McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

A New Framework for Exhaustive and Incremental Data Flow Analysis Using DJ Graphs

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

ACAPS Technical Memo 95
July 21, 1995

† Application Development Technology Institute
IBM Software Solutions Division
San Jose, California

A version of this report has been submitted for publication

ACAPS • School of Computer Science • 3480 University St. • Montréal • Canada • H3A 2A7
FTP: ftp-acaps.cs.mcgill.ca • WWW: http://www-acaps.cs.mcgill.ca/

Abstract

In this paper we propose a new framework for elimination-based exhaustive and incremental data flow analysis using DJ graphs. In this paper we give an overview of our framework. The details can be found in our two long reports [SGL95a, SG95a]. These two reports are available from our WWW URL `http://www-acaps.cs.mcgill.ca/~sreedhar/pubs.html` OR `http://www-acaps.cs.mcgill.ca/doc/memos.html`

Contents

List of Figures

List of Tables

1 Introduction

Traditional elimination methods consist of three steps [RP86]: (1) reducing the flowgraph to a single node, (2) eliminating variables in the data flow equations by substitution, and (3) once the solution to the single node is determined, propagating the solution to other nodes to determine their respective solutions. Elimination methods for data flow analysis have been studied by many authors [AC76, Ull73, GW76, Tar81, Bur90, Ros82]. An excellent survey can be found in [RP86].

In this paper we present a new framework for both *exhaustive* and *incremental* data flow analysis based on elimination methods. We will demonstrate that our approach is simple, easy to implement, practically efficient, able to handle irreducible flowgraphs, and amenable to incremental analysis. At the heart of our approach is a simple program representation called the **DJ graph** which has been successfully applied to other problems [SG95b, SGL95b, Sre95]. A DJ graph is just the dominator tree of a flowgraph augmented with *join* edges from the flowgraph (see Figure 1). A flowgraph edge is a join edge (J edge) if it has no corresponding edge in the dominator tree. A dominator tree edge is called a D edge.

Unlike the previous elimination methods that typically reduce a flowgraph to a single node, our approach only eliminates J edges from a DJ graph in a bottom-up fashion on the dominator tree, which may be compressed in the process. It performs variable elimination either eagerly or in a delayed manner. At the end of the bottom-up elimination phase, all the J edges are eliminated, and the data flow solution at every node, except for the root, is expressed only in terms of its parent's solution in the dominator tree. Once we determine the solution for the root node, we propagate this information in a top-down fashion on the dominator tree to compute the solution for every other node.

A novel aspect in our incremental method is that we use the concept of **(iterated) dominance frontier** for updating data flow solutions. Unlike any previous elimination-based incremental method, our approach can handle arbitrary program changes, including structural changes that cause irreducibility. Since we use dominance frontiers in our incremental analysis, we will also show how to incrementally compute the dominance frontier relation.

To summarize, this paper presents several new contributions:

1. We propose two exhaustive elimination methods (in Section 2): (1) the **eager elimination** method, and (2) the **delayed elimination** method. Although both have a worst case time complexity worse than linear in theory, we observed their execution performance to be linear in practice (see Section 5). They are conceptually simple and do not require a 'parse' [HU72, Ull73, Tar74, Tar81] or 'S-set finding' [GW76] to guide the graph reduction and variable elimination. We simply perform the elimination phase, removing J edges and variables, in a bottom-up manner on the DJ graph. In delayed elimination, we use a linear time pre-processing step to identify the exact level

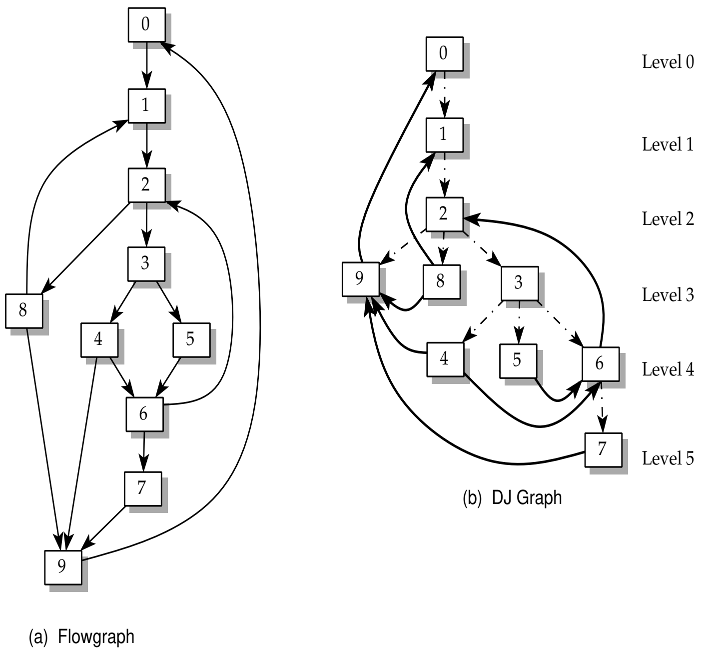

Figure 1: A Flowgraph and its DJ graph

at which each delayed variable should be eliminated. This pre-processing can help avoid the redundant inspection of J edges during the reduction process when we determine whether to eliminate a delayed variable or not.

2. We propose a novel approach to handling irreducibility. Previous approaches form single-entry improper regions, which contain multiple-entry strongly connected components, to accommodate irreducibility [Bur90, Mar89]. By contrast, we perform fixed-point iteration, for an irreducible region, only over the nodes at the same level in the dominator tree during the reduction process (see Section 3). Thus, the number of nodes participating in fixed-point iteration is usually much smaller.

3. We present a new **incremental data flow analysis technique** that can handle arbitrary structural changes, including those causing irreducibility. Its worst case time complexity is $O(|E| \times |N|)$ (see Section 4).

4. We establish a relationship between our approach and **(iterated) dominance frontiers**, and exploit it in two results. First, by utilizing iterated dominance frontiers our incremental method can handle arbitrary program changes, including those causing or removing irreducibility. Since our incremental method uses dominance frontiers and iterated dominance frontiers, we also present an *incremental* algorithm for updating the dominance frontier relation (see Section 4). Second, we exploit the properties of dom-

inance frontiers to demonstrate that both our exhaustive elimination methods should behave *linearly* in practice (see Section 2.3).

5. We have implemented our exhaustive elimination methods for solving the intraprocedural reaching definitions problem. The simplicity of these methods contributed to the ease of their implementation. Our empirical results show that both methods are about 1.3 times as fast as the iterative method (with reverse post order for iteration) for our test suite of 40 Fortran procedures.

Note: This paper only gives an overview of our framework. The details can be found in our two long TR available from
WWW URL site: `http://www-acaps.cs.mcgill.ca/~sreedhar/pubs.html` OR
WWW URL site: `http://www-acaps.cs.mcgill.ca/doc/memos.html`

2 Exhaustive Data Flow Analysis

Both our exhaustive elimination methods consist of two phases: (1) bottom-up DJ graph reduction and variable elimination, called the **elimination phase**; and (2) top-down propagation of data flow solutions, called the **propagation phase**. It is assumed that the initial data flow equations are properly set up according to the original flowgraph.

2.1 The Eager Elimination Method

Our eager elimination method uses the E1 and E2 rules (together called the $\mathcal{E}$-*rules*) to reduce a DJ graph in a bottom-up fashion. Figure 2 gives a graphical illustration of these rules. The $\mathcal{E}$-*rules* are always applied to a J edge $y \rightarrow z$ such that y is a non-join node, and there is no J edge with its source node at a level greater than $y.level$. In this paper we adopt the following relaxed definition of non-join nodes: A node y is a **non-join** node iff $Pred(y)$ contains no nodes other than y and $idom(y)$.[1] We use $\mathcal{G}^0$ to represent the original (zeroth) DJ graph. Each application of the $\mathcal{E}$-*rules* transforms some reduced DJ graph $\mathcal{G}^i$ to $\mathcal{G}^{i+1}$ until the resulting DJ graph "degenerates" into its dominator tree. The E1 rule eliminates a self-loop, and computes the closure of a recursive equation.[2]

Definition 2.1 (E1 rule) *Let $\mathcal{G}^i = (N, E)$ be the ith reduced DJ graph. Let y be a non-join node such that the self-loop $y \rightarrow y$ exists. Let $O_y = f(O_y)$ be the flow equation H_y at node y. The E1 rule is given below:*

- (i) Graph reduction: $E1(<\mathcal{G}^i, N, E, y \xrightarrow{J} y>) \implies <\mathcal{G}^{i+1}, N, E - \{y \xrightarrow{J} y\}>$
- (ii) Variable elimination: $E1(O_y = f(O_y)>) \implies <O_y = f^*(O_y)>$

[1] That is, a self-loop $y \rightarrow y$ will not prevent y from being a non-join node.
[2] Appendix A gives the necessary background material on data flow analysis.

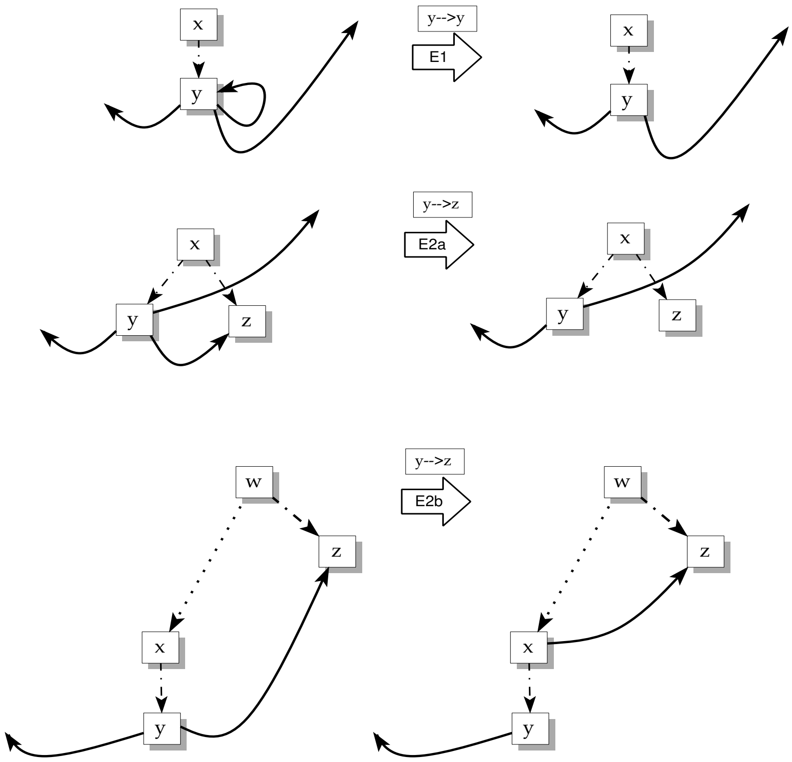

Figure 2: Graphical illustration of $\mathcal{E}$-*rules*.

An E2 rule is applied to a J edge $y \to z$, if y is a non-join node and $y \to y$ does not exist. Recall that $y.level \geq z.level$ for any J edge $y \to z$. We distinguish between two types of E2 rules depending on the levels of y and z. If $y.level = z.level$ we apply E2a rule; otherwise we apply E2b.

Definition 2.2 (E2 rules) *Let $\mathcal{G}^i = (N, E)$ be the ith reduced DJ graph. Let y be a non-join node such that $y \to y$ does not exist. Let $y \to z$ be a J edge, and let $x = idom(y)$. Let $O_y = kO_x + m$ be the equation H_y at node y, such that the parameters k and m does not contain any variables. Finally, let $O_z = aO_y + b$ be the equation H_z at node z, where a and b do not contain the variable O_y. There are two cases:*

(E2a rule) *If $y.level = z.level$, then*

- (i) Graph reduction: $E2a(< \mathcal{G}^i, N, E, y \xrightarrow{J} z >) \implies < \mathcal{G}^{i+1}, N, E - \{y \xrightarrow{J} z\} >$
- (ii) Variable elimination: $E2a(< O_z = aO_y + b >) \implies < O_z = a(kO_x + m) + b >$

(E2b rule) *If $y.level \neq z.level$, then*

- (i) Graph reduction: $E2b(< \mathcal{G}^i, N, E, y \xrightarrow{J} z >) \implies < \mathcal{G}^{i+1}, N, (E - \{y \xrightarrow{J} z\}) \cup \{x \xrightarrow{J} z\} >$ [3]
- (ii)Variable elimination: $E2a(< O_z = aO_y + b >) \implies < O_z = a(kO_x + m) + b >$

The newly inserted edge $idom(y) \to z$ in E2b is called a **derived edge** of $y \to z$, which is removed. An important point to note is that before an E2 rule is applied to an edge $y \to z$, we first eliminate the self-loop $y \to y$, if it exists, by E1. The difference between E2a and E2b is minor, but this distinction is useful when we discuss delayed elimination and when we handle irreducibility.

The algorithm outline for eager elimination is given below. $NumLevel$ is the total number of levels in the DJ graph. *PriorityList*$[i]$ maintains an ordered list of nodes at level i to be processed by procedure **ReduceLevel**(i), such that non-join nodes always appear earlier than join nodes.

Algorithm 2.1 (The Eager Elimination Method) MainExhDFA() *with eager elimination solves a system of data flow equations for a reducible flowgraph.*

```
MainExhDFA()
{
    for i = NumLevel − 1 downto 1 do /* bottom-up elimination */
      ReduceLevel(i) ;
      if(PriorityList[i] is not empty) then
        CollapseIrreducible(i) ;
      endif
    endfor
    DomTDPropagate() ; /* top-down propagation */

}

Procedure ReduceLevel(i)
{
    while((y = GetNode(i)) ≠ NULL) do
      foreach outgoing J edge y → z do
        if(z == y) then /* self-loop */
          Eager1(y → y) ;
        else
          if(idom(y) == idom(z)) then
            Eager2a(y → z) ;
```

[3]We do not insert $x \to z$ in $\mathcal{G}^{i+1}$ if it is already present in $\mathcal{G}^i$.

```
            else
               Eager2b(y → z) ;
            endif
         endif
      endfor
   endwhile
}
```

MainExhDFA() calls **ReduceLevel**() in a bottom-up fashion. When **ReduceLevel**(i) terminates, *PriorityList*[i] will be empty if the flowgraph is reducible. Otherwise, the flowgraph is irreducible. For an irreducible flowgraph, procedure **CollapseIrreducible**(i) is invoked to handle irreducible regions at level i, the details of which are given in Section 3. At the end of the elimination phase, the data flow solution at every node (except for the root) depends only on the solution of its immediate dominator. At this point **MainExhDFA()** calls procedure **DomTDPropagate()** to propagate solutions down the dominator tree and finishes the entire data flow solution process.

Procedure **ReduceLevel**(i) eliminates J edges whose source nodes are at level i (for a reducible flowgraph) by applying the $\mathcal{E}$*-rules*. We ensure that the self-loop edge $y \rightarrow y$, if exists, will be the first outgoing edge from node y. Consequently, when a E2 rule is applied to an outgoing J edge from y, there will be no self-loop at y.

Procedure **GetNode**(i) returns a non-join node at level i whenever one exists. Otherwise, it returns $NULL$ under two circumstances: (1) *PriorityList*[i] is empty; (2) *PriorityList*[i] is not empty but all its remaining nodes are join nodes. The second case indicates the existence of irreducibility.

Procedures **Eager1**(), **Eager2a**(), and **Eager2b**() implement the E1, E2a, and E2b rules, respectively, and their associated variable eliminations. In other words, a variable is eliminated from the RHS of some data flow equation while a J edge is removed from the DJ graph.

Example: Consider the example flowgraph in Figure 1. Figure 3 gives a trace of the graph reduction process. Assume a forward data flow problem with union as the meet operation. The table below gives a partial trace of the variable elimination corresponding to the graph reduction shown in Figure 3.

	Rule	$y \rightarrow z$	$\mathcal{G}^i$ †	$\mathcal{G}^{i+1}$ †	O_z^{i+1}
1	E2b	$7 \rightarrow 9$	(a)	(b)	$P_9O_4 + P_9O_8 + P_9P_7O_6 + P_9G_7 + G_9$
2	E2b	$4 \rightarrow 9$	(b)	(c)	$P_9P_4O_3 + P_9G_4 + P_9O_8 + P_9P_7O_6 + P_9G_7 + G_9$
3	E2a	$4 \rightarrow 6$	(c)	(d)	$P_6P_4O_3 + P_6G_4 + P_6O_5 + G_6$
4	E2a	$5 \rightarrow 6$	(d)	(e)	$(P_6P_4 + P_6P_5)O_3 + P_6G_4 + P_6G_5 + G_6$
5	E2b	$6 \rightarrow 9$	(e)	(f)	$(P_9P_4 + P_9P_7P_6P_4 + P_9P_7P_6P_5)O_3 + P_9O_8 + P_9G_4 + P_9P_7P_6G_4 + P_9P_7P_6G_5 + P_9P_7G_6 + P_9G_7 + G_9$

† Corresponding DJ graphs in Figure 3

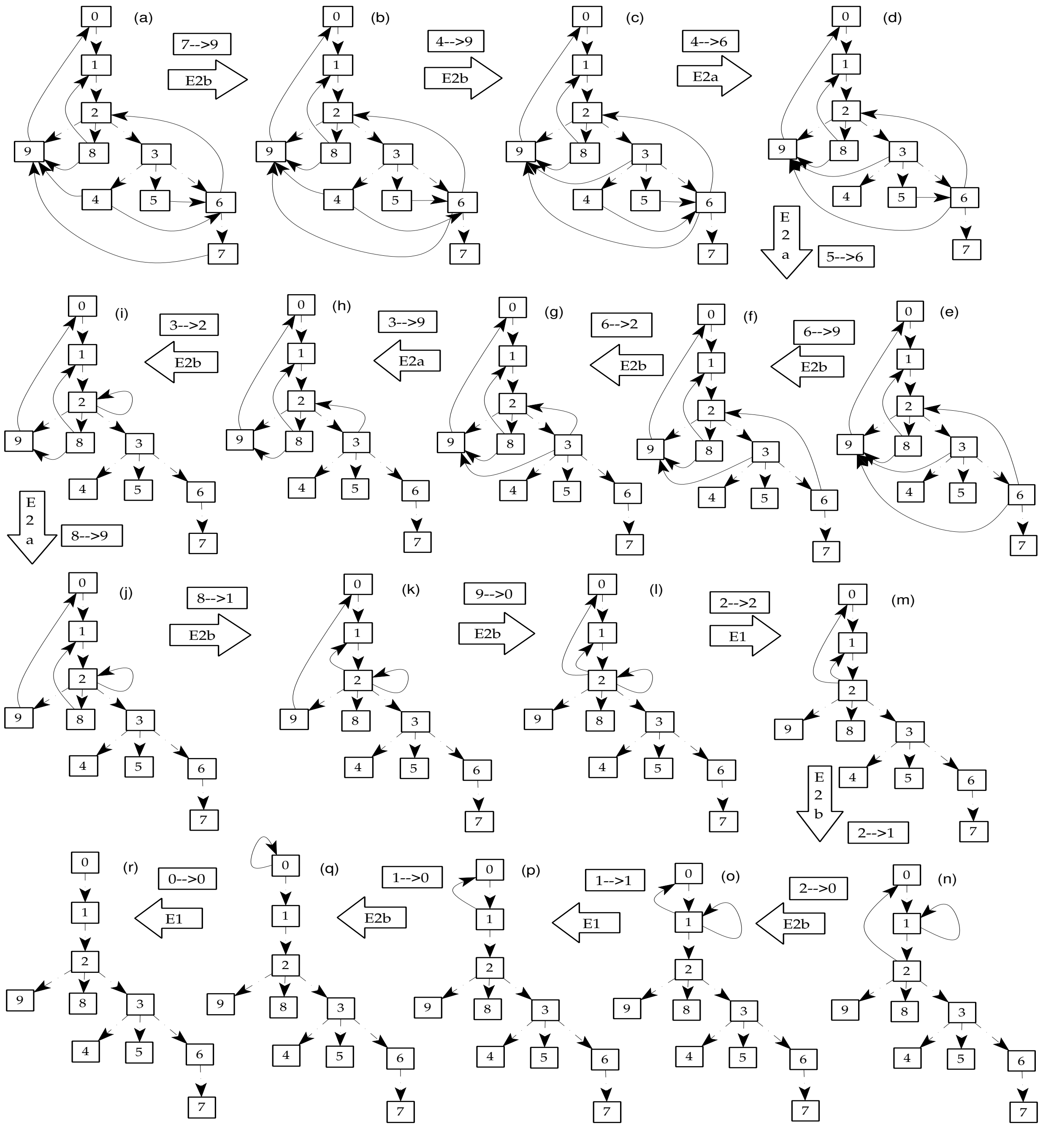

Figure 3: DJ graph reduction with eager elimination (e.g., from a to b, we apply the E2b rule to J edge $7 \to 9$).

2.1.1 Correctness of Eager Elimination

The correctness of eager elimination depends on the correctness of DJ graph reduction, variable elimination, and top-down propagation. Here we provide several lemmas and a theorem to establish this. All proofs can be found in [SGL95a].

Our first lemma states that, for a reducible DJ graph, there must exist a non-join node at a level whose nodes are being processed by procedure **ReduceLevel**().

Lemma 2.1 *Let $\mathcal{G}$ be a DJ graph, and let k be the greatest level number such that there are no J edges originating at levels greater than k. If $\mathcal{G}$ is reducible, then there exists at least one non-join node at level k.*

The second lemma states that applying the $\mathcal{E}$-*rules* preserves the reducibility of a DJ graph.

Lemma 2.2 *Let $\mathcal{G}$ be a reducible DJ graph. Let one of the $\mathcal{E}$-rules be applied to $\mathcal{G}$, resulting in $\mathcal{G}^1$. Then $\mathcal{G}^1$ is also reducible.*

The next lemma ensures that our bottom-up reduction will always reduce a reducible DJ graph to its dominator tree, and the variable elimination performed during the reduction is correct.

Lemma 2.3 *In eager elimination, for a reducible flowgraph, when* **ReduceLevel**(i) *is called at step* $\boxed{\mathbf{2}}$ *and the call terminates:*

1. *all the J edges whose source nodes are at levels greater than or equal to i are eliminated, and*
2. *the RHS of the flow equation at each node y, with $y.level \geq i$, contains only the flow variable $O_{idom(y)}$.*

Our fourth lemma establishes that at the end of the elimination phase, all the J edges have been deleted.

Lemma 2.4 *When the Algorithm 2.1 begins at step* $\boxed{\mathbf{7}}$, *all the J edges have been eliminated from a reducible DJ graph.*

The next lemma ensures that variable elimination is performed correctly in the elimination phase.

Lemma 2.5 *When* **DomTDPropagate()** *is invoked at step* $\boxed{7}$*, the flow equation at each node depends only on the flow variable of its immediate dominator.*

With the above results, we establish the correctness of our eager elimination:

Theorem 2.1 *Algorithm 2.1 correctly computes the solutions to a set of data flow equations for a reducible flowgraph.*

2.2 The Delayed Elimination Method

In this section we demonstrate how to delay certain variable eliminations in our framework so that we can improve the cost of our approach from $O(|E| \times |N|)$ to $O(|E| \times \log |N|)$ function operations. The key intuition behind our **delayed elimination** is based on the following observation. Consider the J edge $7 \rightarrow 9$ in Figure 3(a). The effect of our eager elimination triggered by this edge is as follows.

1. E2b derives $6 \rightarrow 9$ from $7 \rightarrow 9$ (Figure 3(b)).
2. E2b derives $3 \rightarrow 9$ from $6 \rightarrow 9$ (Figure 3(g)).
3. E2a removes $3 \rightarrow 9$ (Figure 3(h)).

Rather than moving the source nodes of derived edges one level up at a time, we can directly generate just one derived edge such that its source and destination nodes are at the same level. We can do so if we know exactly what node is that source node.

For each J edge $y \rightarrow z$, its **top node** is a node $x = \mathbf{Top}_{y \rightarrow z}$ such that x dominates y and $x.level = z.level$. Figure 4 gives a DJ graph, in which every J edge is annotated with its top node $< x >$. Given a DJ graph, we can easily compute the top nodes of *all* J edges in linear time [SGL95b].

An interesting point to note is that the destination node 9 of edges $7 \rightarrow 9$, $6 \rightarrow 9$, and $3 \rightarrow 9$ is in the dominance frontiers of the source nodes 7, 6, and 3. We thus use the term **dominance frontier interval path** of the J edge $7 \rightarrow 9$ to represent the set of nodes on the reverse dominator tree path from node 7 to node 3. The last node on the path is the top node of the J edge.

Now we can define the D2b rule:

$$D2b(< \mathcal{G}^i, N, E, y \stackrel{J}{\rightarrow} z, x >) \implies < \mathcal{G}^{i+1}, N, (E - \{y \stackrel{J}{\rightarrow} z\}) \cup \{x \stackrel{J}{\rightarrow} z\} >$$

where $idom(y) \neq idom(z)$, and $x = \mathbf{Top}_{y \to z}$. No variable elimination is performed by the D2b rule.

We carry over the definition of the E1 and E2a rules to our delayed elimination method to be D1 and D2a, respectively. There is no difference between D1 and E1 (and between D2a and E2a) in terms of graph reduction, but their associated variable eliminations are different. The $\mathcal{D}$-*rules* denotes D1, D2a, and D2b collectively. The algorithm outline for eager elimination is applicable to describing delayed elimination. The only changes are to replace the calls to **Eager1**(), **Eager2a**(), and **Eager2b**() in procedure **ReduceLevel**(i) by the invocations to **Delayed1**(), **Delayed2a**(), and **Delayed2b**() (which implement the $\mathcal{D}$-*rules*), respectively.

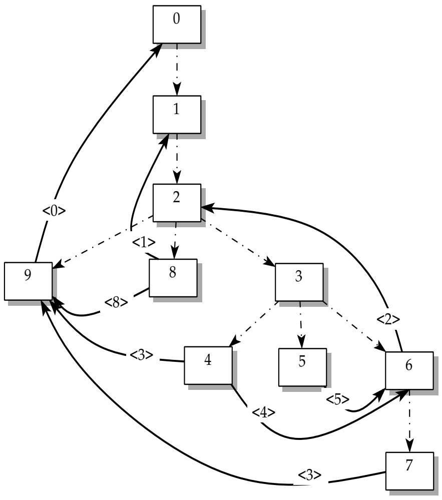

Figure 4: A DJ graph with J edges annotated with their top nodes.

In eager elimination, when the E2b rule is applied to a J edge $y \to z$, we eagerly eliminate variable O_y in equation H_z by replacing it with a function of variable $O_{idom(y)}$. In delayed elimination, we do not eliminate variables in the D2b rule but intentionally delay them instead. A delayed variable will be eliminated eventually when the D1 or D2a rule is applied to the (last) derived edge $x \to z$, where $x = \mathbf{Top}_{y \to z}$, with the following operations performed:

1. **Delayed Substitution:** For each node w on the (maybe compressed) dominator path $x \xrightarrow{+} y$, excluding x, express variable O_w as a function of O_x, by a top-down traversal of the path. Then, eliminate O_y from H_z by substituting it with the RHS of the most recent H_y, which a function of O_x thus computed.

2. **Path Compression:** Make every node on the path $x \xrightarrow{+} y$, excluding x, a child of x.

Note that when a node w on the path $x \xrightarrow{+} y$ is made a child of x, all children of w except the one on the path, are still w's children. Without path compression, the cost of delayed elimination would still be quadratic. But with path compression the complexity becomes $O(|E| \times \log |N|)$. Figure 5 illustrates the graph reduction process for our example DJ graph. At the end of the elimination phase, the flow equation at each node depends only on the flow variable of its parent in the compressed dominator tree, and all the J edges are eliminated. The final data flow solution at each node can be obtained by top-down propagation of data flow information starting at the root node.

The correctness of delayed elimination can be found in [SGL95a].

2.3 The Complexity of Our Exhaustive Elimination Methods

In our eager elimination method, the E1 and E2a rules together will be applied at most $O(|E|)$ times, and the E2b rule will be applied at most $O(|E| \times |N|)$. Therefore, the worst case time complexity of eager elimination is $O(|E| \times |N|)$.

The worst case time complexity of delayed elimination is $O(|E| \times \log |N|)$. This can be established by using the results due to Lucas [Luc90] and Tarjan and van Leeuwen [TvL84].

Although both our exhaustive elimination methods are theoretically worse than linear, they can behave as if they were linear *in practice*. This is evidenced by our empirical results. Furthermore, we state two claims to relate our elimination methods to dominance frontiers, thereby establishing this. The first claim is for eager elimination.

Claim 2.1 *The number of the $\mathcal{E}$-rules applied during the entire elimination phase is bounded by the total size of dominance frontiers of the flowgraph.*

In [CFR$^+$91] Cytron et al. have shown that, in practice, the size of the dominance frontier relation is linear in the size of its flowgraph. Therefore, the time complexity of our eager elimination is expected to be linear in practice.

For delayed elimination, the time complexity is dominated by path compression performed during the D1 and D2a rules. The following claim implies that we can again expect its time complexity to be linear in practice.

Claim 2.2 *In delayed elimination, the total length of the paths compressed during the D1 and D2a rules is bounded by the total size of the dominance frontier relation.*

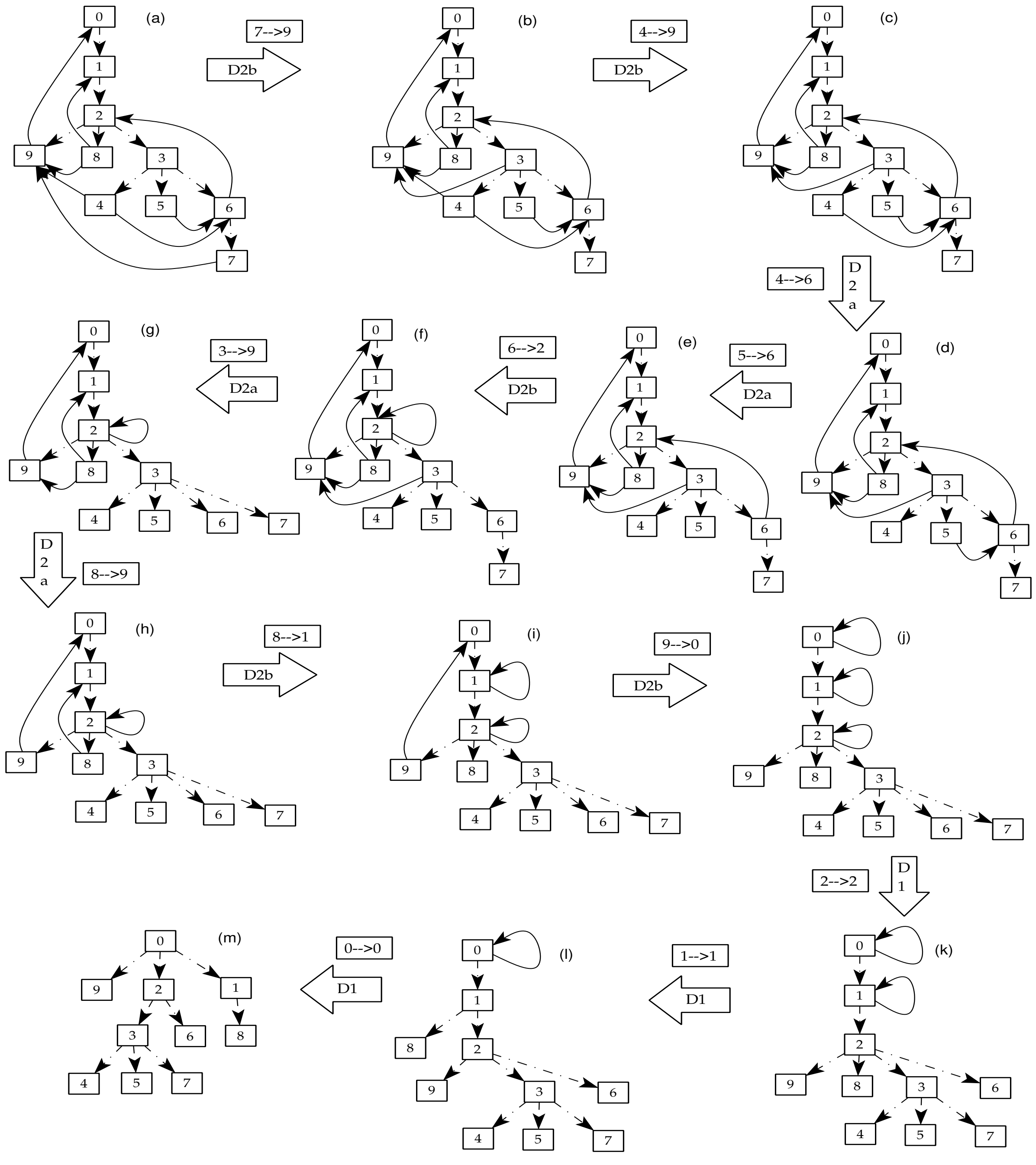

Figure 5: DJ graph reduction with delayed elimination (e.g., from a to b, we apply the D2b rule to J edge $7 \rightarrow 9$; and path compressions are performed from f to g, from j to k, from k to l, and from l to m).

3 Handling Irreducibility

We detect irreducibility if, during the bottom-up reduction, **GetNode**(i) returns $NULL$ but there are still nodes to be processed at level i. Here we outline the operations performed in procedure **CollapseIrreducible**(i), which is called to handle irreducibility detected at level i. A more detailed discussion can be found in [SGL95a].

1. Apply the E2b (or D2b) rule to every J edge whose source node is at level i and destination node is at a level less than i.
2. Perform all variable eliminations that are delayed to this level.
3. Apply Tarjan's Strongly Connected Component (SCC) algorithm on nodes at level i. This can produce a forest of directed acyclic graphs (DAGs) where a node represents an SCC.
4. Process each SCC in each DAG in topological order.
   (a) If the SCC contains only one node, express this node's flow equation in terms of its parent's flow variable.
   (b) If the SCC contains more than one node, perform fixed-point iteration over the nodes in the SCC and then express every node's flow equation in terms of its parent's flow variable.
5. Eliminate all the J edges with both source and destination nodes at level i.

Since we perform the SCC algorithm only on nodes at the same level, the number of nodes in an SCC should be small in practice, as evidenced by our empirical results (see Section 5). Consequently, fixed-point iteration will be performed over a small number of nodes at a time when it is needed.

4 Incremental Data Flow Analysis

In this section we present our incremental method for updating data flow solutions based on our eager elimination method. To simplify the presentation, we will only handle two types of program changes:

- a non-structural change that affects the transfer function of a flowgraph node, and
- a structural change that inserts or deletes a flowgraph edge.

We outline our approach for incremental analysis as follows:

MainIncDFA()
{

if (incremental change is non-structural) **then**
  Update the final flow equations for a non-structural change (Section 4.1)
else
  Update the final flow equations for a structural change (Section 4.2)
endif
Update the final data flow solutions (Section 4.3).

}

The final flow equation at node x is the equation for x at the end of the bottom-up elimination phase; its RHS contains only one variable $O_{idom(x)}$. By contrast, the original equations are called *initial flow equations*. For example, the final flow equation at node 6, in Figure 1, is (where superscript F means final):

$$H_6^F: \quad O_6 = (P_6P_4 + P_6P_5)O_3 + P_6G_4 + P_6G_5 + G_6$$

We can rewrite it to be:

$$H_6^F: \quad O_6 = P_6^F O_3 + G_6^F$$

P^F and G^F are called the *final parameters*, and P and G, the *initial parameters*. All of them have constant values.

The final flow solution at a node is its data flow solution at the end of the top-down propagation phase. In our incremental analysis, we annotate every node in the DJ graph with its initial parameters, final parameters, and final flow solution.

4.1 Updating the Final Flow Equations: Non-Structural Changes

In response to a non-structural program change, we want to know what nodes will need their final flow equations to be updated. Consider the final flow equation H_6^F for node 6. Its final parameters P_6^F and G_6^F are made of the initial parameters for nodes 4 and 5. In general, how are the final parameters at a node related to the initial parameters at some other node(s)? We found a surprisingly simple relation between them.

Claim 4.1 *Let P_w^F and G_w^F be the parameters of the final flow equations at node w. P_y and G_y will appear in P_w^F and G_w^F if and only if either $w = y$ or $w \in IDF(y)$.*

By *appear* we mean that the value of a final parameter is computed from the values of some initial parameters during the elimination phase.

This claim has an important implication in our incremental method. Suppose that we introduce a non-structural change to a node y, thereby affecting P_y and G_y. From Claim 4.1 we should update the final parameters at all nodes in $IDF(y)$. In addition, the direct changes to P_y and G_y naturally affect the final parameters P_y^F and G_y^F at node y. Let $\mathsf{FEqAffected}(y)$ be the set of all nodes whose final parameters (and thus final flow equations) change because of a non-structural change at y (i.e., $\mathsf{FEqAffected}(y)$ is the set of "affected" nodes). We use the following key result to determine this set.

Claim 4.2 *Let the initial parameters P_y and G_y at node y be changed. Then the final flow equation at a node w is* **affected** *(i.e., w is in $\mathsf{FEqAffected}(y)$) if and only if $w \in \{y\} \cup IDF(y)$.*

To efficiently and easily compute $IDF(y)$, we use a DF graph, which is just the dominator tree of a flowgraph plus all edges $u \rightarrow w$ such that $w \in DF(u)$, where $DF(u)$ is the dominance frontier of node u. Figure 6(a) shows the DF graph for our example flowgraph. It is easy to see that $w \in IDF(y)$ if and only if there exists a path from y to w in the DF graph and the path contains no D edges.

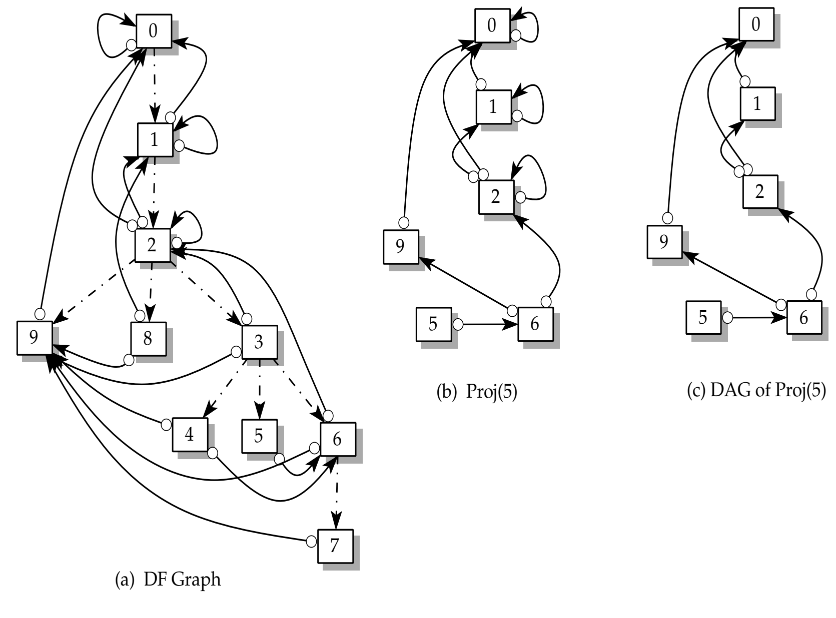

Figure 6: DF graph, $Proj(5)$, and DAG of $Proj(5)$.

Once we have computed $IDF(y)$, we construct the **projection graph** $Proj(y)$ with respect to the nodes in $\{y\} \cup IDF(y)$. $Proj(y)$ is an *induced subgraph* of a DF graph and has its node set as $\{y\} \cup IDF(y)$. Figure 6(b) gives an example projection graph $Proj(5)$.[4]

The projection graph helps provide an ordering in which we can update the final flow equations at the affected nodes. We first apply Tarjan's Strongly Connected Component (SCC) algorithm to the projection graph. Then we determine a topological ordering on the strong components. Finally we visit the strong components S in topological order to update their final flow equations. There are three cases to consider:

Case 1: S contains a single node s and has no self-loops. Assume that the final flow equation at every predecessor of node s is correct (either previously updated or not affected by the incremental change). To determine the updated final flow equation H_s^F at node s, we start with its initial flow equation. Recall that the final flow equation at s should only have one variable $O_{idom(s)}$ on its RHS. Therefore, we eliminate every variable O_w other than $O_{idom(s)}$ on the RHS of a "transient" equation at node s by substituting it with a function of $O_{idom(w)}$ in a bottom-up fashion (on the DJ graph), similar to exhaustive eager elimination but without the graph reduction. In the end, we obtain the updated final flow equation at s, which contains only constant parameters and variable $O_{idom(s)}$.

Case 2: S contains a single node s and has a self-loop. Here we assume that the final flow equation at every predecessor of node s, excluding s itself, is correct. We perform the operations described in Case 1 but allow variable O_s in addition to $O_{idom(s)}$ to be present on the RHS of an equation. Finally we perform a closure operation to break the dependency of O_s on itself and obtain the updated final flow equation H_s^F.

Case 3: S contains more than one node. We encounter irreducibility, so we must simultaneously determine the final flow equations for all nodes in S. These nodes have the same immediate dominator. In this case we assume that the final flow equation at every predecessor of any node $s \in S$ is correct (if this predecessor node is not in S). Similar to what is performed in the above two cases, we eliminate every variable O_w from the equation for any node $s \in S$ by appropriate substitution. However, we allow any variable $O_s, s \in S$, to be present on the RHS of an equation. Finally we perform fixed-point iteration over the "almost final" flow equations at all nodes in S and obtain their updated final flow equations.

4.2 Updating the Final Flow Equations: Structural Changes

We handle structural changes with the following steps:

- Update the dominator tree of the flowgraph.

[4] Note that a projection graph can be constructed during the IDF computation.

- Update the DF graph.
- Update the final flow equations.

In [SGL95b] we gave a simple algorithm for updating the dominator tree (and thus the DJ graph) of a flowgraph. Recall that $p \to q$ is a DF edge in a DF graph iff $q \in DF(p)$. Therefore the problem of updating DF graphs is isomorphic to the problem of updating the dominance frontier relation. Let $x \to y$ be some edge that is inserted/deleted in the flowgraph. Without loss of generality, we assume that both x and y are reachable from the START node before and after the update. We can easily handle other cases, where the reachability of x and/or y changes, as special cases [SGL95b, SG95a].

Updating Dominance Frontiers

Let $\mathsf{DomAffected}(y)$ be the set of all nodes whose immediate dominators change when $x \to y$ is updated in the flowgraph. In [SGL95b] we showed, for the insertion case, that the new immediate dominator of all the nodes that are in $\mathsf{DomAffected}(y)$ is $nca(x, y)$, the nearest common ancestor of x and y on the dominator tree.

Given $\mathsf{DomAffected}(y)$ and $nca(x, y)$, we next show how to incrementally compute the dominance frontier relation in response to an edge being newly inserted into the flowgraph. Recall that a node $w \in \mathsf{DomAffected}(y)$ will move up in the dominator tree after an edge is inserted in the flowgraph [CR88]. When this happens, the dominance frontier of a node that dominates node w or x prior to the update will *possibly* be affected. Now let $z = nca(x, y)$, and let $\mathsf{DFAffectedI}(y) = \{u | z$ **stdom** u **and** u dominates a node $w \in \{x\} \cup \mathsf{DomAffected}(y)$ **prior** to the DJ graph update$\}$. We claim that if a node is not in $\mathsf{DFAffectedI}(y)$, then its dominance frontiers will not change. But for a node in $\mathsf{DFAffectedI}(y)$, its dominance frontier may change or may not. Notice that the set $\mathsf{DFAffectedI}(y)$ can be easily computed by a simple bottom-up traversal of the (old) dominator tree starting from nodes in $\{x\} \cup \mathsf{DomAffected}(y)$.

Once we have determined the set of possibly affected nodes, we can update their dominance frontiers. To do this, we first update the dominator tree using the algorithm given in [SGL95b]. Next, for each node $w \in \mathsf{DFAffectedI}(y)$, we recompute $DF(w)$ in a bottom-up fashion on the new DJ graph as follows: Let w be a node in $\mathsf{DFAffectedI}(y)$ and assume that the dominance frontier of every child of w is correct. Then the new, updated $DF(w)$ can be computed with the following formulas [CFR+91, SG95a].

$$
\begin{aligned}
DF_{local}(w) &= \{r | w \to r \text{ is a J edge}\} \\
DF_{up}(p) &= \{q | q \in DF(p) \textbf{ and } q.level \le idom(p).level\} \\
DF(w) &= DF_{local}(w) \cup \bigcup_{p \in Children(w)} DF_{up}(p)
\end{aligned}
\tag{1}
$$

The effect of deleting an edge is opposite and complementary to that of an edge insertion. More notably, when deleting the edge $x \rightarrow y$, we pull down the affected nodes [CR88]. So we must recompute the dominance frontiers of all possibly affected nodes. Again let $z = nca(x, y)$, and let $\mathsf{DFAffectedD}(y) = \{u | z$ **stdom** u **and** u dominates a node $w \in \{x\} \cup \mathsf{DomAffected}(y)$ **after** the DJ graph update$\}$. Note that here we compute $\mathsf{DFAffectedD}(y)$ *after* updating the DJ graph. Now we claim that if a node is not in $\mathsf{DFAffectedD}(y)$, its dominance frontier does not change. Once the $\mathsf{DFAffectedD}(y)$ set is determined, we update the dominance frontier for every node in $\mathsf{DFAffectedD}(y)$ as in the insertion case.

Updating the Final Flow Equations

With the updated dominance frontier relation, we next describe how to update the final flow equations at all the "affected" nodes. Let the edge $x \rightarrow y$ be the structural update (either inserted or deleted). Now the key question to ask is: At which nodes are the final data flow equations affected when edge $x \rightarrow y$ is updated? The answer to it is in the following claim:

Claim 4.3 *Let $x \rightarrow y$ be the edge that is updated in the flowgraph. The final data flow equation at a node w is affected if and only if $w \in \{y\} \cup IDF(y)$.*

This is very interesting. Recall that in Section 4.1 we made a similar claim (Claim 4.2) for non-structural updates. This implies that we can essentially use the same approach as given in Section 4.1 for updating the final data flow equations even in response to structural changes. But rather than using the 'old' DF graph, we must use the updated DF graph to compute the final flow equations.

4.3 Updating the Final Flow Solutions

Once the final data flow equations have been updated, we can update the final flow solution α^F for every affected node. Let w be a node in $Proj(y)$ with the *new* final data flow equation:

$$O_w^{Fnew} = P_w^{Fnew} O_{idom(w)}^{Fnew} + G_w^{Fnew}$$

Let α_w^{Fold} be the old (and maybe incorrect) final solution at this node. Let $u = idom(w)$ have the correct data flow solution α_u^{Fcor}. By correct we mean that u's flow solution either is not affected or has been correctly updated. Then we can compute the updated data flow solution at node w as follows:

$$\alpha_w^{Fcor} = P_w^{Fnew} \alpha_u^{Fcor} + G_w^{Fnew}$$

If the following relation holds at node w:

$$\alpha_w^{Fold} = \alpha_w^{Fcor},$$

then we need not update the final solutions for the child nodes of w. Otherwise, we must visit each child of w and repeat what was just done for w. Note that a node's final flow solution can change even when its final flow equation stays the same.

It is important to observe that before we can update the final solution at a node w, we must ensure that the final solution of its immediate dominator is correct. Also, once (and if) we update w's final solution we have to mark all its child nodes as potentially affected, and so their final solutions must be updated. To do this, we order the nodes in $Proj(y)$ with their levels on the dominator tree and process them in a top-down fashion.

4.4 Correctness and Complexity

The worst case time complexity of our incremental analysis is $O(|E| \times |N|)$. The proofs of the complexity and the correctness of **MainIncDFA()** can be found in [SG95a].

5 Experiments and Empirical Results

We have implemented our eager and delayed elimination methods for solving the intraprocedural reaching definitions problem. For comparison, we also implemented the iterative method that uses reverse post order for iteration. Our implementation was built upon the Parafrase2 compiler. For efficient set manipulation, we implemented the Briggs-Torczon sparse sets technique [BT93].

We used as test data a set of 40 FORTRAN procedures taken from SPEC92, RiCEPS, LAPACK, and GATOR. We carried out our experiments on a SPARC-20 workstation. Details of our experiments can be found in [SGL95a].

- For our test data, both the eager and the delayed elimination method behaved as if they were linear algorithms.
- Both eager and delayed elimination are, on average, about 1.3 times as fast as the iterative method.
- Let C and NC be the total length of the *dominance frontier interval paths* with and without path compressions in effect, respectively. The average ratio of $\frac{NC}{C}$ is about 1.3. This suggests that, in an ideal situation, delayed elimination can be about 1.3 times as fast as eager elimination. In practice, delayed method incurs overhead from bookkeeping and non-profitable path compressions.
- The number $1 - \frac{C}{NC} = 23\%$ can serve as an estimate of the percentage of edges that lie on more than one dominance frontier interval path. It is also a rough estimate of the cost savings of delayed elimination over eager elimination.

- Of the 40 procedures tested, five procedures have irreducible loops. Three of them have only one irreducible loop each. The maximum size of the SCCs is just 4 (found in procedure `coef`) in our handling irreducibility.

Tables 1 and 2 give a summary of our empirical results. Figure 7 profiles the execution performance of the three data flow analysis methods for our test data. The notation used in the tables are given below:

Notation used in Table 1 and Table 2

Name	Name of the procedure.
$\lvert N\rvert$	Number of flowgraph nodes.
$\lvert E_f\rvert$	Number of flowgraph edges.
$\lvert G\rvert$	Size of GEN set (i.e., total number of downward exposed definitions)
Ir	Number of times the procedure **CollapseIrreducible**() is called
S	Maximum size of SCCs detected during **CollapseIrreducible**()
$ED1$	Number of times E1 (or D1) rule applied
$ED2a$	Number of times E2a (or D2a) rule applied
$E2b$	Number of times E2b rule applied
nE	Total number of $\mathcal{E}$-$rules$ applied
$D2b$	Number of times D2b rule applied
nD	Total number of $\mathcal{D}$-$rules$ applied
NC	Total length of the dominance frontier interval paths w/o path compression
C	Total length of the dominance frontier interval paths with path compression
$\frac{NC}{C}$	Ratio of NC to C
$\lvert DF_e\rvert$	Size of dominance frontiers in the flowgraph
nI	Number of iterations for the iterative method to converge
T_i	Execution time in seconds for iterative method
T_e	Execution time in seconds for eager elimination method
T_d	Execution time in seconds for delay elimination method
$S_{i/e}$	$\frac{T_i}{T_e}$
$S_{i/d}$	$\frac{T_i}{T_d}$
$S_{d/e}$	$\frac{T_e}{T_d}$

6 Related Work

6.1 Elimination Methods

Our two exhaustive elimination methods are related to all of the four classical elimination methods: the Allen-Cocke method, the Hecht-Ullman method, the Graham-Wegman method,

Name	$\lvert N\rvert$	$\lvert E_f\rvert$	$\lvert G\rvert$	Ir	S	$ED1$	$ED2a$	$E2b$	nE	$D2b$	nD	NC	C	$\frac{NC}{C}$	$\lvert DF_e\rvert$
aerset	329	460	166	0	0	44	79	200	323	81	204	232	213	1.1	371
aqset	189	258	74	0	0	24	42	118	184	48	114	127	116	1.1	202
bjt	135	187	213	0	0	1	62	114	177	50	113	132	95	1.4	211
card	150	216	63	2	3	11	43	170	224	58	112	230	165	1.4	316
chemset	229	320	94	0	0	23	68	191	282	57	148	198	166	1.2	300
chgeqz	174	248	115	0	0	13	58	142	213	61	132	203	141	1.4	298
clatrs	214	308	160	0	0	17	79	129	225	76	172	152	142	1.1	276
coef	95	137	117	1	4	11	32	69	112	32	75	82	59	1.4	142
comlr	69	91	63	1	3	11	13	47	71	20	44	49	49	1.0	78
dbdsqr	228	327	144	0	0	19	74	176	269	75	168	275	188	1.4	391
dcdcmp	137	187	75	1	2	14	35	127	176	50	99	176	127	1.4	245
dcop	186	261	127	0	0	10	87	143	240	69	166	182	154	1.2	295
dctran	326	458	151	0	0	10	119	278	407	93	222	577	273	2.1	745
deseco	175	236	240	0	0	18	54	107	179	47	119	135	114	1.2	220
dgegv	160	232	108	0	0	11	51	102	164	55	117	200	120	1.7	287
dgesvd	321	470	405	0	0	6	102	206	314	111	219	406	257	1.6	585
dhgeqz	285	408	261	0	0	32	92	238	362	94	218	335	221	1.5	484
disto	133	191	183	0	0	7	53	98	158	58	118	107	102	1.1	186
dlatbs	167	238	134	0	0	12	60	102	174	57	129	127	113	1.1	220
dtgevc	321	459	257	0	0	50	93	213	356	114	257	274	230	1.2	439
dtrevc	248	353	148	0	0	21	85	157	263	75	181	192	173	1.1	318
elprnt	162	227	138	0	0	18	59	99	176	44	121	99	96	1.0	183
equilset	327	451	179	0	0	55	74	205	334	93	222	212	202	1.1	353
errchk	346	482	154	0	0	40	99	267	406	102	241	358	278	1.3	528
iniset	333	486	280	0	0	154	0	154	308	154	308	154	154	1.0	308
init	122	175	129	0	0	17	35	66	118	31	83	70	68	1.0	125
initgas	189	263	114	0	0	40	37	126	203	52	129	126	126	1.0	205
jsparse	281	403	214	0	0	52	70	185	307	76	198	185	183	1.0	313
modchk	306	419	137	0	0	25	111	205	341	74	210	240	213	1.1	390
moseq2	161	217	248	0	0	2	67	137	206	50	119	181	122	1.5	267
mosfet	214	295	263	0	0	1	91	231	323	76	168	307	184	1.7	427
noise	115	159	135	0	0	7	57	72	136	43	107	77	76	1.0	147
out	403	589	404	0	0	92	92	306	490	136	320	330	313	1.1	525
reader	182	235	159	0	0	8	53	28	89	15	76	28	28	1.0	89
readin	406	611	240	2	2	29	84	685	798	174	285	1457	574	2.5	1675
setupgeo	188	275	195	0	0	44	37	112	193	61	142	127	118	1.1	218
setuprad	195	286	223	0	0	55	38	124	217	68	161	124	124	1.0	220
smvgear	212	310	224	0	0	66	23	248	337	91	180	363	203	1.8	468
solveq	196	289	186	0	0	72	28	160	260	93	193	162	162	1.0	265
twldrv	168	243	594	0	0	16	59	127	202	49	124	189	133	1.4	280
Average	219	312	188	0.2	0.3	29	62	167	258	72	163	229	164	1.3	341

Table 1: Structural characteristics

Name	$\lvert E_f \rvert$	nI	T_i	T_e	T_d	$S_{i/e}$	$S_{i/d}$	$S_{d/e}$
aerset	460	6	2.6	1.4	1.4	1.9	1.9	1.0
aqset	258	4	0.6	0.5	0.7	1.2	0.9	0.7
bjt	187	3	1.1	1.0	0.9	1.1	1.2	1.1
card	216	5	0.7	0.5	0.3	1.4	2.3	1.7
chemset	320	4	0.8	0.8	0.7	1.0	1.1	1.1
chgeqz	248	4	0.8	0.7	0.5	1.1	1.6	1.4
clatrs	308	3	0.7	0.9	1.0	0.8	0.7	0.9
coef	137	6	1.6	0.5	0.4	3.2	4.0	1.2
comlr	91	5	0.3	0.4	0.5	0.7	0.6	0.8
dbdsqr	327	4	1.3	1.1	1.0	1.2	1.3	1.1
dcdcmp	187	5	0.6	0.5	0.5	1.2	1.2	1.0
dcop	261	3	0.5	0.7	0.7	0.7	0.7	1.0
dctran	458	4	1.6	1.5	1.2	1.1	1.3	1.2
deseco	236	4	1.2	1.1	1.2	1.1	1.0	0.9
dgegv	232	3	0.5	0.6	0.5	0.8	1.0	1.2
dgesvd	470	2	1.1	2.9	3.5	0.4	0.3	0.8
dhgeqz	408	4	2.4	2.1	1.8	1.1	1.3	1.2
disto	191	4	1.1	0.9	0.9	1.2	1.2	1.0
dlatbs	238	3	0.5	0.6	0.6	0.8	0.8	1.0
dtgevc	459	6	4.1	2.2	2.2	1.9	1.9	1.0
dtrevc	353	4	1.2	1.0	1.2	1.2	1.0	0.8
elprnt	227	4	0.5	0.5	0.5	1.0	1.0	1.0
equilset	451	5	2.0	1.5	1.4	1.3	1.4	1.1
errchk	482	4	1.2	1.5	1.5	0.8	0.8	1.0
iniset	486	2	1.5	2.2	2.3	0.7	0.7	1.0
init	175	5	0.9	0.5	0.5	1.8	1.8	1.0
initgas	263	4	0.9	0.8	0.8	1.1	1.1	1.0
jsparse	403	5	2.2	1.6	1.6	1.4	1.4	1.0
modchk	419	5	1.4	1.2	1.2	1.2	1.2	1.0
moseq2	217	3	1.0	1.2	1.1	0.8	0.9	1.1
mosfet	295	3	2.0	2.1	1.6	1.0	1.2	1.3
noise	160	3	0.4	0.5	0.5	0.8	0.8	1.0
out	590	6	6.3	3.9	4.1	1.6	1.5	1.0
reader	235	5	1.2	0.6	0.7	2.0	1.7	0.9
readin	611	4	3.7	3.3	2.4	1.1	1.5	1.4
setupgeo	275	4	1.2	1.1	1.1	1.1	1.1	1.0
setuprad	286	5	1.8	1.2	1.2	1.5	1.5	1.0
smvgear	310	6	3.4	1.7	1.6	2.0	2.1	1.1
solveq	289	7	2.1	1.1	1.2	1.9	1.8	0.9
twldrv	243	9	10.8	3.6	3.4	3.0	3.2	1.1
Average	312	4	1.7	1.3	1.3	1.3	1.4	1.1

Table 2: Timings and speedups

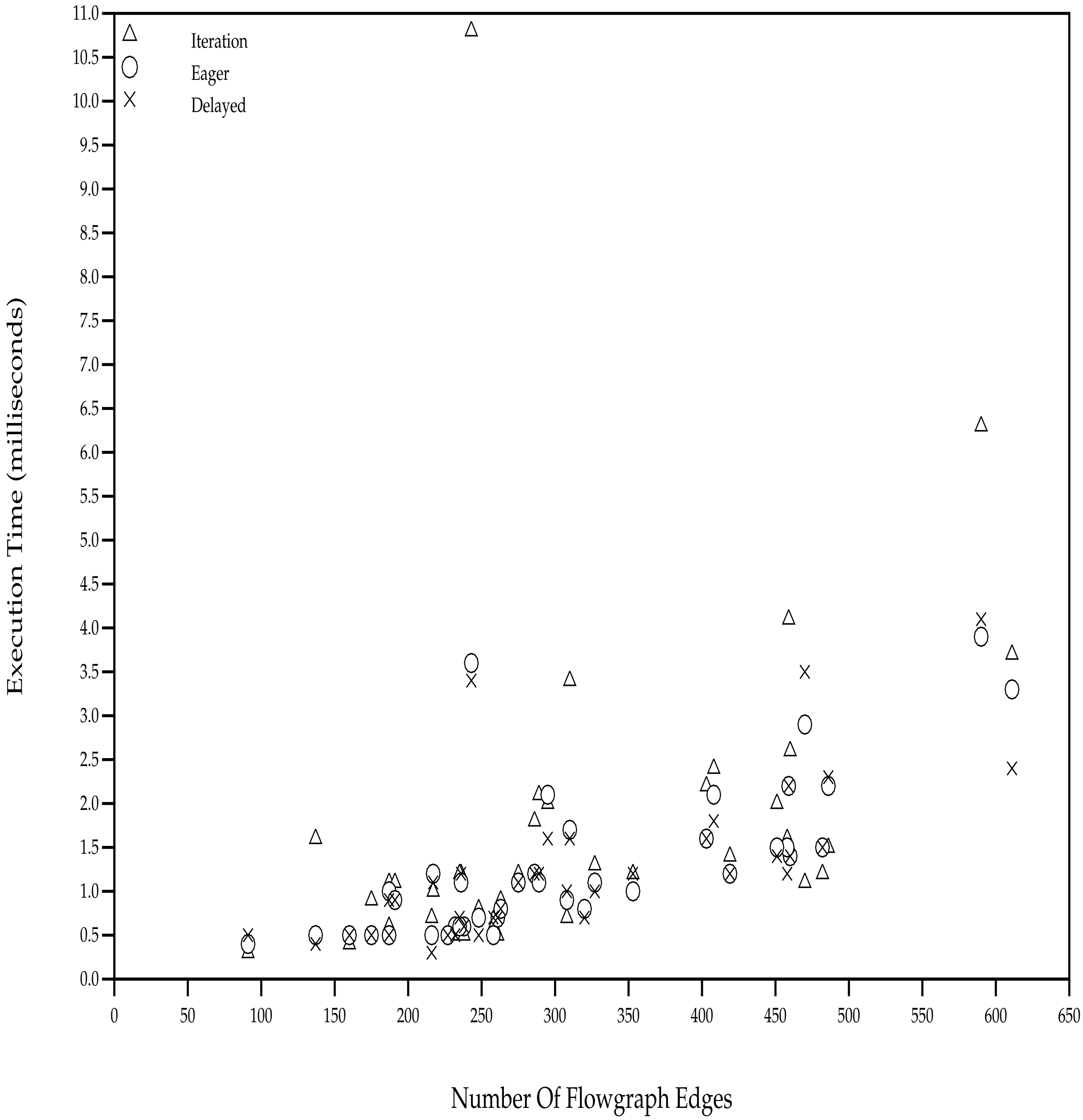

Figure 7: The execution profile of three data flow analysis methods on 40 Fortran procedures.

and the Tarjan method. But our approach has several significant differences. Except for the Graham-Wegman method, these elimination methods are applicable only to reducible flow graphs. They mainly use two approaches to handle irreducible flow graphs. One is *node splitting*, which generates an "equivalent," reducible flowgraph by splitting nodes [Hec77]. The other approach is to form improper regions to accommodate irreducibility and use fixed-point iteration in those regions [Bur90]. We take the second approach but utilize the strength of elimination methods within the reducible portions of an improper region. Consequently, fixed-point iteration is performed on a normally much smaller set of equations.

Our delayed elimination method is comparable to Tarjan's path compression-based interval analysis, but simply performed on a static dominator tree. Our reduction rules are similar to Hecht and Ullman's T1 and T2 rules. But we use the notion of top nodes to remember delayed variables, whereas Hecht and Ullman use a more complicated data structure, height-balanced 2-3 tree, to save delayed variables.

The Graham-Wegman method uses non-disjoint sets called the S-sets, which are strongly connected regions and analogous to Tarjan's intervals, for reduction and elimination. Unlike Tarjan's method, in the Graham-Wegman method not all the nodes in an S-set are collapsed into the S-set entry node. The variables representing solutions at the remaining nodes, therefore, still exist in a reduced system of equations after the S-set is processed, thereby making explicit the delayed substitutions of variables. These delayed variables are remembered in a reduced flowgraph. Unlike our approach, Graham and Wegman perform path compression on a depth-first spanning tree. Furthermore, in their T2′ rule they need to "inspect" which outgoing edges from a node are within the current S-set. In our case, the D2b rule eliminates such edges so we never need to "inspect" any J edges during path compression.

Elimination methods are general-purpose data flow solution procedures [Mar89, Bur90, Ros81, Tar81]. In [Bur90] Burke reformulates Tarjan's interval analysis so that it can be applied to any monotone data flow problem. Burke proposes to use the closure of an interval in order to summarize local data flow information for monotone problems. In our approach, we similarly define a closure operation for a recursive data flow equation.

In Section 5 of [Tar81], Tarjan defines a derived graph G' of a flow graph G in order to solve path problems on both reducible and irreducible graphs. Using our terms, we observe that all the D edges in G also appear in G'. On the other hand, for each J edge in G, the corresponding edge in G' is exactly the same as the new J edge we would create in our D2b rule. We suspect that this coincidence may partially explain why we can handle irreducibility gracefully.

6.2 Incremental Data Flow Analysis

For incremental analysis, previous work most relevant to ours is due to Carroll and Ryder [CR88]. Carroll and Ryder's approach is based on the notion of *reduce and borrow* for updating data flow solutions [CR88]. They *reduce* a monotone data flow problem to an attributed (dominator) tree problem, and then *borrow* the well-known Reps attribute update algorithm for updating data flow solutions [Rep82, RTD83]. They use the Graham-Wegman method as a starting point for mapping data flow problems to attributed dominator tree problems [GW76]. In contrast, we utilize properties of (iterated) dominance frontiers for updating data flow solutions. Unlike Carroll and Ryder's method, we can handle both reducible and irreducible flowgraphs.

Burke proposes an elimination-based incremental data flow analysis technique that uses interval graphs for updating and propagating data flow solutions [Bur90]. His approach can only handle structural changes (to flowgraphs) that do not change the depth-first spanning tree of the flowgraph.

Marlowe and Ryder propose an incremental hybrid method that combines iterative and elimination methods [MR90]. They first identify the strongly connected components in the flowgraph. Then they use fixed-point iteration within each component and propagate data flow information among components using an elimination-like method. Although they can handle arbitrary program updates, their incremental method is more coarse-grained; they update and propagate solutions to a much larger set of nodes than our approach or the Carroll-Ryder approach.

7 Conclusion

We have presented a new framework for both exhaustive and incremental data flow analysis using the DJ graph. Both the eager and the delayed exhaustive elimination method are simple, practically fast, easy to implement, and able to handle irreducible flowgraphs efficiently by utilizing the properties of DJ graphs. Our incremental method can handle arbitrary program changes, including those causing irreducibility in the flowgraph. We have implemented both exhaustive methods for solving reaching definitions and experimented with them on 40 FORTRAN procedures. Our experimental results show that they are 1.3 times as fast as the iterative method for our test data. These results also show that eager elimination is competitive in running time when compared to delayed elimination. It is also simpler and easier to implement, and is amenable to incremental data flow analysis. We thus recommend its use in practice.

Acknowledgement

The first and second authors thank the National Sciences and Engineering Research Council (NSERC) and the Canadian Centers of Excellence (IRIS) for their continued support of this research. The last author would like to thank Vivek Sarkar for his management support.

We also thank Adam Buchsbaum for his pointing out an error in one of our original proofs on delayed elimination and directing us to some key results on path compression. We also thank Tom Marlowe for his comments that help improve our presentation.

References

[AC76] Allen, F. E., and Cocke, J. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1976.

[BT93] Briggs, P., and Torczon, L. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1-4):59–63, 1993.

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

[CFR+91] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):452–490, October 1991.

[CR88] Carroll, M., and Ryder, B. G. Incremental data flow update via attribute and dominator updates. In *ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, pages 274–284, January 1988.

[GW76] Graham, S. L., and Wegman, M. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM*, 23(1):172–202, January 1976.

[Hec77] Hecht, M. S. *Flow Analysis of Computer Programs*. Elsevier North-Holland, Inc., 1977.

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

[Kil73] Kildall, G. A. A unified approach to global program optimization. In *Conference Record of the First ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, 1973. ACM SIGACT and SIGPLAN.

[KU76] Kam, J. B., and Ullman, J. D. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[Luc90] Lucas, J. M. Postorder disjoint set union is linear. *SIAM Journal of Computing*, 19(5):868–882, October 1990.

[Mar89] Marlowe, T. J. *Data Flow Analysis and Incremental Iteration.* PhD thesis, Rutgers University, New Brunswick, New Jersey, October 1989.

[MR90] Marlowe, T. J., and Ryder, B. G. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–196. ACM SIGACT and SIGPLAN, January 1990.

[Rep82] Reps, T. Optimal-time incremental semantic analysis for syntax-directed editors. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages.* ACM SIGACT and SIGPLAN, January 1982.

[Ros81] Rosen, B. K. Linear cost is sometimes quadratic. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 117–124. ACM SIGACT and SIGPLAN, January 1981.

[Ros82] Rosen, B. K. A lubricant for data flow analysis. *SIAM Journal of Computing*, 11(3):493–511, August 1982.

[RP86] Ryder, B. G., and Paull, M. C. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[RTD83] Reps, T., Teitelbaum, T., and Demers, A. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[SG95a] Sreedhar, V. C., and Gao, G. R. An elimination-based approach to incremental data flow analysis. Technical Report ACAPS Memo 94, McGill University, June 1995.

[SG95b] Sreedhar, V. C., and Gao, G. R. A linear time algorithm for placing ϕ-nodes. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1995. A longer version accepted in Journal of Programming Languages.

[SGL95a] Sreedhar, V. C., Gao, G. R., and Lee, Y. Efficient data flow analysis using DJ graphs: Elimination methods revisited. Technical Report ACAPS Memo 93, McGill University, June 1995.

[SGL95b] Sreedhar, V. C., Gao, G. R., and Lee, Y. Incremental computation of dominator trees. In *ACM SIGPLAN Workshop on Intermediate Representation (in conjunction with the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages)*, January 1995. Also in appear as SIGPLAN Notices Volume 30, Number 4.

[Sre95] Sreedhar, V. C. *Efficient Program Analyses Using DJ Graphs.* PhD thesis, School of Computer Science, McGill University, 1995. In preparation.

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

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

[TvL84] Tarjan, R. E., and van Leeuwen, J. Worst-case analysis of set union algorithms. *Journal of the ACM*, 31(2):245–281, 1984.

[Ull73] Ullman, J. D. Fast algorithms for the elimination of common subexpressions. *Acta Informatica*, 2(3):191–213, 1973.

A Background and Notation

A.1 Flowgraph Terminology

A **flowgraph** is a directed graph $G = (N, E, \mathrm{START})$, where N is the set of nodes, E is the set of flow edges, and $\mathrm{START} \in N$ is the distinguished start node.[5] A *path* of length n is a sequence of edges $x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_n$, where every $x_i \rightarrow x_{i+1} \in E$. Given the flowgraph of a program, we define the reachable subgraph $\mathsf{REACH}(\mathrm{START})$ to be a subgraph of G such that all nodes in $\mathsf{REACH}(\mathrm{START})$ are reachable from START.

In the reachable subgraph $\mathsf{REACH}(\mathrm{START})$, 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 when x does not dominate y. If x **dom** y and $x \neq y$, then x **strictly dominates** y. We write x **stdom** y to indicate x strictly dominates y, and write x **!stdom** y when x does not strictly dominate y. The dominance relation is reflexive and transitive, and can be represented by a tree, called the dominator tree. If x is a parent node of y in a dominator tree, then x **immediately dominates** y; we write $idom(y)$ to denote the immediate dominator of y. Given a node x in the dominator tree, we define $SubTree(x)$ to be the dominator subtree rooted at x. Note that the nodes in $SubTree(x)$ are simply those dominated by x.

For each node in the dominator tree we associate a *level number* that is the depth of the node from the root. We write $x.level$ to indicate the level number of a node x. $\mathrm{START}.level$ is assumed to be zero.

The **dominance frontier** $DF(x)$ of a node x is the set of all y such that x dominates a predecessor of y but does not strictly dominate y [CFR+91]. It can be extended to mean a set of edges $z \rightarrow y$ such that x dominates z but does not strictly dominate y.

[5] Note that a flowgraph does not need to be connected; that is, some nodes may not be reachable from the START node.

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

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

We define **iterated dominance frontier** $IDF(S)$ for a set of nodes S as the limit of the increasing sequence:

$$\begin{aligned} IDF_1(S) &= DF(S), \qquad (2) \\ IDF_{i+1}(S) &= IDF(S \cup IDF_i(S)) \qquad (3) \end{aligned}$$

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

> **Clarification:** We would like to clarify some terminology in order to avoid confusion in later discussions. Dominator trees and DJ graphs are constructed only for the reachable subgraph REACH(START). Whenever we use the phrase "a flowgraph and its DJ graph," we are referring to " the reachable subgraph REACH(START) and its DJ graph."
>
> A DJ graph consists of a dominator tree and join edges. Therefore, whenever we use the term dominator tree, we also mean a DJ graph without J edges.
>
> Since a flowgraph and its DJ graph contain the same set of nodes, whenever we use the term *node* x, the node x may belong to either the flowgraph or the DJ graph, unless explicitly specified by using subscript notation with f for flowgraph and d for DJ graph. When we describe a graph property without using subscripts, the property holds on both a flowgraph and its DJ graph.
>
> Finally, the iterated dominance frontier of a node y is the same when computed on either the DJ graph or the flowgraph. Thus, all of $IDF(y)$, $IDF(y_f)$ and $IDF(y_d)$ will mean the same thing.

A.2 Data Flow Analysis

Data flow analysis is a process of estimating facts about a program statically. These facts, or data flow information, can be modeled by elements of a lattice $\mathcal{L}$. Associated with each node x is a flow function f_x that maps input information to output information [Kil73, Mar89].[6]

Let $I_x \in \mathcal{L}$ be the information at the entry of a node x, and let $O_x \in \mathcal{L}$ be the information at the exit of the node. Then the input-output relation can be expressed as

$$O_x = f_x(I_x)$$

[6] For some problems, it is more convenient to associate flow functions with flowgraph edges instead of nodes.

We can rewrite this equation as follows for forward union problems:[7]

$$O_x = f_x(I_x) = I_x \cap P_x \cup G_x \tag{4}$$

where $P_x, G_x \in \mathcal{L}$.[8] We can interpret the above equation as follows: The *O*utput flow information at a node's exit is either (1) what is *G*enerated within the node or (2) what arrives at its *I*nput and is *P*reserved through the node. For convenience, we will use the following notation instead of the notation in Equation (4).

$$O_x = f_x(I_x) = P_x I_x + G_x \tag{5}$$

where + is the union operation and juxtaposition is the intersection operation.

We need another set of equations to relate the output information at a node y to the input information at x when an edge $y \to x$ exists. The input information I_x is the merge of all the output information O_y of nodes y in $Pred_f(x)$; i.e.,

$$I_x = \bigwedge_{y \in Pred_f(x)} O_y \tag{6}$$

$\bigwedge$ is usually a union or intersection operation, depending upon the data flow problem being solved. Combining Equation (5) and 6, we obtain the following equation, denoted H_x, for each $x \in N$.

$$\begin{aligned} H_x : O_x &= f_x(\bigwedge_{y \in Pred_f(x)} O_y) & (7) \\ H_x : O_x &= P_x(\bigwedge_{y \in Pred_f(x)} O_y) + G_x & (8) \end{aligned}$$

Since there is one equation for each node, we have a total of $|N|$ equations. Notice that the above equation has two variations. The first variation (Equation (7)) is more general than the second (Equation (8)), since for some data flow problems the information generated within a node x is not independent of I_x [Mar89]. We use the term **output variable** to name the variable O_x that appears on the left-hand side (LHS) of equation H_x. Any variable appearing on the right-hand side (RHS) of the equation is called an **input variable**. Furthermore, P_x and G_x are called the **parameters** of the equation.

Given any two equations H_x and H_y, we say that H_x **depends on** H_y if the output variable of H_y appears on the RHS of H_x. That is, we will need the solution of O_y in order to

[7] Forward intersection problems have a dual formulation, and backward problems can be solved on the "reverse" flowgraph.

[8] In general G_x may not be a constant [Mar89]. It can also depend on I_x. That is, what information is generated at a node depends on (1) the local data flow information at the node and (2) the input data flow information to the node.

compute the solution of O_x. Also, if H_x depends on H_y then there is an edge from y to x in the corresponding flowgraph.

We will define the closure operation for recursive equations. If at any node y, the data flow equation is of the form

$$H_y : O_y \quad = \quad mO_y + k = f(O_y), \tag{9}$$

where m and k are terms that do not contain O_y, then the closure operation of this equation is

$$H_y : O_y \quad = \quad f^*(O_y) \tag{10}$$

In other words, O_y is a fixed point of the above recursive equation. Such a fixed point exists if the flow function associated with each node is monotone, and the lattice does not contain infinite descending chains [Mar89, KU76, Kil73].

For many of the classical problems, such as Reaching Definitions and Available Expressions, f^* can be computed very fast, essentially in constant time [Mar89]. Ryder and Paull call such closure operations the loop-breaking rules [RP86]. They show how to compute f^* very fast for the classical bit-vector union and intersection problems. Marlowe gives an in-depth treatment on the computational complexity for various classes of functions and domains that can occur in data flow analysis [Mar89]. Throughout our discussion we will assume a monotone data flow framework in which all flow functions are monotone; i.e., for any flow information α and β, $\alpha < \beta \Rightarrow f(\alpha) < f(\beta)$ for any flow function f [Mar89, KU76, Kil73].