text{if} (T(s) \text{ and not } \text{OnStack}(s)) \textbf{then}\\
\quad \quad \quad \text{call IncludeNode}(Z)\\
\quad \text{fi}\\
\textbf{od}\\
\text{if} (\text{LL}(Z) = \text{Cnum}(Z)) \textbf{then}\\
\text{repeat}\\
\quad Q \leftarrow \text{pop()}\\
\quad \text{Map}(Q) \leftarrow Z\\
\quad \text{if} (Q \in \mathcal{N}_a) \textbf{then}\\
\quad \quad \text{call IncludeNode}(Q)\\
\quad \text{fi}\\
\quad \text{if} (T(Q)) \textbf{then}\\
\quad \quad \text{call IncludeNode}(Z)\\
\quad \text{fi}\\
\text{until} (Q = Z)\\
\text{end}\\
Fig. 4.

on the behavior of \textit{FindSnode}(\text{\scriptsize )}. To obtain our almost-linear time bound, we modify our algorithm as discussed in Section 4.

The algorithm is shown in Figures 3, 4, and 5. We now illustrate the application of the algorithm to the flow graph in Figure 1, assuming that \( \mathcal{N}_a = \{ B \} \). Although the full proofs appear in Section 3, we mention here that:

\[ Z \in \text{DF}(X) \Rightarrow \text{dfn(idom}(X)) \geq \text{dfn(idom}(Z)) \]

By ensuring that nodes with higher depth first number have already been corre-
Procedure \textit{IncludeNode}(X) \\
\hspace{1em} \mathcal{T}(X) \leftarrow true \\
end \\

Procedure \textit{finish}(Z) \\
\hspace{1em} \textbf{foreach} ((Y, Z) \in \mathcal{E}_I \mid Y \neq idom(Z)) \textbf{do} \\
\hspace{2em} s \leftarrow \textit{FindNode}(Y, idom(Z)) \\
\hspace{2em} \textbf{if} (\mathcal{T}(s)) \textbf{then} \\
\hspace{3em} \mathcal{T}(Z) \leftarrow true \\
\hspace{2em} \textbf{fi} \\
\hspace{1em} \textbf{od} \\
\hspace{1em} \textbf{if} (\mathcal{T}(\textit{Map}(Z))) \textbf{then} \\
\hspace{2em} \textit{call IncludeNode}(Z) \\
\hspace{1em} \textbf{fi} \\
end \\

Fig. 5.

directly determined to be in \(\mathcal{N}_\sigma\), the algorithm can correctly determine this property for nodes of lower depth first number.

The loop at 1 begins with \(V\) (depth-first numbered 10 in Figure 1), but since no nodes are dominated by \(V\), no steps are taken by 2; the same holds for nodes \(W\) and \(D\). When 1 considers \(C\), loop 2 calls \textit{Visit} with \(D\). The loop in \textit{Visit} at 7 is empty, since the only predecessor of \(D\) immediately dominates \(D\). Thus, \textit{Map}(D) is set to \(D\) in loop 13. Loop 1 then considers in turn \(Z\), \(Y\), and \(X\), each of which dominates no node, so 2 is empty. When 1 considers \(B\), \textit{Visit} is called on \(C\), but since \(C\)'s only predecessor immediately dominates \(C\), no action is taken by 7, and \textit{Map}(C) is set to \(C\). When \textit{Visit} is next called on \(B\), \textit{Map}(B) is set to \(B\); also, since \(B \in \mathcal{N}_\alpha\), step 15 places \(B\) in \(\mathcal{N}_\sigma\).

When 1 considers \(E\), suppose loop 2 considers the nodes immediately dominated by \(E\) in order \(A\), \(Y\), \(W\), \(X\), \(V\), and \(Z\) (although some of these will already have been processed by recursive calls to \textit{Visit}). When \textit{Visit} works on \(A\), no steps of 7 are taken, and \textit{Map}(A) becomes \(A\). When \textit{Visit} works on \(Y\), \textit{FindNode} will be called on \(C\) and \(X\).

- For \(C\), \textit{FindNode} returns \(B\); since \(B\) and \(Y\) are not equidominates, \(Y\) is placed in \(\mathcal{N}_\sigma\) by 9.
- For \(X\), \textit{FindNode} returns \(X\); since \(X\) and \(Y\) are equidominates, \textit{Visit} is called recursively on \(X\). In this invocation, loop 7 considers nodes \(B\) and \(W\).
  * For \(B\), \textit{FindNode} returns \(B\), so \(X\) is placed in \(\mathcal{N}_\sigma\).
  * For \(W\), \textit{FindNode} returns \(W\), so \textit{Visit} is called recursively on \(W\). In this invocation, loop 7 considers nodes \(A\), \(V\), and \(X\).
    * For \(A\), \textit{FindNode} returns \(A\). Since that node has already been visited, and since \(A\) is not in \(\mathcal{N}_\sigma\), nothing happens to \(W\) because of
A.

* For $V$, FindSnode returns $V$; Visit is then called recursively for $V$, where $Map(V)$ becomes $V$.
* For $X$, FindSnode returns $X$, which is already on stack, so nothing happens. In particular, $Map(X)$ is not set.

Now, $Map(W)$ and $Map(X)$ are both set to $W$. Since $X \in \mathcal{N}_\sigma$, $W$ is placed in $\mathcal{N}_\sigma$.

When loop $[2]$ considers $W$, $X$, and $V$, each has already been visited. When Visit is called for $Z$, nodes $D$ and $Y$ are considered by $[7]$.

- For $D$, FindSnode returns $B$, so $Z$ is placed in $\mathcal{N}_\sigma$.
- For $Y$, FindSnode returns $Y$, which has already been visited.

In contrast, previous methods $[9, 6]$ would not only have constructed all dominance frontier sets in Figure 2, which are asymptotically quadratic in size, but would also have iterated through the dominance frontier sets of all nodes put on the worklist, namely \{ $E, B, V, W, X, Y, Z$ \}.

3 Correctness

We begin with some preliminary lemmas.

Lemma 2.

$$(Y, Z) \in E_f \implies \text{idom}(Z) \gg Y$$

Proof: Suppose not. Then there exists a path

$$P : \text{Root} \uparrow Y$$

that does not contain idom($Z$). If path $P$ is extended by an edge $(Y, Z)$, then we obtain a path to $Z$ that doesn’t contain idom($Z$). Thus, $(Y, Z) \notin E_f$.

\[\square\]

Corollary 3.

$$(Y, Z) \in E_f \text{ and } Y \neq \text{idom}(Z) \implies\text{idom}(Z) \leq \text{dfn}(\text{idom}(Z)) < \text{dfn}(Y)$$

Corollary 4.

$$(Y, Z) \in E_f \text{ and } Y \neq \text{idom}(Z) \implies\text{idom}(Z) \gg \text{idom}(Y) \gg Y$$

Lemma 5. If $X$ is an ancestor of $\text{idom}(Z)$ in the depth-first spanning tree $\text{DFST}$ of $G_f$, and $\text{idom}(Z) \gg Y$, then

$$X \gg Y \implies X \gg Z$$
Proof: Suppose not. Then there exists a path
\[ \text{Root} \rightarrow \text{idom}(Z) \rightarrow Z \]
that excludes \( X \). Since \( \text{idom}(Z) \not\supseteq Y \), there is a path of DFST edges from \( \text{idom}(Z) \) to \( Y \) that excludes \( X \). Thus, \( X \) cannot dominate \( Y \).
\( \square \)

Theorem 6. If \( X \) is an ancestor of \( \text{idom}(Z) \) in the depth-first spanning tree DFST of \( G_f \), then \( Z \) cannot be in the dominance frontier of \( X \).

Proof: Suppose \( Z \in \text{DF}(X) \). Then \( \exists (Y, Z) \in E_f \) such that \( X \not\supseteq Y \) and \( X \not\succeq Z \). It cannot be the case that \( \text{idom}(Z) = Y \), since if so \( X \not\supseteq Y = \text{idom}(Z) \not\supseteq Z \), a contradiction. Therefore, \( \text{idom}(Z) \not\supseteq Y \). By Corollary 4, \( \text{idom}(Z) \not\supseteq \text{idom}(Y) \not\supseteq Y \). By Lemma 5, \( X \not\supseteq Y \rightarrow X \not\succeq Z \). But since \( X \not\supseteq Y \), we have \( X \not\succeq Z \), a contradiction. Hence \( Z \notin \text{DF}(X) \).
\( \square \)

Theorem 7.

\[ Z \in \text{DF}(X) \implies \text{dfn}(X) > \text{dfn}(\text{idom}(Z)) \]

Proof: Suppose \( \text{dfn}(X) \leq \text{dfn}(\text{idom}(Z)) \). In the DFST of \( G_f \), either
1. \( X \) is an ancestor of \( \text{idom}(Z) \). By Theorem 6,
\[ Z \notin \text{DF}(X) \]
or
2. \( X \) is to the "left of" \( \text{idom}(Z) \). Suppose there existed some node \( Y \in \text{Preds}(Z) \) dominated by \( X \). Since \( Y \) must be a descendant of \( X \) in DFST, \( Y \) is also to the left of \( \text{idom}(Z) \), and therefore \( Y \) is to the left of \( Z \). But \( Z \) cannot have a predecessor to its left in DFST. Therefore \( Z \notin \text{DF}(X) \).
\( \square \)

Theorem 8.

\[ Z \in \text{DF}(X) \implies \text{idom}(Z) \not\supseteq X \]

Proof: Suppose not. Then either
1. \( \text{idom}(Z) = X \), which implies \( Z \notin \text{DF}(X) \); or,
2. \( \text{idom}(Z) \not\equiv X \) and \( \text{idom}(Z) \not\supseteq X \). Consider then the path
\[ P : \text{Root} \rightarrow X \]
that doesn't contain \( \text{idom}(Z) \). If \( Z \in \text{DF}(X) \), then we can extend path \( P \) to
\[ Q : \text{Root} \rightarrow X \rightarrow Y \rightarrow Z \]
such that \( X \not\supseteq Y \) and \( X \not\succeq Z \). With \( X \not\supseteq Y \), we can construct \( Q \) such that edges between \( X \) and \( Y \) are DFST edges. Since \( \text{idom}(Z) \) doesn't occur on path \( P \), \( \text{idom}(Z) \) must occur on path \( Q \) after node \( X \) and before node \( Z \). Thus, \( X \) is a DFST ancestor of \( \text{idom}(Z) \). By Theorem 6, \( Z \notin \text{DF}(X) \).
Corollary 9.

\[ Z \in DF(X) \implies dfn(idom(X)) \geq dfn(idom(Z)) \]

The following lemmas which formalize those properties of our algorithm that participate in our correctness proof. We omit proofs that directly follow from inspection of our algorithm.

Lemma 10. Visit is called exactly once for each node in Flowgraph.

Lemma 11. During all calls to Visit, each node is pushed and popped exactly once.

Proof: By Lemma 10, Visit(Z) is invoked exactly once, and on this call Z is pushed exactly once. We need to show that Z is popped exactly once. If \( Z = Map(Z) \), then Z is popped by the invocation of Visit in which Z was pushed. Otherwise, Z belongs to a strongly-connected component represented by \( H, H \neq Z \), in which case Z is popped by the iteration in which H is pushed. □

Lemma 12.

\[ Y = Map(X) \implies idom(Y) = idom(X) \]

Proof: Since initially \( X = Map(X) \), the lemma holds at the start of the algorithm. Otherwise, Map(X) is only set during the loop at step 13, when squidominates are popped off the stack. In this case, we have idom(Map(X)) = idom(X).

□

Lemma 13. At step 12, node s (referenced at step 11) has already been pushed and popped.

Proof: By the predicate of step 11, node s cannot be on stack at step 12. Thus, s either hasn’t been pushed yet, or else s has been pushed and popped. If s hasn’t been pushed then \( Cnum(s) = 0 \), but then step 10 would have pushed s before step 12 is reached. Therefore, s has been pushed and popped. □

Corollary 14. Any invocation of IncludeNode(s) must already have occurred at step 12.

Lemma 15.

\[ OnStack(X) \text{ and } OnStack(Y) \implies idom(X) = idom(Y) \]

Lemma 16. At step 8, FindSnode() returns

\[ s \mid s = Map(S), Z \in DF(S) \]
Proof: Follows from inspection of \textit{FindNode()} and the definition of dominance frontiers. □

Lemma 17.

$$\map(X) = \map(Y) \implies Y \in DF^+(X) \text{ or } X = Y$$

Proof: Follows from initialization ($\map(X) = X$), Lemma 16, and the observation that $\map(X)$ represents the strongly-connected component containing $X$. □

Corollary 18.

$$s \in N_o \iff \map(s) \in N_o$$

Theorem 19. As of step 3,

$$\tau(\map(X)) \iff \map(X) \in N_o$$

Proof:

⇒ We actually prove a stronger result:

$$\tau(X) \implies X \in N_o$$

To accommodate the iteration of step 1, we prove the induction hypothesis

$$IH_A(n) \equiv (dfn(idom(X)) = n) \land \tau(X) \implies X \in N_o$$

by backward induction on $n$ (following the progression of our algorithm), noting that $\tau(X)$ can only be set in procedure $Visit(Z)$ when $idom(Z) = idom(X)$.

Base case: Node vertex$(N)$ is childless in its depth-first spanning tree, and so cannot immediately dominate any node. With loop at step 2 empty, this case is trivially satisfied.

Inductive step: Consider those steps that potentially set

$$\tau(X), dfn(idom(X)) = n$$

Step 3: With $idom(s) \neq idom(Z)$, step 3 must have returned

$$s \mid \tau(s), s = \map(S), Z \in DF(S)$$

From Lemma 16 we obtain

$$Z \in DF^+(s)$$

From Lemma 12 and with $idom(s) \neq idom(Z)$, Corollary 9 implies

$$dfn(idom(s)) > dfn(idom(Z))$$

Applying $IH_A(k) \mid N \geq k \geq n + 1$,

$$\tau(s) \implies s \in N_o$$

Thus,

$$Z \in N_o$$
Steps [12], [15], and [16]: Each of these steps determines \( \mathcal{T}(X) \) by consulting nodes whose immediate dominator is \( idom(X) \). Each node in the set
\[
\{ X \mid dfn(idom(X)) = n \}
\]
gests pushed and popped exactly once (by steps [6] and [14]). We name such nodes
\[
\{ x_1, x_2, \ldots, x_L \}
\]
according to the order in which they emerge from the stack: node \( x_1 \) is the first such node popped, node \( x_i \) is popped before node \( x_{i+1} \), and node \( x_L \) is the last such node popped. Accordingly, we define the predicate
\[
Popped(k) \equiv \bigwedge_{i=k+1}^{L} Onstack(x_i)
\]
which is true when exactly \( k \) such nodes have been popped. We now prove the following induction hypothesis:
\[
IH_B(n) \equiv (Popped(n)) \land \mathcal{T}(X) \implies X \in \mathcal{N}_o
\]

Base case: Prior to popping \( x_1 \), Lemma 15 ensures that steps [15] and [16] cannot affect any of the \( x_i \). By Lemma 13, step [12] requires \( s \) to be an already popped \( x_i \), so step [12] cannot affect any of the \( x_i \).

Inductive step: We now prove \( IH_B(n) \).

Step [12]: By Lemma 13, \( s \) has already been pushed and popped. By Corollary 14, and assuming \( IH_B(k) \mid 1 \leq k \leq n - 1 \), we obtain
\[
s \in \mathcal{N}_o, \quad Z \in DF^+(s)
\]
Thus,
\[
Z \in \mathcal{N}_o
\]

Step [15] By definition,
\[
Q \in \mathcal{N}_o \implies Q \in \mathcal{N}_o
\]

Step [16] Two cases:
\( Q \neq Z \): This statement cannot affect \( \mathcal{T}(Q) \), whose membership in \( \mathcal{N}_o \) is then covered by the other cases in this proof.
From Lemma 17, \( Z \in DF^+(Q) \), and so \( Z \in \mathcal{N}_o \).
\( Q = Z \): This statement becomes tautologous.
To prove 
\[ \text{Map}(Y) \in \mathcal{N}_\alpha \implies T(\text{Map}(Y)) \]

it is sufficient to show 
\[ Y \in \text{Map}(\mathcal{N}_\alpha) \implies T(Y) \]

We formalize iterative dominance frontiers by:
\[
\begin{align*}
\mathcal{DF}^0(\mathcal{N}_\alpha) &= \text{Map}(\mathcal{N}_\alpha) \\
\mathcal{DF}^i(\mathcal{N}_\alpha) &= \text{Map}(DF(\mathcal{DF}^{i-1}(\mathcal{N}_\alpha)))
\end{align*}
\]

so that
\[
\bigcup_{i=0}^{\infty} \mathcal{DF}^i(\mathcal{N}_\alpha) = \text{Map}(\mathcal{N}_\alpha)
\]

We now prove the following induction hypothesis:
\[ IH_C(n) \equiv Y \in \mathcal{DF}^n(\mathcal{N}_\alpha) \implies T(Y) \]

**Base case:** Every node in the flow graph is pushed and popped by steps [6] and [14]. At step [15],
\[ y \in \mathcal{N}_\alpha \implies T(y) \]

At step [16],
\[ Y \in \text{Map}(\mathcal{N}_\alpha) \implies T(Y) \]

where \( Y = \text{Map}(y) \).

**Inductive step:** We now prove \( IH_C(n) \) assuming \( IH_C(n - 1) \). Consider any
\[ Y \in \mathcal{DF}^n(\mathcal{N}_\alpha), n > 0 \]

Since
\[ Y \in \text{Map}(DF(\mathcal{DF}^{n-1}(\mathcal{N}_\alpha)(\mathcal{N}_\alpha))) \]

we have
\[ Y = \text{Map}(y) \mid y \in DF(X), X \in \mathcal{DF}^{n-1}(\mathcal{N}_\alpha) \]

By Lemma 12
\[ \text{idom}(Y) = \text{idom}(y) \]

and so Corollary 9 can be extended to:
\[ y \in DF(X) \implies \text{dfn}(\text{dom}(X)) \geq \text{dfn}(\text{dom}(Y)) \]

We show that \( \text{Visit}(y) \) will set \( T(Y) \) true. By \( IH_C(n - 1) \), we have \( T(X) \) true. Since \( X \in \mathcal{DF}^{n-1}(\mathcal{N}_\alpha) \), the call to \( \text{FindSnod}() \) at step [8] will return \( X \). There are two cases:
- \( \text{dfn}(\text{dom}(Y)) < \text{dfn}(\text{dom}(X)) \): Step [9] sets \( T(y) \) true. When \( y \) is popped at step [14], step [16] sets \( T(Y) \) true.
- \( \text{dfn}(\text{dom}(Y)) = \text{dfn}(\text{dom}(X)) \): Two cases:
\( X \neq Y \): Step \([12]\) sets \( T(y) \) true. When \( y \) is popped at step \([14]\), step \([16]\) sets \( T(Y) \) true.
\( X = Y \): then \( T(X) \implies T(Y) \).
\( \square \)

**Theorem 20.** After the call to \texttt{finish},
\[
\Phi(Z) \iff Z \in \mathcal{N}_\phi
\]

**Proof:**
\[
\Phi(Z) \iff \exists (Y, Z) \in E_f \mid \quad \\
Y \neq \text{idom}(Z), \\
s = \text{FindSnod}(Y, \text{idom}(Z)), \\
s \in \mathcal{N}_\sigma
\]
by construction in \texttt{finish()} and Theorem 19. But this holds
\[
\iff Z \in D\phi^+(s), s \in \mathcal{N}_\sigma
\]
by construction in \texttt{FindSnod()}, and this holds
\[
\iff Z \in \mathcal{N}_\phi
\]
by definition of \( \mathcal{N}_\phi \). \( \square \)

**Theorem 21.** After calls to \texttt{finish()},
\[
T(X) \iff X \in \mathcal{N}_\sigma
\]

**Proof:**
\[
X \in \mathcal{N}_\sigma \iff \text{Map}(x) \in \mathcal{N}_\sigma
\]
(by Corollary 18). But
\[
\text{Map}(x) \in \mathcal{N}_\sigma \iff T(\text{Map}(X))
\]
(by Theorem 19). But in \texttt{finish},
\[
T(\text{Map}(X)) \iff T(X)
\]
\( \square \)

4 **Complexity**

In this section, we first show that our algorithm is \( O(N + E + T) \), where \( N \) is the number of nodes and \( E \) the number of edges in the input flowgraph, and \( T \) is the total time for all calls to \texttt{FindSnod}. Unfortunately, \( T \) is not linear using \texttt{FindSnod} as written. We then provide a faster version of our algorithm and show that our correctness results still hold. In our faster algorithm, \( T \) is \( O(E\alpha(E)) \), obtaining our desired almost-linear complexity bound.
4.1 Analysis of Initial Algorithm

Theorem 22. The algorithm of Figure 3 is $O(N + E + T)$, where $N$ is the number of nodes and $E$ the number of edges in the input flowgraph, and $T$ is the total time for all calls to FindNode.

Proof: The algorithm consists of

- an initialization phase,
- a phase where Visit is called recursively once for each node, and
- a call to finish.

We analyze the complexity of each of these phases. The initialization phase is $O(N)$. For each call Visit(Z), there is a constant amount of work not inside any loop, the loop starting at step 7 over predecessor edges into Z, and the loop starting at step 13 where the contents of the stack are popped. The constant amount of work can be ignored in determining the bound. Consider the loop starting at step 7. Since Visit is called only once for each node, over all calls to Visit, this loop is executed $O(E)$ times. Thus over all calls to Visit, the loop will execute at most $O(E)$ calls to FindNode in step 5, and $O(E)$ other work. For the last loop starting at step 13 in Visit, all nodes are pushed and popped exactly once, so this loop is $O(N)$. Finally, consider the call to finish. It consists of $O(E)$ work plus at most $O(E)$ calls to FindNode. Summing all of this work, we obtain the desired result. □

We now analyze the asymptotic behavior of the function $\text{FindNode}(Y, P)$ as shown in Figure 3. Each invocation could require visiting each node on a dominator tree path from Entry to $Y$. The cost of applying $\text{FindNode}(\cdot)$ to each of $N$ flow graph nodes is then $O(N^2)$. As such, the overall asymptotic behavior of the simple version of our algorithm is $O(E \times N)$.

4.2 Faster Algorithm

Using a path-compression result due to Tarjan [17], we rewrite certain parts of our algorithm (shown in Figures 6 and 7) to use the instructions:

- $Eval(Y)$: Using the links established by the $\text{Link}()$ instruction, $Eval()$ ascends the dominator tree from $Y$, returning the node of maximum label. The label initially associated with each node $X$ is $-\text{dfn}(X)$; the link of each node is initially $1$.
- $Link(Y, \text{idom}(Y))$: sets $\text{link}(Y) = \text{idom}(Y)$. Any $Eval()$ search that includes node $Y$ now also includes the immediate dominator of $Y$.
- $Update(X, \text{dfn}(X))$: changes the label associated with $X$ to $\text{dfn}(X)$. This instruction is issued when node $X$ becomes included in set $N_F$.

The path-compressing version of algorithm is obtained as follows:

1. We initialize the path-compression at steps 20 and 24.
Procedure Initialize

NodeCount ← 0
foreach (X ∈ N_f) do
    T(X) ← false
    φ(X) ← false
    Map(X) ← X
od
foreach (X ∈ N_f) do
    link(X) ← ⊥
    Label(X) ← dfn(X)
od
Cnum ← 0
end

Procedure IncludeNode(X)

γ(X) ← true
    call Update(X, dfn(X))
end

Fig. 6.

2. Links are inserted to extend the Eval() search at steps 22 and 25.
3. Path information is updated whenever a node X is added to the sparse graph, at step 21.
4. We redefine function FindNode() by:

   Function FindNode(Y, P) : node
       return (Map(Eval(Y)))
   end

We must now show:

1. We use the above instructions in a manner consistent with their definition [17]: operations Link() and Update() are applied to nodes at the end of a “link-path”.

Lemma 23. At steps 23 and 26, link(Z) = ⊥ prior to invoking Link().

Proof: Steps 20 and 24 initialize link(X) = ⊥ for each node X ∈ N_f. Since each node Z has a unique immediate dominator in idom(Z), and a unique map representative in Map(Z), and since n is a strictly decreasing sequence at steps 22 and 25, link(Z) = ⊥ just prior to applying Link() at steps 23 and 26. □

Lemma 24. When Update(X, dfn(X)) is invoked at step 21, link(X) = ⊥.
call Initialize
for $n = N$ downto 1 do
  foreach ($\{ Z \mid dfn(idom(Z)) = n \}$) do
    if $(Cnum(Z) = 0)$ then
      call Visit(Z)
    fl
  od
  foreach ($\{ Z \mid dfn(idom(Z)) = n \}$) do
    if $(Z = Map(Z))$ then
      call Link(Z, idom(Z))
    else
      call Link(Z, Map(Z))
    fi
  od
od
foreach ($X \in N_f$) do
  link(X) ← ⊥
  Label(X) ← − dfn(X)
od
for $n = N$ downto 1 do
  foreach ($\{ Z \mid dfn(idom(Z)) = n \}$) do
    call Finish(Z)
  od
  foreach ($\{ Z \mid dfn(idom(Z)) = n \}$) do
    if $(Z = Map(Z))$ then
      call Link(Z, idom(Z))
    else
      call Link(Z, Map(Z))
    fi
  od
od

Fig. 7.

Proof: The procedure IncludeNode() is invoked only from Visit(), which processes only equidominates. Since step $[22]$, has not yet executed for any node considered by Visit(), each such node has ⊥ for its link. 

2. Use of these instructions does not change the output of our algorithm.

Lemma 25. As invoked during the course of their respective algorithms, each implementation of FindNode(Y, P) returns the same answers.

Proof:
By inspection, $\text{FindNode}(Y, P)$ of Figure 3 begins at node $Y$ and considers each ancestor $X$ of $Y$, up to but excluding node $P$. As each node $X$ is considered, the function returns $\text{Map}(X)$ if $\text{Map}(X)$ is already included in the sparse graph. If no $\text{Map}(X)$ is already in the sparse graph, then the function returns $\text{Map}(X)$, where $X$ is ancestor of $Y$ just prior to $P$.

We now argue that $\text{FindNode}(Y, P)$ at \cite{27} simulates exactly this behavior. First, notice that the path of links established at steps \cite{23} and \cite{26} link each node $X$ to $\text{Map}(X)$ if $X \neq \text{Map}(X)$, and otherwise link each node $X$ to $\text{idom}(X)$. Thus, strongly connected nodes are linked to their representative member, while that member is linked to its dominator. Each node $X$'s label begins as $-\text{dfn}(X)$, but can be changed by step \cite{21} to be $\text{dfn}(X)$. There are two cases to consider:

(a) If $\text{Map}(X)$ is in the sparse graph, for any node $X$ between $Y$ and $P$ (excluding $P$), then there is a link path to that node and its label has been changed to $\text{dfn}(\text{Map}(X))$. $\text{Eval}(Y)$ returns the node of maximum label on the link path from $Y$, up to but excluding node $P$, since $P$ has not been linked in yet. Since any node is depth-first numbered higher than its immediate dominator, $\text{Eval}(Y)$ returns some node in the strongly-connected component closest to $Y$ whose representative node is already in the sparse graph. Applying $\text{Map}()$ once again ensures that the representative node is returned.

(b) If no node on the link path is included in the sparse graph, then each such node must be labeled by its negative depth-first number. Thus, when $\text{Eval}(Y)$ returns the node of maximum label, this will be the node of minimum negative label, which will be some node in the strongly-connected component containing the ancestor of $Y$ just below $P$. Applying $\text{Map}()$ once again ensures that the representative node is returned.

\[ \square \]

**Theorem 26.** Our faster algorithm takes $O(\text{Ea}(E))$ time.

**Proof:** Follows from $O(E)$ calls to $\text{FindNode}()$, Theorem $22$, and Tarjan's result \cite{17}. \[ \square \]

5 Preliminary Experiments

Although we have described flow graphs where the worst case, quadratic behavior of the standard algorithms does occur, previous experiments \cite{9} have indicated this behavior is not expected in practice on real programs. We performed an experiment, wherein $\phi$-functions were placed (toward construction of SSA form) in $139$ Fortran procedures taken from the Perfect \cite{4} ($\text{Oceen}$, $\text{Spice}$, $\text{QCD}$) benchmark suite and from the Eispack \cite{16} and Linpack \cite{12} subroutine library. In Figure 8 we compare the speed of our simple (i.e., without balanced
path-compression) but asymptotically faster algorithm given in Section 2 with the speed of the usual algorithm [9]; these execution times were obtained on a SparcStation 10, and they represent only the time necessary to compute the location of $\phi$-functions.

![Graph showing speedup vs. execution time](image)

**Fig. 8.** Speedup of our algorithm vs. execution time (in milliseconds) of usual $\phi$-placing algorithm

Most of the runs show that the execution time of our algorithm is linear (with a small constant) in the execution time of the usual algorithm, and so can be expected to also exhibit linear behavior in practice on the same programs. Because each of these runs took under 1 second, both algorithms are fast on this collection of programs. We have not implemented the balanced path-compression, and so these experiments did not reflect any improvements that might be gained by this theoretically more efficient algorithm.

Though not represented in Figure 8, we also experimented with a series of increasingly taller “ladder” graphs of the form shown in Figure 1, where worst-case behavior is expected. Our algorithm demonstrated better performance when these ladder graphs had 10 or more rungs, though smaller graphs take scant execution time anyway. Our algorithm is twice as fast as the usual algorithm for a ladder graph of 75 rungs, taking 20 milliseconds while the usual algorithm took 40 milliseconds.

In summary, comparison of our simple algorithm to the usual algorithm shows

- linear performance for the same cases as the usual algorithm, although our median test case exhibited performance degradation of a factor of $8$;
- a factor of $2$ better performance for some artificially generated cases.

Thus, preliminary evidence indicates comparable expected performance using our simple algorithm.
6 Conclusions and Future Work

In this paper we have shown how to eliminate the potentially costly step of computing dominance frontiers when constructing Sparse Evaluation Graphs or SSA form. By directly determining the conditions under which a node is a $\phi$-node, rather than by iterating through the dominance frontiers, we obtain a worst case almost-linear bound for constructing SEG's and a worst case almost-quadratic bound for constructing SSA form. In both cases, we have eliminated the $O(N^2)$ behavior associated with computing and using dominance frontiers. We have also given preliminary experimental evidence that our simple algorithm's behavior, though slower in many cases, is comparable in practice to the usual algorithm.

Future work will incorporate balanced path-compression into our simple algorithm, and compare the results on real and artificially generated cases.

Acknowledgements

We thank Dirk Grunwald and Harini Srinivasan for their suggestions as we began work on this paper. We are grateful to those who reviewed a preliminary version of this paper [11], especially the reviewer who found a mistake in one of our proofs.

References


This article was processed using the \LaTeX\ macro package with LLNCS style