McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

Efficient Data Flow Analysis Using DJ Graphs: Elimination Methods Revisited

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

ACAPS Technical Memo 93
June 21, 1995

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

A version of this report has been submitted for publication

Abstract

In this paper we present a new approach to elimination based data flow analysis that uses a program representation called the **DJ Graph**. The skeleton of the DJ graph of a program is the dominator tree of its flowgraph (whose edges are called D edges in this paper), and the tree skeleton is augmented with *join edges* (called J edges in this paper). Unlike the previous elimination methods, which first reduce a flowgraph to a single node, our approach only eliminate J edges from the DJ graph in a bottom-up fashion during the reduction process, while maintainting the dominator tree structure (which may be compressed).

We propose two methods for eliminating variables: (1) **eager elimination method**, and (2) **delayed elimination method**. With eager elimination, we first perform variable elimination on the DJ-graph in a bottom-up manner. Once we determine the solution for the root node, we propagate this information in a top-down fashion on the dominator tree and determine the corresponding solution for every other node. Although our eager elimination method exhibits the same worst case quadratic behavior as the Allan-Cocke method, it provides an attractive framework for implementation due to its simplicity. Furthermore, our approach also can handle irreducible graphs.

Based on the framework, we propose a simple extension called delayed elimination method. It avoids some redundant processing by delaying variable eliminations and by performing path compressions on the dominator tree. This method can achieve $O(|E|log(|N|))$ function operations for a reducible flowgraph, where $|E|$ is the number of edges and $|N|$ is the number of nodes in the flowgraph.

To profile the performance of our approach on real programs, we have implemented (1) the eager elimination method, (2) the delayed elimination method, and (3) the iterative method (using a depth-first ordering for iteration) for solving the intraprocedural reaching definitions problem. We used 40 FORTRAN procedures in our experiments. 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 procedures. In addition, the eager elimination method performed almost as well as the delayed elimination method in these experiments.

Contents

1 Introduction **1**

2 Our Approach **3**

3 Eager Elimination Method **5**

3.1 The $\mathcal{E}$-*rules* 6

3.2 Algorithm Description 7

3.3 Bottom-Up Variable Elimination 12

3.4 An Example 14

3.5 Top-Down Propagation 15

4 Correctness and Complexity of the Eager Elimination Method **15**

4.1 Correctness 16

4.2 Complexity 20

4.3 Discussion 21

5 Delayed Elimination Method **22**

5.1 Dominance Frontier Interval 22

5.2 Derived Edges 23

5.3 Worst-Case Quadratic Complexity of Eager Elimination Method 23

5.4 $\mathcal{E}$-*rules* Revisited: The $\mathcal{D}$-*rules* 24

5.5 Delaying Variable Elimination 25

5.6 Top-Down Propagation 29

5.7 An Example 30

6 Correctness and Complexity of Delayed Elimination Method **32**

6.1 Correctness 32

6.2 Complexity 35

6.3 Discussion 37

7 Handling Irreducibility **37**

8 Experiments and Empirical Results **41**

8.1 Structural Characteristics 43

8.2 Execution Performance 47

9 Discussion and Related Work **50**

10 Conclusion **53**

A Background and Notation **56**
A.1 Flowgraph Terminology 57
A.2 Data Flow Analysis 57

B A Linear Algorithm for Computing Top Nodes **60**

List of Figures

1 An example of a flowgraph, its dominator, and its DJ graph. 4
2 A graphical illustration of $\mathcal{E}$-*rules*. 7
3 A trace of the DJ graph reduction using $\mathcal{E}$-*rules*. 13
4 DJ graph annotated with **cTop** nodes. 23
5 A trace of the DJ graph reduction using $\mathcal{D}$-*rules*. 27
6 An irreducible flowgraph and its DJ graph. 38
7 A trace of DJ graph reduction for the irreducible flowgraph. 40
8 A profile of the number of $\mathcal{E}$-*rules* applied. 45
9 A profile of the number of $\mathcal{D}$-*rules* applied. 46
10 A comparison of $|DF_e|$ with C and C'. 48
11 Execution profile of the three algorithms when executed on our test suites. 49

List of Tables

1 Structural characteristics. 44
2 Timings and speedups. 51
3 Initial data flow equation at each node. 60

1 Introduction

Elimination-based data flow analysis have been studied by many authors [ASU86, AC76, Ull73, Hec77, GW76, Tar81, Bur90, Ros80, Ros82, SS79]. An excellent survey can be found in [RP86]. Traditional elimination-based data flow analysis techniques 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. In this paper we present our approach for exhaustive elimination-based data flow analysis that uses DJ graphs as its main data structure. We propose two variations of our approach: (1) **eager elimination method**, whose worst case time complexity is $O(|E| \times |N|)$, where $|N|$ and $|E|$ are nodes and edges in the flowgraph, respectively; and (2) **delayed elimination method**, whose worst case time complexity is $O(|E| \times \log(|N|))$.[1]

Our work is related to the four classical elimination methods (Allen-Cocke, Hecht-Ullman, Graham-Wegman, and Tarjan), but with a number of significant differences. The Hecht-Ullman method uses a forest of height-balanced 2-3 tree to remember the common substitution sequences. Merging and balancing the forest of 2-3 trees complicates this method, but it can help achieve the $O(|E| \times \log(|N|))$ time complexity. Tarjan proposes two variations of his approach: a simple $O(|E| \times \log(|N|))$ algorithm that uses path compression tree (for remembering the common substitution sequences), and a $O(|E| \times \alpha(|E|, |N|))$ algorithm that also balances the path compression tree (where $\alpha()$ is the inverse Ackermann's function). Although his balanced path-compression tree method is almost linear, Tarjan favors the simple $O(|E| \times \log(|N|))$ algorithm for practical implementation [Tar81]. In the Graham-Wegman method, the common substitution sequences are remembered explicitly in the (reduced) flowgraph rather than an auxiliary data structure. This method also uses a form of path compression, but it is more complicated than Tarjan's simple algorithm [GW76, Tar81]. Its time complexity is again $O(|E| \times \log(|N|)$.

In our approach we do not collapse a region into a node. Instead we maintain the dominator tree (which may be compressed) of the flowgraph. One unique feature of our approach is that graph reduction and variable substitution (or elimination) are performed in a bottom-up fashion on the nodes in the DJ graph. Rather than reducing a DJ graph to a single node, we only eliminate J edges from the DJ graph, and in the process we also perform variable substitution along D edges when necessary, in either an eager or a delayed fashion. At the end of the bottom-up elimination phase, all the J edges will be eliminated. Meanwhile the equation at every node is expressed only in terms of its parent node in the (maybe compressed) dominator tree. Once we determine the solution for the root node, we propagate this information in a top-down fashion on the (maybe compressed) dominator tree to compute the solution for every other node.

[1] Here we assume fast data flow problems when discussing complexity, although our approach is applicable to more general monotone data flow problems [Bur90, Tar81, Ros80, Ros82].

To achieve both efficiency and simplicity in our two methods, we exploited several key concepts.

- We neither use any auxiliary data structures nor group nodes to form regions. Instead, we manipulate (and delay) all the variable eliminations explicitly on the DJ graph itself.
- In the delayed elimination method, we use a linear time pre-processing step to identify the exact level at which each delayed variable should be eliminated. This can help avoid redundant inspection of J edges during the reduction process to determine whether to eliminate a delayed variable or not. In our pre-processing step we use structural properties of DJ graphs to identify the levels.
- In the delayed elimination method, we perform simple path compressions on the dominator tree (without balancing the tree) to evaluate and eliminate delayed variables. Our path compression is similar to Tarjan's simple path compression, but performed on a static dominator tree.
- Our approach does not require a 'parse' [HU72, Ull73, Tar74, Tar81] or 'S-set finding' [GW76] to determine the order in which J edges will be eliminated. The J edges are eliminated from the DJ graph in a bottom-up manner.
- Another interesting feature of our approach is its relation to the concept of dominance frontier and iterated dominance frontier. We will exploit this relationship to establish that our elimination approach (both eager and delayed) should behave linearly in practice.
- Finally, another interesting feature of our approach is that it can easily identify and handle *irreducibility* gracefully in the bottom-up reduction process. When irreducible loops exist, the worst case cost of our elimination methods can be as good as (or as poor as) that of the iterative method. This happens when the root node is the immediate dominator of all the other nodes, and these nodes belong to the same irreducible region. We believe this to be extremely unusual in practice. Our approach to handling irreducibility does not perform fixed-point iteration over all the nodes in an irreducible region. Instead, we apply our elimination method to every reducible region contained in an irreducible region, and perform iteration only over nodes within the irreducible region that are at the same level (of the dominator tree).

To study the effectiveness of our approach, we have implemented both the eager and the delayed method for intraprocedural reaching definitions. Our implementation was built upon the Parafrase2 compiler [PGH+91]. To compare the results, we also implemented the iterative method for reaching definitions that uses reverse postorder of nodes for iteration [HU77]. Both the eager and the delayed method perform better than the iteration method (on average we get a speedup of 1.3 compared to the iteration method). Although, theoretically, our delayed

elimination method is superior to the eager elimination method, we observed that for many procedures the eager method ran faster than the delayed method did. As we will demonstrate in this paper, the time complexity of our eager elimination method is directly related to the size of the dominance frontiers, suggesting that the time complexity of the eager method should be linear in practice (since the size of the dominance frontiers is linear in practice [CFR+91]). For those cases where the eager method performed better than the delayed method, we suspect that the overhead of path compressions may actually overshadow the benefit of delaying variable eliminations. This can happen if there are not many "overlapping paths" to take advantage of delayed variable eliminations. As we will also demonstrate in this paper, the number of overlapping paths is again related to the size of dominance frontiers; the flowgraphs of real programs do not contain many overlapping paths. Based on our observations, we recommend the eager elimination method for practical implementation. Also, our eager method is amenable to incremental data flow analysis, the details of which is presented in our companion paper [SG95a].

2 Our Approach

In the next several sections we present our approach for elimination-based data flow analysis. In this section we highlight several features of our approach. First, we use the DJ graph of a flowgraph rather than the flowgraph itself for performing data flow analysis.

First, we use the DJ graph of a flowgraph rather than the flowgraph itself in performing data flow analysis. Furthermore, we reduce a DJ graph into its dominator tree instead of a single node. A DJ graph is nothing but the dominator tree of a flowgraph augmented with join edges (J edges) [Sre95, SG95b, SGL95]. An edge $x \rightarrow y$ in a flowgraph is a join edge if x does not immediately dominate y. Notice that if x immediately dominates y (denoted by $x = idom(y)$) then $x \rightarrow y$ is an edge in the dominator tree.[2] With every node x in the DJ graph we can associate a level number, denoted by $x.level$, that is the depth of the node from the dominator tree root. Figure 1(b) and (c) give the dominator tree and the DJ graph, respectively, of an example flowgraph shown in Figure 1(a). Given the level information, we can observe that if $x \rightarrow y$ is a J edge, then $x.level \geq y.level$. Also, if $x \rightarrow y$ is a J edge, then $x \rightarrow y$ is in the dominance frontiers of all nodes that dominate x but do not strictly dominate y.

We propose two methods for reducing a DJ graph and solving the corresponding system of data flow equations. Both methods perform the following actions:

1. Reduce the DJ graph in a bottom-up fashion by eliminating J edges.

2. Use $\mathcal{E}$-$rules$ with eager variable elimination, or use $\mathcal{D}$-$rules$ with delayed variable elimination for reducing the system of data flow equations.

[2] Appendix A gives the necessary background materials and standard definitions relevant to our discussion.

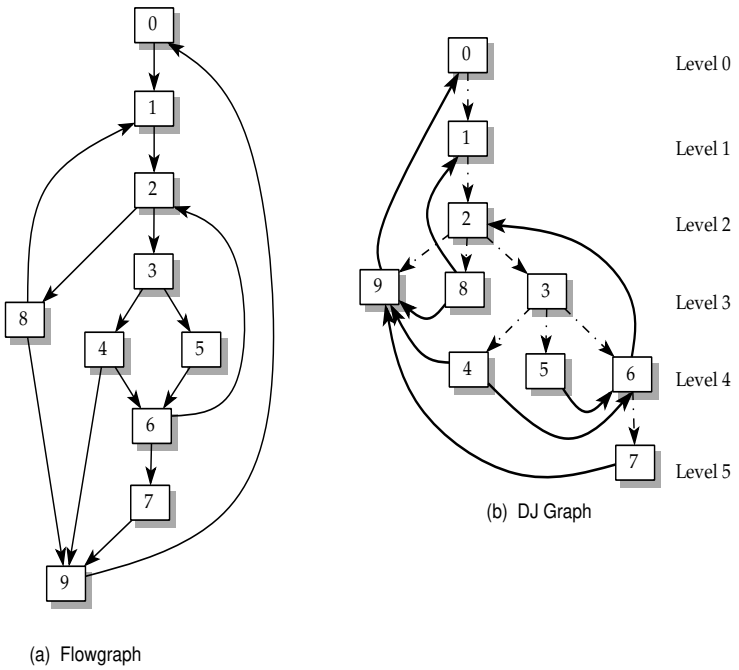

Figure 1: An example of a flowgraph, its dominator, and its DJ graph.

3. Propagate the final data flow solutions in a top-down manner on the dominator tree (that may be compressed).

The bottom-up graph reduction is important because it enables us to visit nodes such a way that when a node y is being processed, all J edges originating at a level greater than $y.level$ have been eliminated. Therefore we can systematically reduce a DJ graph to its dominator tree by applying the reduction rules in a bottom-up fashion. Furthermore, the bottom-up reduction simplifies the implementation of our algorithm.

Our **eager elimination method** uses $\mathcal{E}$-$rules$ and eagerly eliminates variables by substitution. An important aspect of this method is that after we reduce a DJ graph to its dominator tree, the data flow equation at every node (except for the root node) depends only on the output variable of its immediate dominator. Consequently, once we obtain the data flow solution at the root node, we can compute the solution for any other node in a top-down manner. As we will demonstrate in this paper, eager elimination method is very simple and easy to implement.

The main drawback with eager elimination is that theoretically it exhibits a worst-case quadratic time complexity. To improve this we modify $\mathcal{E}$-$rules$ and propose $\mathcal{D}$-$rules$. Using $\mathcal{D}$-$rules$ with delayed variable elimination, called the **delayed elimination method**, we can

achieve $O(|E| \times \log(|N|))$ time complexity. Recall that all three other previous elimination methods improve over the quadratic Allen-Cocke method by *delaying* substitution of variables [RP86]. They perform this either explicitly as in the Graham-Wegman method, or implicitly by using auxiliary data structures to keep track of the delayed variables (height-balanced 2-3 trees in the Hecht-Ullman method [Ull73] or the balanced, binary path compressed tree in Tarjan's method [Tar81]). The time complexity of all three methods is $O(|E| \times \log(|N|))$, but Tarjan also gave a more complex and almost linear method (of the complexity $O(|E| \times \alpha(|N|, |E|))$ that uses balanced compression trees [Tar81].

Our delayed elimination method also delays variable substitution. It performs variable substitutions along a path on the (maybe compressed) dominator tree during the bottom-up graph reduction in order to eliminate "delayed variables." The path is compressed when delayed variable substitutions take place.

An important feature of our approach (including both eager and delayed elimination) is that during the bottom-up reduction we can easily detect and handle irreducibility in an efficient manner. There is irreducibility whenever we cannot eliminate J edges at some level using our reduction rules. In this situation we apply Tarjan's Strongly Connected Components algorithm only over nodes at that level. Every nontrivial strong component (with more than one node) represents an irreducible region. Strong components are visited in topological order. When an irreducible region is processed, fixed-point iteration is performed over its nodes in order to eliminate interdependencies among the solutions at these nodes. Consequently, we are able to represent the solution at each node only in terms of the solution at its parent.

3 Eager Elimination Method

Our eager elimination method consists of three parts: (1) bottom-up reduction of the DJ graph, (2) variable elimination, and (3) top-down propagation. The first two are performed together in the **elimination phase**. In the **propagation phase**, data flow information is propagated in a top-down manner on the dominator tree after the solution of the root node is determined.

In this section we propose two rules, E1 and E2 rules (together called the $\mathcal{E}$-*rules*), for reducing a DJ graph in a bottom-up fashion. 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 $y.level$. Technically, a node in flowgraph is a non-join node if it has only one predecessor (which strictly dominates it). In this paper we will adopt the following relaxed definition:

Definition 3.1 (non-join node) *A node y is a* **non-join** *node iff $Pred_f(y)$ contains no other nodes than y and $idom(y)$.*

That is, a self-loop $y \rightarrow y$ will not prevent y from being a non-join node.

In the next subsection we formally define the two $\mathcal{E}$-*rules*. Each application of $\mathcal{E}$-*rules* transforms some reduced DJ graph $\mathcal{G}^i$ to $\mathcal{G}^{i+1}$ until the resulting DJ graph "degenerates" into its dominator tree. We first focus only on the graph reduction, and we will discuss variable elimination in Section 3.3.

3.1 The $\mathcal{E}$-*rules*

The first of our two $\mathcal{E}$-*rules* is the E1 rule, which eliminates a self-loop.

Definition 3.2 (E1 rule) *Let $\mathcal{G}^i = (N, E)$ be the ith reduced DJ graph. Let y be a non-join node. If y contains a self-loop, i.e., if there is a J edge $y \rightarrow y$, then we apply the E1 rule*

$$E1(< \mathcal{G}^i, N, E, y \xrightarrow{J} y >) \; = < \mathcal{G}^{(i+1)}, N, E - \{y \xrightarrow{J} y\} > .$$

An E2 rule is applied to a J edge $y \rightarrow z$, if y is a non-join node and it does not contain a self-loop. We distinguish between two types of E2 rules depending on the levels of y and z. If $y.level = z.level$, then we apply the E2a rule; otherwise we apply the E2b rule.

Definition 3.3 (E2 rules) *Let $\mathcal{G}^i = (N, E)$ be the ith reduced DJ graph. Let y be a non-join node, let $y \rightarrow z$ be a J edge, and let $x = idom(y)$. There are two cases:*

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

$$E2a(< \mathcal{G}^i, N, E, y \xrightarrow{J} z >) \; = < \mathcal{G}^{i+1}, N, E - \{y \xrightarrow{J} z\} > .$$

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

$$E2b(< \mathcal{G}^i, N, E, y \xrightarrow{J} z >) \; = < \mathcal{G}^{i+1}, N, (E - \{y \xrightarrow{J} z\}) \cup \{x \xrightarrow{J} z\} > .^3$$

Figure 2 graphically illustrates the two $\mathcal{E}$-*rules* (where the dotted line from w to x represents the dominator tree path from w to x). An important point to note here is that when an E2 rule is applied to an edge $y \rightarrow z$, we should ensure that the self-loop at node y, if any, is eliminated first by applying the E1 rule. The distinction between the two E2 rules is minor. An E2a rule is applied to an edge $y \rightarrow z$ only if $y.level = z.level$; otherwise we apply an E2b rule. This distinction is useful when we discuss delayed elimination method and in handling irreducible flowgraphs.

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

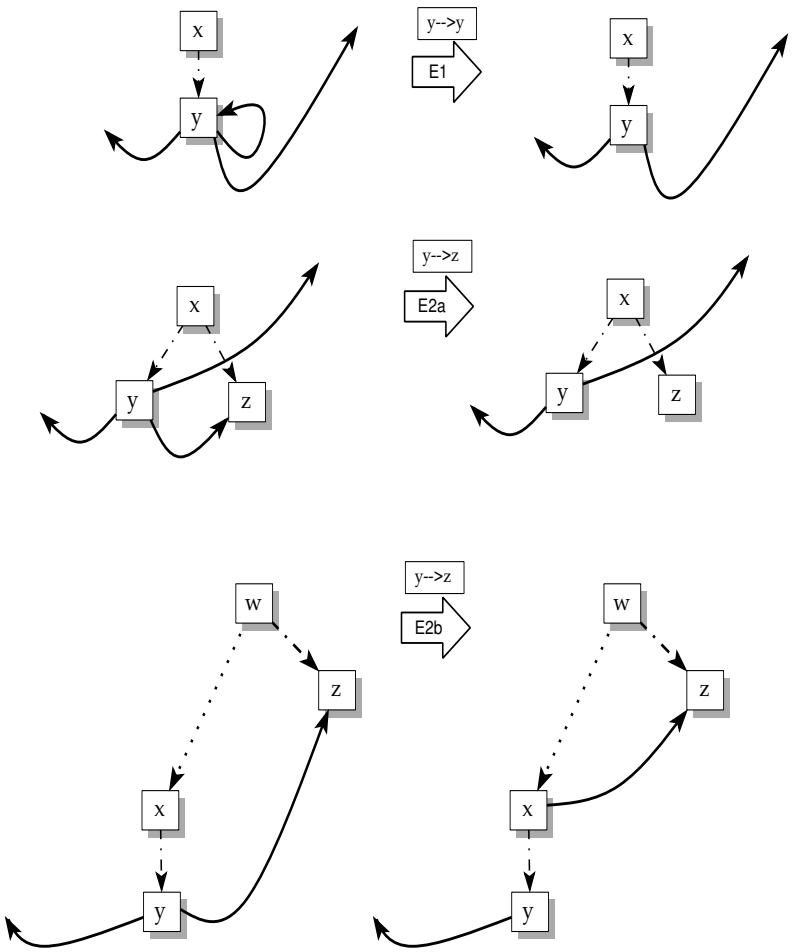

Figure 2: A graphical illustration of $\mathcal{E}$-*rules*.

3.2 Algorithm Description

The complete algorithm for DJ graph reduction is given below. To simplify its presentation we use the following notation and data structures:

- $NumLevel$ is the total number of levels in the DJ graph.
- Each edge $x \rightarrow y \in E$ has an attribute that specifies its type: $\{Dedge, Jedge\}$.
- Each node $x \in N$ has the following attributes:

```
struct NodeStructure {
   int indegree ; /* indegree of the node.  */
```

```
    int level ; /* level number of the node */
}
```

At each node we also maintain a linked list of outgoing edges such that the self-loop edge at this node, if any, will be the first edge in the list.

- *PriorityList* is an array of doubly linked list of nodes. First we will define the linked list structure.

```
struct ListStructure{
    struct NodeStructure *node ; /* pointer to node structure */
    struct ListStructure *next ; /* next node in the list  */
    struct ListStructure *prev ; /* prev node in the list  */
}
```

The structure of the *PriorityList* is defined as follows:

```
struct OrderedBucketStructure {
struct ListStructure *head ; /* nodes at the same level */
} OrderedBucket[NumLevel] ; /* array of list of nodes  */
```

We restrict the way in which nodes can appear in the *PriorityList*$[i]$, at each index i. At all times we ensure that non-join nodes, if any, appear at the head end of the list.

Algorithm 3.1 (Eager Elimination Method) *The algorithm* **MainElim()** *with* $\mathcal{E}$*-rules and eager elimination can be used for solving a system of data flow equations.*

The first step in the algorithm is to insert all the nodes into the *PriorityList* at their respective levels. Then we call **ReduceLevel**() in a bottom-up fashion. For reducible flowgraphs, the condition at step **3** will fail. In other words, after the call to **ReduceLevel**() terminates, for reducible flowgraphs, the list *PriorityList*$[i]$, for the current level i, will be empty (step **3**); otherwise, the flowgraph is irreducible. For an irreducible flowgraph we call **CollapseIrreducible**() to handle the irreducible portion. A complete description of how to handle the irreducible portion is given in Section 7. The procedure **DomTDPropagate()** is for propagating solutions down the dominator tree, and is explained in Section 3.5.

♣ **Input:** A DJ graph $\mathcal{G}^0$ and the corresponding system of initial flow equations.

♣ **Output:** Solution to a system of data flow equations.

♣ **Initialization:**

• Determine the level number of each node x, and stores its level information in $x.level$.

• For all nodes $x \in \mathcal{G}^0$, $x.indegree$ is initialized to the number of predecessor nodes of x (in the DJ graph). If ($x.indegree = 1$) **or** (($x.indegree = 2$) **and** ($x \rightarrow x$)) then x is a non-join node.

• Initially deposit all nodes in the *PriorityList*, such that at each index non-join nodes appear at the head of the list (i.e., before join nodes).

```
MainElim()
{

1:      for i = NumLevel − 1 downto 1 do /* bottom-up reduction */
2:        ReduceLevel(i) ;
3:        if(PriorityList[i].head ≠ NULL) then
4:          CollapseIrreducible(i) ;
5:        endif
6:      endfor
7:      DomTDPropagate() ;

}
```

The procedure **ReduceLevel**(i) eliminates J edges whose source nodes are at level i by applying the $\mathcal{E}$-*rules*. We apply either E1 or E2 for each outgoing J edge. Notice that we do not process outgoing D edges (step **10**). At step **11** we check if the non-join node y has a self-loop. If so we apply the E1 rule; otherwise we apply an E2 rule. At step **15** we apply the E2a rule if $idom(y) = idom(z)$; otherwise we apply the E2b rule (step **17**). The order in which we apply the rules ensures that there is no self-loop at node y when E2 rules are applied to outgoing edges from y.

```
Procedure ReduceLevel(i)
{

8:      while((y = GetNode(i)) ≠ NULL) do
9:        foreach z ∈ Succ(y) do
            /* if y → z is a self-loop edge then it is first  */
            /* in the list of edges at node y */
10:       if(y → z == Jedge) then
11:         if(z == y) then /* self-loop */
12:           Eager1(y → y) ; /* apply E1 rule */
13:         else
14:           if(idom(y) == idom(z)) then
15:             Eager2a(y → z) ;
16:           else
```

17: **Eager2b**$(y \rightarrow z)$;
18: **endif**
19: **endif**
20: **endif**
21: **endfor**
22: **endwhile**

}

The function **GetNode**(i) returns either a non-join node (if one exists at the current level i) (step **27**) or $NULL$ (step **24** or step **29**). When **GetNode**(i) returns $NULL$ from step **29**, we encounter irreducibility. This is because there are still nodes in *PriorityList*$[i]$ at level i, but none of them are non-join nodes. In Section 4 we will prove why this condition is sufficient to detect irreducibility.

Function GetNode(i)
{

23: **if**$(PriorityList[i].head == NULL)$ **then**
24: **return** $NULL$;
25: **endif**
26: **if** the first node at *PriorityList*$[i]$.*head* is non-join node **then**
27: Remove that node from the list and **return** it.
28: **endif**
29: **return** $NULL$;
30: **endif**

}

The procedures for the two $\mathcal{E}$-*rules* are given below. Each call to the procedure **Eager1**$(y \rightarrow y)$, deletes the self-loop $y \rightarrow y$ (back) edge. The operation within $[\![\ldots]\!]$ is for variable elimination and will be explained in the next section.

Procedure Eager1$(y \rightarrow y)$
{

31: $[\![$Compute the closure $H_y : O_y = f^*(O_y)$ $]\!]$
32: $z.indegree = z.indegree - 1$;
33: Delete the edge $y \stackrel{J}{\rightarrow} y$;

}

The procedures **Eager2a**$(y \rightarrow z)$ and **Eager2b**$(y \rightarrow z)$ implement E2a and E2b rules, respectively. Again, the operation within $[\![\ldots]\!]$ is for variable elimination and will be explained

in the next section. The deletion of $y \rightarrow z$ can make z become a non-join node; if so we put z at the head of the $PriorityList[z.level]$ list (step **38**). We do this to ensure constant time operation for the procedure **GetNode**().

Procedure Eager2a($y \rightarrow z$)
{

34: [[Eliminate O_y in H_z by replacing it with the RHS of H_y.]]
35: Delete the edge $y \xrightarrow{J} z$;
36: $z.indegree = z.indegree - 1$;
37: **if**(z is a non-join node) **then** `/* z becomes a non-join node */`
38: Put z at the head of $PriorityList[z.level]$ list ;
39: **endif**

}

Unlike the procedure **Eager2a**(), the procedure **Eager2b**() decreases $z.indegree$ by one only if the edge $x \rightarrow z$ already exists. Notice the difference between E2a and E2b rule. In E2a rule we never introduce a new edge from $x \rightarrow z$, since a D edge already exists from x to z.

Procedure Eager2b($y \rightarrow z$)
{

40: [[Eliminate O_y in H_z by replacing it with the RHS of H_y.]]
41: $x = idom(y)$;
42: Delete the edge $y \xrightarrow{J} z$;
43: **if**($x \rightarrow z$ exists) **then**
44: $z.indegree = z.indegree - 1$;
45: **else**
46: Insert a new J edge $x \xrightarrow{J} z$;
`/* If` $x \rightarrow z$ `is a self-loop edge we ensure that this */`
`/* is the first edge in the list of edges at node` x`. */`
47: **endif**

}

Next we will intuitively describe why the above algorithm always reduces a DJ graph to its dominator tree (remember that we are assuming reducible flowgraphs for the moment). Later, in Section 4, we will give a formal proof for this. The reason is based on the following key property. Let $\mathcal{G}$ be a DJ graph and let k be a level in the dominator tree such that there are no J edges in $\mathcal{G}$ whose source nodes are at levels greater than k. If $\mathcal{G}$ is reducible then there exists at least one non-join node at level k. Therefore if we eliminate J edges in a bottom-up

fashion then we can always find at least one candidate non-join node for applying the $\mathcal{E}$-*rules* rules.

In addition, we will intuitively argue the time complexity of the reduction process. Notice that each application of E1 and E2a rules eliminates one edge. Since there are at most $|E|$ edges in the DJ graph. We will apply E1 and E2a rules at most $|E|$ times. But how many times the E2b rule can be applied? A naive argument shows that we may be applying this rule for $O(|E| \times |N|)$ times. We will give a formal proof in Section 4

To illustrate our reduction method, consider the example flowgraph in Figure 1. Figure 3 gives a trace of the reduction process. In Section 3.4 we will give a detailed explanation of the trace, after we discuss variable elimination.

3.3 Bottom-Up Variable Elimination

In this section we will show how to eagerly eliminate variables from data flow equations for each application of the $\mathcal{E}$-*rules*. Prior to applying the E1 rule to a node y, the flow equation at y will resemble

$$H_y : O_y \quad = \quad kO_y + m, \tag{1}$$

where k and m are terms or constants that do not contain the variable O_y. Recall that this recursive equation can be solved with fixed-point iteration, giving rise to a closure operation. After E1 rule is applied, the equation for node y would become:

$$H_y : O_y \quad = \quad f^*(O_y). \tag{2}$$

In the above closure operation, $f^*()$ does not contain the variable O_y. Ryder and Paull call this operation as the loop-breaking rule. In the procedure **Eager1**$(y \rightarrow y)$, the variable elimination is done at step **31**.

Let $x = idom(y)$. Prior to applying an E2 rule to a J edge $y \rightarrow z$, the data flow equation at z will resemble:

$$H_z : O_z \quad = \quad kO_y + m,$$

where the terms k and m do not contain the variable O_y. Supposing the flow equation at y is

$$H_y : O_y \quad = \quad aO_x + b$$

where a and b are terms that do not contain any O_x (or any other variable). After applying E2 rule to $y \rightarrow z$ we eliminate O_y on the RHS of H_z by replacing it with the RHS of H_y. The resulting equation of H_z is

$$H_z : O_z = k(aO_x + b) + m.$$

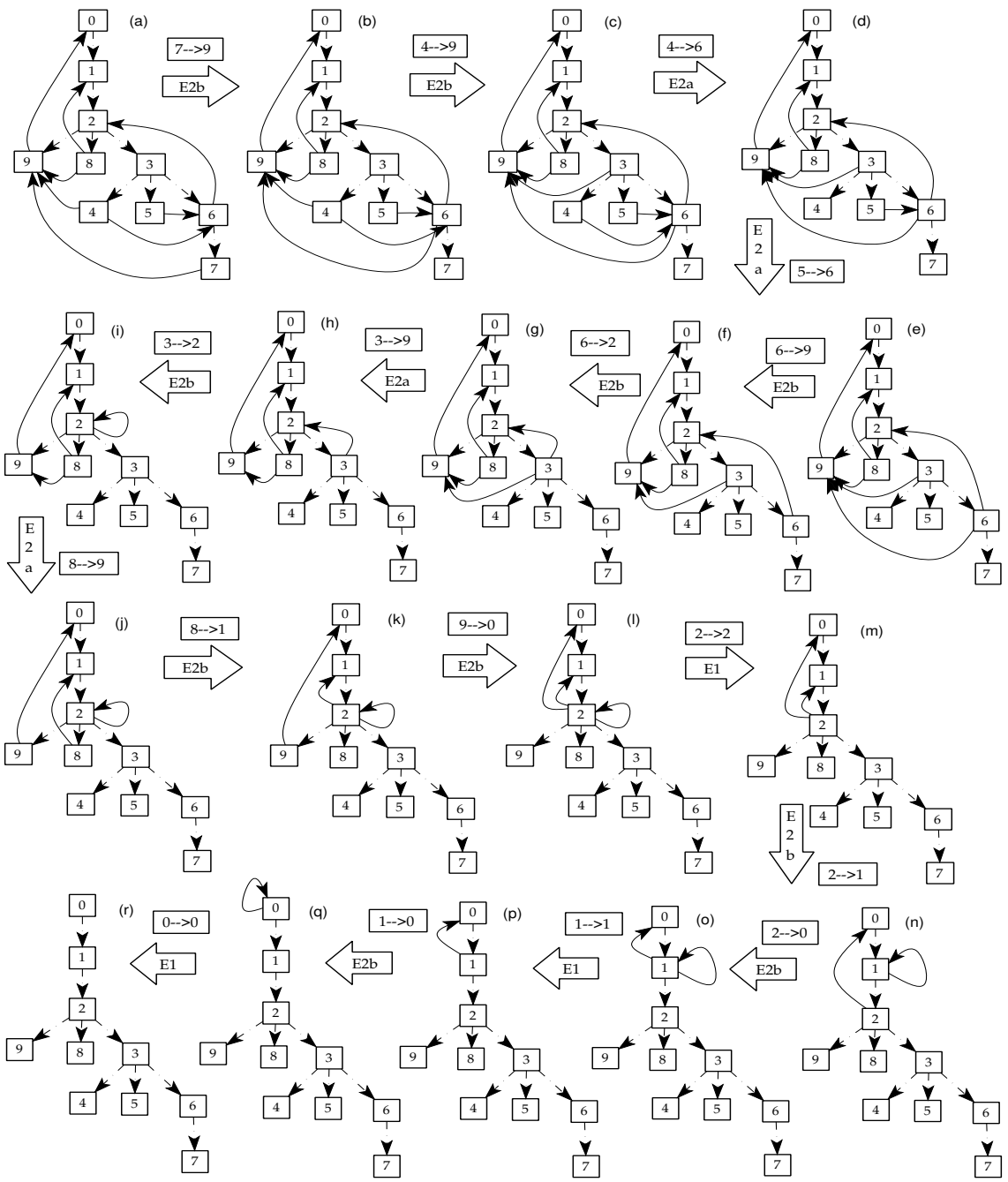

Figure 3: A trace of the DJ graph reduction using $\mathcal{E}$-*rules*.

In the procedure **Eager2a**($y \rightarrow z$) (**Eager2b**($y \rightarrow z$)), the variable elimination is done at step **34** (step **40**). Notice that whenever we eliminate a variable O_y in H_z, we also delete the J edge $y \rightarrow z$ in the corresponding DJ graph reduction, and vice-versa. In the E2a or the E2b rule, to eliminate O_y in H_z we substitute the RHS of H_y in H_z, this introduces a new variable dependence of O_z on O_x. We also ensure that an edge exists from node x to node z. In E1 rule, eliminating O_y in H_y is done by fixed-point computation, and we correspondingly remove the self-loop edge $y \rightarrow y$.

In the eager elimination method, we eagerly replace every occurrence of variable O_y in node z's flow equation during $\mathcal{E}$-*rules*. This could lead to poor performance for deeply nested loops [RP86]. In Section 5 we will show how to delay certain variable elimination so that we can speed up the overall algorithm.

3.4 An Example

Let us assume a forward data flow problem, with union as the meet operation. For such problems, the table below gives a partial trace of the variable elimination corresponding to the trace of DJ 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

1. Recall that the DJ graph reduction and variable elimination are performed in a bottom-up manner (the **for** loop at step **1**).

2. At step **8** **GetNode**(5) would return node 7, and we apply E2b rule to the outgoing J edge $7 \rightarrow 9$ by invoking **Eager2b**($7 \rightarrow 9$) (step **17**). This transforms the DJ graph shown in Figure 3(a) to the DJ graph shown in Figure 3(b). This transformation also eliminates O_7 in H_9, and the new equation at node 9 is

$$H_9 : O_9 = P_9O_4 + P_9O_8 + P_9P_7O_6 + P_9G_7 + G_9$$

3. When **GetNode**(4) is invoked at step **8**, there are two non-join nodes, node 4 and node 5, at level 4. Assume that **GetNode**(4) returns node 4, and we apply E2b rule to the outgoing J edge $4 \rightarrow 9$ by invoking **Eager2b**($7 \rightarrow 9$) (step **17**). This

transforms the DJ graph shown in Figure 3(b) to the DJ graph shown in Figure 3(c). The corresponding equation at node 9 is transformed to

$$H_9 : O_9 = P_9P_4O_3 + P_9G_4 + P_9O_8 + P_9P_7O_6 + P_9G_7 + G_9$$

4. Next **GetNode**(4) would return node 5, and we invoke $rulea5 \rightarrow 6$. This transforms the DJ graph shown in Figure 3(c) to Figure 3(d). The corresponding equation of node 6 would be transformed to

$$H_6 : O_6 = P_6P_4O_3 + P_6G_4 + P_6O_5 + G_6$$

5. We can continue to eliminate variables as described above at other nodes. We encourage interested readers to do so.

3.5 Top-Down Propagation

When the **for** loop in the **MainElim()** procedure terminates, the original DJ graph would be transformed to its dominator tree. At this point the data flow solution at each node depends only on the solution of its immediate dominator, that is, the only variable that can appear on the RHS of equation H_y is $O_{idom(y)}$. Consequently, the call to **DomTDPropagate()** at step **7** will complete the data flow solution process. In the procedure **DomTDPropagate()**, we first solve the equation at the root node of the dominator tree; we then propagate the solution to all other nodes in a top-down fashion as given below.

DomTDPropagate()
{

48: Solve the data flow equation of the START node.
49: Propagate the solution on the dominator tree in a top-down fashion,
using the solution at node $x = idom(y)$ to substitute variable O_x
on the RHS of equation $O_y = f(O_x)$, to compute the solution at y.

}

4 Correctness and Complexity of the Eager Elimination Method

In this section we prove the correctness of Algorithm 3.1 and analyze it time complexity.

4.1 Correctness

The main theorem which establishes the correctness of Algorithm 3.1 is Theorem 4.1. To prove the main theorem:

- we have to show that the $\mathcal{E}$-*rules* when applied in a bottom-up fashion reduce a reducible DJ graph to its dominator tree (Lemma 4.4); and
- we have to show that variable elimination and top-down propagation are correct (Lemma 4.5).

We first define the reducibility of DJ graphs as well as flowgraphs [HU74].

Definition 4.1 *A DJ graph $\mathcal{G}$ is reducible if and only if we can partition the edges into two disjoint groups, called the* **forward** *edges and* **back** *edges, with the following two properties:*

1. *The forward edges form an acyclic graph in which every node can be reached from the* START *node of $\mathcal{G}$.*
2. *The back edges consist only of edges whose destination nodes dominate their source nodes.*

We can easily see that a DJ graph is reducible if and only if the corresponding flowgraph is also reducible. This can be easily verified by the construction of DJ graphs. The above definition has an important implication for verifying whether a DJ graph is reducible or not. If there is a simple cycle containing two *distinct* nodes that are at the same level, then the DJ graph is irreducible. This is easy to see, since for such graphs, we will have a "back" edge whose destination node does not dominate its source node. We will use this key insight in proving our first lemma, Lemma 4.1. Given a DJ graph $\mathcal{G}$ and a level k such that there are no J edges whose source nodes are at levels greater than k, Lemma 4.1 establishes that if $\mathcal{G}$ is reducible then there must be at least one non-join node at level k. This is important for our approach, since we apply the reduction rules in a bottom-up fashion.

In Lemma 4.2, we will show that the reduction rules preserve the reducibility of a DJ graph. Given this and Lemma 4.1 we can easily see that at every stage of the reduction process we can always find a non-join node for applying the $\mathcal{E}$-*rules*.

Next we show that our variable elimination is correct. For this we will first show, in Lemma 4.3, that when **ReduceLevel**(i) is called at step **2** and the call terminates, every J edge whose source node is at levels greater than or equal to i is eliminated; also, the flow

equation of each node y, whose level is greater than or equal to i, will depend only on the output flow variable $O_{idom(y)}$ of its immediate dominator node.

Finally we will prove the correctness of **DomTDPropagate()**. For this we will first show, Lemma 4.5, that when the procedure **DomTDPropagate()** is called at step **7**, the DJ graph has been reduced to its dominator tree. Given this we will also show, in Lemma 4.5, that when the **DomTDPropagate()** is invoked at step **7**, the flow equation at each node depends only on the flow variable of its immediate dominator, except the root node which depends on none.

We begin the proof chain by showing that if $\mathcal{G}$ is reducible, there always exists at least one non-join node at the maximum level.

Lemma 4.1 *Let $\mathcal{G}$ be a DJ graph, and let k be a 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.*

Proof: Suppose to the contrary that there were no non-join nodes at level k of the DJ graph. Let M be the set of nodes at this level. Without loss of generality we will remove all the self-loops in M.

We then show that there must exist a nontrivial cycle at level k, concluding that the DJ graph is irreducible. Since every node in M is a non-join node, it must have at least one incoming J edge, and the source node of this J edge must be at k level too, according to the property of DJ graphs. (Remember that every node can have at most one incoming D edge.) If this is the case we can traverse backwards over the J edges and still stay at the same level. Since there are only a finite number of nodes in M, we will eventually visit a node twice by this backward traversal. This implies the existence of a simple cycle consisting only of nodes from M. Notice that we have removed self-loops. By Definition 4.1, the DJ graph is not reducible. But this contradicts the assumption that it is reducible. Therefore the lemma is true and there must exist a non-join node at level k. ∎

The next lemma (Lemma 4.2) shows that applying $\mathcal{E}$-*rules* preserves reducibility of the DJ graph.

Lemma 4.2 *Let $\mathcal{G}$ be a reducible DJ graph. Let one of $\mathcal{E}$-rules be applied to $\mathcal{G}$, resulting in $\mathcal{G}^1$. Then $\mathcal{G}^1$ is also reducible.*

Proof:

The proof of the lemma is based on the following observation. In the three $\mathcal{E}$-*rules* we delete an edge, but only in E2b rule we also insert an edge.

First we consider the deletion case. From Definition 4.1 we know that if $\mathcal{G}$ is reducible we can partition the edges into two sets, called forward edges and back edges. The source node of a back edge dominates its destination node. It is obvious to see that deleting an edge from $\mathcal{G}$ does not violate the Definition 4.1, i.e., $\mathcal{G}^1$ will still be reducible.

Now for the insertion case we notice that a new edge is inserted only when E2b rule is applied. By applying E2b rule we delete an edge $y \rightarrow z$ and insert $x \rightarrow z$, where $x = idom(y)$. We will first show that if $y \rightarrow z$ is a forward edge so is $x \rightarrow z$, and if $y \rightarrow z$ is a back edge so is $x \rightarrow z$. Let $y \rightarrow z$ be a back edge. From Definition 4.1 we know that z **dom** y. Now since $x = idom(y)$ and $y.level > z.level$, z should also dominate x. Therefore $x \rightarrow z$ is a back edge too.

We can similarly argue for the case when $y \rightarrow z$ is a forward edge. Let $y \rightarrow z$ be a forward edge. From Definition 4.1 we know that z **!dom** y. Now since $x = idom(y)$ and $y.level > z.level$, z will also not dominate x. Therefore $x \rightarrow z$ is also a forward edge.

Since by adding $x \rightarrow z$ we never violate the definition of reducibility, $\mathcal{G}^1$ should also be reducible. ∎

Based on Lemmas 4.1 and 4.2, we can always apply $\mathcal{E}$-*rules* when there are still J edges in the DJ graph. Furthermore, at each stage of the reduction the rules preserve reducibility. Given these we will next show that our bottom-up reduction will always reduce a reducible DJ graph to its dominator tree, and any variable elimination performed during the reduction is correct. We state these results in the following lemma.

Lemma 4.3 *In eager elimination, when* **ReduceLevel**(i) *is called at step* **2** *and the call terminates:*

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

Proof:

First of all observe that when one of $\mathcal{E}$-*rules* is applied to a J edge $y \rightarrow z$, the edge $y \rightarrow z$ is eliminated (and a new edge $idom(y) \rightarrow z$ is possibly inserted). Also, in the procedure **ReduceLevel**(i) we apply the $\mathcal{E}$-*rules* for each outgoing edge of the non-join node returned by **GetNode**(i) (steps **8** and **9**). Finally, when one of $\mathcal{E}$-*rules* is applied to the J edge $y \rightarrow z$, O_y is eliminated in H_z, (either by substitution, as in E2a rule and E2b rule, or by computing the closure, as in E1 rule).

We will prove the rest of the lemma using induction on the loop index i at step [1]

Base Case: Loop index ($i = NumLevel - 1$). When **ReduceLevel**($NumLevel - 1$) is called we apply the appropriate $\mathcal{E}$-$rules$ for each J edge $y \rightarrow z$, such y is a non-join node at level $NumLevel - 1$ (step [9]). Since $\mathcal{E}$-$rules$ eliminates J edges $y \rightarrow z$, and also eliminates O_y in H_z, the two assertions of the lemma are true for the base case.

Induction hypothesis: Assume that the two assertions of the lemma are true for some loop index $i = k + 1$ less than the maximum level. This means that all J edges whose source nodes at levels greater than or equal to $k + 1$ are eliminated, and the flow equation of each node y with $y.level \geq k$ contains, on their RHS's, only the flow variable $O_{idom(y)}$.

Induction step: Given the hypethesis, we will show that the two assertions of the lemma are true for loop index $i = k$. From Lemma 4.2 we know that $\mathcal{E}$-$rules$ preserve reducibility, and from Lemma 4.1 we know that there exists at least one non-join node at level k. Let one of $\mathcal{E}$-$rules$ be applied to some $\mathcal{G}^j$ resulting in $\mathcal{G}^{j+1}$. Assume that the maximum level of $\mathcal{G}^i$ and $\mathcal{G}^{i+1}$ are the same. Since $\mathcal{G}^{j+1}$ is also reducible, there exists a non-join node at the maximum level (according to Lemma 4.1 again). Therefore, **GetNode**(k) when called at step [8] with the current level k, will return a non-join node. The procedure **GetNode**(k) will return $NULL$ only when there are no more non-join nodes at level k. Since the given DJ graph is assumed to be reducible, this situation can happen only when all the outgoing J edges from level k are eliminated. Hence when **ReduceLevel**(k) terminates, all the outgoing J edges from level k are eliminated.

Next to see that variables are appropriately eliminated, we examine each $\mathcal{E}$-$rules$. We know that each $\mathcal{E}$-$rules$, when applied to $y \rightarrow z$ will eliminate O_y from H_z by substituting it with a linear function of $O_{idom(y)}$. Therefore, when the procedure **ReduceLevel**(k) returns, two assertions of the lemma will be true.

∎

Lemma 4.4 *When the* Algorithm 3.1 *begins at step* [7]*, all the J edges have been eliminated.*

Proof: From Lemma 4.3 we know that when the call to **ReduceLevel**(i) at step [2] terminates, all J edges whose source nodes are at level i are eliminated. Also, the **foreach** loop at step [1] calls **ReduceLevel**(i) in the decreasing order of i. Therefore when the loop terminates, all the J edges have been eliminated. ∎

In Lemma 4.5, we will argue that when the **DomTDPropagate()** is invoked at step [7], the flow equation at each node depends only on the flow variable of its immediate dominator, except the root node which depends on none.

Lemma 4.5 *When* **DomTDPropagate()** *is invoked at step* [7]*, the flow equation at each node depends only on the flow variable of its immediate dominator.*

Proof: The procedure **DomTDPropagate()** is invoked at step 7 only when the call to **ReduceLevel**(1) is terminated. From Lemma 4.3 we know that when the call to **ReduceLevel**(1) terminates the flow equation of each node depends only on the output flow variable of its immediate dominator node. From the the validity of the lemma follows. ∎

Finally, the main theorem for correctness.

Theorem 4.1 Algorithm 3.1 *correctly computes the solutions to a set of data flow equations for a reducible flowgraph.*

Proof: From Lemma 4.4 we know that the $\mathcal{E}$-*rules*, when applied in a bottom-up fashion, reduces a DJ graph to its dominator tree. At this point, from Lemma 4.5 we know that the equation at each node contains, on its RHS, only the flow variable of the node's immediate dominator, while the equation at the START node depends on no one, meaning that the RHS of the equation is constant. Therefore, we can propagate the solution of the START node down the dominator tree to compute the solutions at all other nodes. Recall that the flow equation at each node depends on its immediate dominator. Therefore our top-down propagation yields a correct solution at every node. ∎

4.2 Complexity

In this section we will establish the time complexity of the eager elimination method. We will first show that the worst-case time complexity of eager elimination is $O(|E| \times |N|)$. Then we will show how the time complexity of eager elimination is related to the size of the dominance frontier relation. Since the size of the dominance frontier relation is linear in practice, we expect the time complexity of our eager elimination method to be linear for most practical programs.

It is easy to see that each time E1 and E2a rules are applied we eliminate one edge, and when E2b rule is applied we merely transform an edge $y \rightarrow z$ to $idom(y) \rightarrow z$. From this we can see that E1 and E2a are applied at most $O(|E|)$ times, whereas E2b rule can be applied as many as $O(|E| \times |N|)$ times. We state this in the following lemma.

Lemma 4.6 *The E1 and E2a rules will be applied at most $O(|E|)$ times, and E2b rule will be applied at most $O(|E| \times |N|)$ times.*

Proof: Each application of E1 and E2a rules removes one edge, and since none of the $\mathcal{E}$-*rules* increases the number of edges in the DJ graph, we can apply these two rules at most $O(|E|)$.

Each application of E2b rule to an edge $y \rightarrow z$ removes this edge and introduces $idom(y) \rightarrow z$. Since there are at most $O(|N|)$ levels, the derived edge of $y \rightarrow z$ moves up

at most $O(|N|)$ times. Finally, since there are only $O(|E|)$ edges, we can apply E2b rule at most $O(|E| \times |N|)$ times. ∎

Next it is also easy to see that for each application of E1 and E2 rules, we eliminate one flow variable. Since E2b rule is applied $O(|E| \times |N|)$, time complexity of eager elimination is $O(|E| \times |N|)$ function operation.

Theorem 4.2 *The number of steps required to transform a DJ graph $\mathcal{G}^0$ to its dominator tree $\mathcal{G}^M$ using $\mathcal{E}$-rules is bounded by $O(|E| \times |N|)$ steps.*

Proof: Follows from Lemma 4.6 and Lemma 4.4. ∎

Although an eager elimination can exhibit the $O(|E| \times |N|)$ time complexity in the worst case, we expect it to behave linearly in practice. The reason for this is based on the following observation. Remember that it is E2b rule that potentially makes eager elimination non-linear. In E2 rule we eliminate a J edge $y \rightarrow z$ and insert another J edge $x \rightarrow z$, where $x = idom(y)$. We can think of $x \rightarrow z$ as being "derived" from $y \rightarrow z$ (see also the Section 5). Notice that z will be in the dominance frontier of both x and y. An astute reader may observe that the total number of times $\mathcal{E}$-*rules* are applied is bounded by the size of the dominance frontier relation. Empirical studies have shown that the size of the dominance frontier relation is linear (with respect to the size of the original flowgraph) for most practical programs [CFR+91] (see also Section 8). Therefore, our eager elimination method can be expected to be linear for most practical programs.

4.3 Discussion

In this section we highlight some of the interesting features of the reduction process.

- In our reduction process we apply $\mathcal{E}$-*rules* in a bottom-up fashion. The order in which we apply $\mathcal{E}$-*rules* conforms to one of T1-T2 reduction sequences of the Hecht-Ullman method. Our E1 rule is equivalent to T1 rule, whereas our E2 rule is equivalent to T2 rule with one difference: Our E2 rule eliminates outgoing edges of a non-join node one at a time, whereas T2 rule eliminates all the outgoing edges in one shot.

- In this paper we follow the equation model of Ryder and Paull to present our framework. In their model, data flow problems are modeled using GEN and PRESERVE sets. In a more general setting, equations at each node are treated like functions and these functions are associated with the outgoing edges. We can easily extend our framework to this more general setting [Mar89, Tar81, Ros80].

- In proving the time complexity we ignored the time for **GetNode**(). During the initialization portion of the algorithm we ensure that non-joins appear before join nodes in *PriorityList*[i] at each level. Also, during the bottom-up reduction (at step **38**) we ensure that if a node z becomes a non-join node when the edge $y \rightarrow z$ is eliminated, we put the node at the head of the list. This way, we can guarantee constant time operation for **GetNode**().

5 Delayed Elimination Method

All elimination methods, except for the Allen-Cocke, optimize certain variable eliminations by delaying them (either implicitly as in the Hecht-Ullman and the Tarjan method, or explicitly as in the Graham-Wegman method) [RP86]. In previous sections we have shown how to eagerly eliminate variables in data flow equations using DJ graphs. Eager elimination can exhibit a worst-case quadratic complexity [RP86]. In this section we show how to "delay" some variable eliminations so that we can improve the asymptotic time complexity to $O(|E| \times \log(|N|))$ function operations. Before presenting our method, we will introduce the concept of Dominance Frontier Interval (Section 5.1) and Derived Edges (Section 5.2) of a J edge, that will simplify our presentation.

5.1 Dominance Frontier Interval

Let $y \rightarrow z$ be a J-edge in $\mathcal{G}^0$, and let $w = idom(z)$. By definition of dominance frontiers, we know that $y \rightarrow z$ will be in the dominance frontiers of all nodes on the reverse dominator tree path $y \xrightarrow{+} w$, excluding w. We call the path $y \xrightarrow{+} w$ the Dominance Frontier Interval path of the J-edge $y \rightarrow z$ [PB95]. Let x be the last node on this path. Then it is trivial to show that x will be a child of w and at the same level as z. The dominance frontier interval of a J-edge $y \rightarrow z$ can be compactly represented as $[y, w)$, the **half-open interval**, or as $[y, x]$, the **closed interval** (where $w = idom(z)$ and x is the node before w on the path $y \xrightarrow{+} w$). We will call y the **Bottom**, and w (x) the **Top** of the half-open (closed) interval.

Given the notion of dominance frontier interval for each J-edge $y \rightarrow z$, how fast can we determine **Top** node of the interval? For half-open interval it is trivial— for a J-edge $y \rightarrow z$, $w = idom(z)$ will be the **Top** node. In this paper we are interested in the **Top** element of the closed interval. For a single J-edge $y \rightarrow z$, computing the **Top** node of the closed interval $[y, x]$ takes time proportional to the number of nodes on the path $y \xrightarrow{+} w$, where w is $idom(z)$. But for a set A of J edges we can determine the **Top** node in time $O(|N| + |A|)$ by a depth-first traversal of the dominator tree. Since there are at most $O(|E|)$ J edges, the time complexity of determining **Top** nodes for all J-edges is, in the worst-case, $O(|N| + |E|)$. We have given one such algorithm in Appendix B. In Figure 4, the **Top** node of the closed interval for each J edge is annotated as $< x >$. In the rest of the section we will use the notation $\mathbf{cTop}_{y \rightarrow z}$, to denote the top node of the closed interval $[y, x]$ of a J edge $y \rightarrow z$.

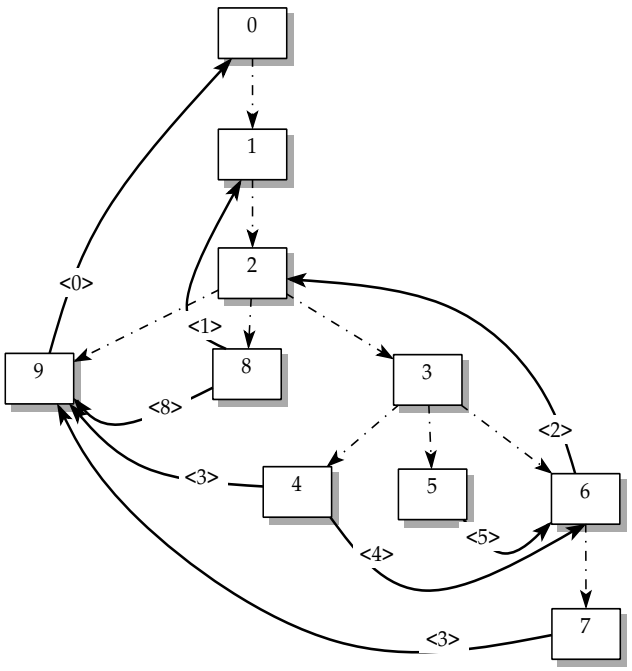

Figure 4: DJ graph annotated with **cTop** nodes.

5.2 Derived Edges

Next we introduce the concept of "derived edges". Recall that when an E2b rule is applied to an edge $y \rightarrow z$ in $\mathcal{G}^i$, we replace $y \rightarrow z$ by $idom(y) \rightarrow z$ in $\mathcal{G}^{i+1}$. Here we "derive" the edge $idom(y) \rightarrow z$ from $y \rightarrow z$. More formally, the notion of derived edge is given below.

Definition 5.1 (derived edge) *An edge in $x \rightarrow z$ in $\mathcal{G}^i$ is called a* **derived edge** *of an edge $y \rightarrow z$ in $\mathcal{G}^0$ if either $x = y$ or $x \rightarrow z$ is created in $\mathcal{G}^i$ and x* **dom** *y in $\mathcal{G}^0$.*

For example, consider Figure 3. We can see that the edge $6 \rightarrow 9$ in Figure 3(b) is a **derived edge** of $7 \rightarrow 9$ in Figure 3(a). The edge $2 \rightarrow 1$ in Figure 3(k) is again derived from the edge $8 \rightarrow 1$ in Figure 3(j).

5.3 Worst-Case Quadratic Complexity of Eager Elimination Method

In this section we will examine why eager elimination exhibits quadratic behavior. First let us examine the $\mathcal{E}$-*rules*. Each application of E1 or E2a rule eliminates one edge from $\mathcal{G}^i$ to produce $\mathcal{G}^{i+1}$. Therefore we will apply these two rules at most $O(|E|)$ times. What about E2b rule?

This rule, when applied to an edge $y \rightarrow z$ in $\mathcal{G}^i$, merely transforms $y \rightarrow z$ to its derived edge $idom(y) \rightarrow z$ in $\mathcal{G}^{i+1}$. This can cause an efficiency problem, because for each edge $y \rightarrow z$ in $\mathcal{G}^0$, we may potentially create $O(|N|)$ derived edges for $y \rightarrow z$ in the reduction sequence $\mathcal{G}^0 \stackrel{*}{\Longrightarrow} G^k$ (for some k) before eliminating it. In total, the E2b rule can be applied $O(|E| \times |N|)$ times. Thus we can see that the E2b rule is a bottle-neck that makes eager elimination method to have worst-case quadratic time complexity. Ideally, we want to be able to apply the E2b rule at most $O(|E|)$ times, so that we can reduce a DJ graph to a trivial node in $O(|E|)$ reduction steps. We introduce a new rule, the D2b rule, in place of the E2b rule, that will do the job for us. We will show later that the D2b rule will be applied at most $O(|E|)$ times.

With the D2b rule we must also modify how variables in flow equations are eliminated. We will introduce the concept of delayed variable elimination that enable us to solve a system of data flow equations in $O(|E| \times \log(|N|)$ function operations.

5.4 $\mathcal{E}$-*rules* Revisited: The $\mathcal{D}$-*rules*

Recall that when an E2b rule is applied to an edge $y \rightarrow z$ in $\mathcal{G}^0$, we replace it by $idom(y) \rightarrow z$ in $\mathcal{G}^1$. Therefore the **derived edge** of $y \rightarrow z$ moves up the DJ graph until its source and destination nodes are at the same level. When this (eventually) happens we will remove it by applying either E1 or E2a rule. An astute reader will immediately observe that the source node x of the **derived edge** of $y \rightarrow z$ (when the source and the destination node are at the same level) is the same as $\mathbf{cTop}_{y \rightarrow z}$ for the closed interval $[y, x]$ in $\mathcal{G}^0$. Therefore, rather than moving up the **derived edge** of $y \rightarrow z$ one step at a time (by applying the E2b rule), we will move it in one shot using the D2b rule. The complete definition of the D2b rule is given below.

Definition 5.2 (D2b rule) *Let $\mathcal{G}^i = (N, E)$ be the ith reduced DJ graph. Let y be a non-join node, let $y \rightarrow z$ be an outgoing edge, and let $x = \mathbf{cTop}_{y \rightarrow z}$. If $idom(y) \neq idom(z)$ then*

$$D'2b(< \mathcal{G}^i, N, E, y \stackrel{J}{\rightarrow} z, x >) \implies < \mathcal{G}^{i+1}, N, (E - \{y \stackrel{J}{\rightarrow} z\}) \cup x \stackrel{J}{\rightarrow} z > .^4$$

We will carry over the definition of E1 and E2a rules to our delay elimination method and call their corresponding rules as the D1 and the D2a rule. There is no difference between D1 and E1, and between D2a and E2a as far as the graph reduction is concerned, but the two sets of rules are completely different for variable elimination. In the rest of the paper we will use $\mathcal{D}$-*rules* to mean D1, D2a, and D2b collectively. With the definition of $\mathcal{D}$-*rules*, we will next focus on variable elimination.

[4] Again we will not insert $x \rightarrow z$ in $\mathcal{G}^{i+1}$ if it is already present in $\mathcal{G}^i$.

5.5 Delaying Variable Elimination

The key intuition behind delayed elimination is to delay the elimination of variable O_y in equation H_z until the source and the destination node of the **derived edge** of $y \rightarrow z$ are at the same level. Before presenting this method we need to describe the concept of path compression on a dominator tree.

The path compression on a dominator tree is performed whenever we invoke the procedure **CompPath**(x, y), where x and y are nodes on the (compressed) dominator tree such that x is an ancestor of y. **CompPath**(x, y) performs the following operations on a dominator tree:

1. **Delayed Substitution:** For each node w on the (compressed) dominator path $x \stackrel{+}{\rightarrow} y$, excluding x, express variable O_w as a linear function of O_x, by top-down traversal of the path.

2. **Path Compression:** Make all the nodes on the path $x \stackrel{+}{\rightarrow} y$, the children of x.

These two operations are performed in the path order, that is for any two nodes u and v in the path, if u is an ancestor of v in the tree then we process u earlier than v. It is important to note that when a node w on the path $x \stackrel{+}{\rightarrow} y$ is made a child of x, all children of w except the one on the path, are still w's children. In other words, only one of w's children (i.e., the one on the path $x \stackrel{+}{\rightarrow} y$) will change its parent from w to x.

Procedure CompPath(x, y)
{

50: **foreach** node $w \neq x$ in the path from $x \stackrel{+}{\rightarrow} y$ and in the path order **do**
51: Express O_w as a linear function of O_x.
52: Make w a child of x.
53: **endfor**

}

Given the operation **CompPath**(), we will next discuss how to delay the elimination of variables. The key point in delayed elimination is to eliminate variables only when applying D1 or D2a rule, but not when applying D2b rule. We will revise the NodeStructure of Algorithm 3.1 as follows:

```
struct NodeStructure {
    int indegree ; /* as defined in Algorithm 3.1 */
    int level ; /* as defined in Algorithm 3.1 */
    struct ListStructure *jedges ; /* List of J edges y → z for which */
    /* this node is the cTop.  This list is initially empty */
}
```

This structure is the same as in Algorithm 3.1, but with an additional attribute `*jedges`, which is nothing but a pointer to a list of J edges. As we will show shortly, this list is built during D2b rules and consumed during either D1 or D2a rules.

Algorithm 5.1 *The algorithm* **MainElim()** *with $\mathcal{D}$-rules and delayed elimination can be used for solving a system of data flow equations.*

In Algorithm 5.1 we will use the main procedure of Algorithm 3.1 for reducing the DJ graph in a bottom-up fashion. The only difference between Algorithm 3.1 and Algorithm 5.1 is that we call procedures **Delayed1**($y \rightarrow y$), **Delayed2a**($y \rightarrow z$) and **Delayed2b**($y \rightarrow z$), instead of **Eager1**($y \rightarrow y$), **Eager2a**($y \rightarrow z$) and **Eager2b**($y \rightarrow z$), respectively.

In the delayed elimination, variables are eliminated only when **Delayed1**($y \rightarrow y$) and **Delayed2a**($y \rightarrow z$) are invoked. We do not eliminate variables when **Delayed2b**($y \rightarrow z$) is invoked. When **Delayed2b**($y \rightarrow z$) is invoked we keep track of **Bottom** node y in $x = \mathbf{cTop}_{y \rightarrow z}$ in the list x->$jedges$ (step **71**). In **Delayed2b**($y \rightarrow z$) we also delete the edge $y \rightarrow z$ and insert $x \rightarrow z$ (if it does not already exist). When **Delayed1**($x \rightarrow x$) or **Delayed2a**($x \rightarrow z$) is later invoked, we first perform the path expression by invoking **CompPath**(x, y) (step **55** or step **61**, respectively). As explained earlier, for each node w on the (compressed) dominator tree path $x \xrightarrow{+} y$, excluding x, this procedure first expresses variable O_w as a linear function of O_x, by going down the tree path. It also compresses the path by making every w a child of x. We will illustrate the path compression through an example later in Section 5.7.

In the procedure **Delayed1**($y \rightarrow y$), the function **GetJedge**(y) returns a J edge $u \rightarrow w$ from the list y->$jedges$ (step **54**). For each such edge we invoke **CompPath**(y, u) to perform path compression and delayed variable substitution (step **55**). Finally, we compute the closure of the recursive equation H_y to break the loop (step **58**). At step **59** we eliminate the edge (as in **Eager2a**()). It is important to remember that the destination node w of the J edge $u \rightarrow w$, returned by **GetJedge**(y), will not necessarily be the same as y. This is because the y can be a **cTop** node for many different J edges that were inserted in y->$jedges$ by some previous invocations of **Delayed2b**(). We will eliminate all such J edges from y->$jedges$ the first time the procedure **Delayed1**($y \rightarrow y$) is invoked on y.

Procedure Delayed1($y \rightarrow y$)
{

54: **while**(($u \rightarrow w$ = **GetJedge**(y)) $\neq NULL$)) **do**
`/* Get the next J edge for which this node is a top. */`
55: **CompPath**(y,u) ; `/* Replace node equations on the path */`
`/*` $y \xrightarrow{+} u$ `as a function of` O_y `*/`
56: Eliminate O_u on the RHS of H_y by replacing it
with a linear function of O_y computed in the previous step.
57: **endwhile**

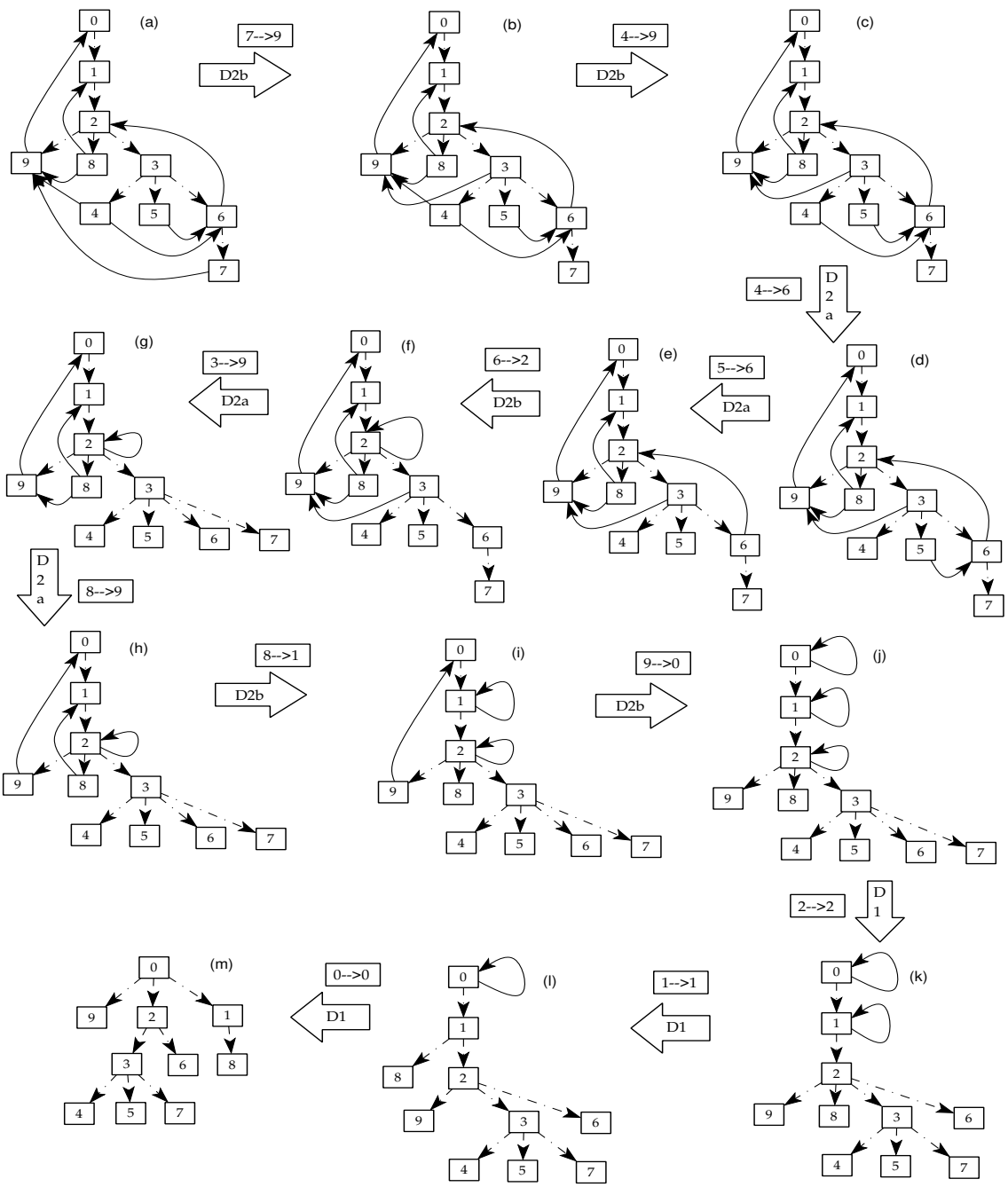

Figure 5: A trace of the DJ graph reduction using $\mathcal{D}$-*rules*.

58: Compute $f^*(O_y)$. `/* Compute the closure. */`
59: Delete the edge $y \rightarrow y$;

}

The steps in **Delayed2a**() are similar to **Delayed1**() except that we do not compute closure (since there are no self-loops). The function **GetJedge**(y) returns J edge $u \rightarrow w$ for which y is the **cTop** node. The function **CompPath**(y, u) then performs delayed variable substitution and compress the path. At step **62** we replace O_u in equation H_w with the linear function of O_y computed in step **61**. After the **while** loop, we replace O_y in H_w by the RHS of H_y (step **64**). Again, as in **Delayed1**(), the destination node w of the J edge $u \rightarrow w$ will not necessarily be the same as z. Finally, note that w can never be same as y, if it were then we should have invoked **Delayed1**() prior to invoking **Delayed2a**().

The operations from step **65** to step **69** are the same as in **Eager2a**(), and essentially perform a DJ graph reduction.

Procedure Delayed2a($y \rightarrow z$)
{

60: **while**$(((u \rightarrow w = \textbf{GetJedge}(y)) \neq NULL))$ **do**
`/* Get the next J edge for which this node is a top. */`
61: **CompPath**(y,u) ; `/* Express the node equations on the path */`
`/*` $y \xrightarrow{+} u$ `as a function of` O_y `*/`
62: Eliminate O_u on the RHS of H_w by replacing it
with the linear function of O_y computed in the previous step.
63: **endwhile**
64: Finally eliminate O_y in H_z by replacing it with a linear function of O_x,
where $x = parent(y)$ on the compressed dominator tree.
65: Delete the edge $y \rightarrow z$;
66: $z.indegree = z.indegree - 1$;
67: **if**$(z.indegree \leq 1)$ **then** `/*` z `becomes a non-join node */`
68: Put z at the head of $PriorityList[z.level]$ list ;
69: **endif**

}

In the procedure **Delayed2b**($y \rightarrow z$) we do not eliminate variables. At step **71** we save the J edge $y \rightarrow z$ in x->$jedges$, where $x = \textbf{cTop}_{y \rightarrow z}$ (step **71**). These J edges (returned by the function **GetJedge**()) will be processed later in the procedure for D1 and D2a rules at step **54** and step **60**, respectively. The operations from step **72** to step **77** are the same as in **Eager2b**(), and essentially do DJ graph reduction. Again, notice that x can be a **cTop** node for many different J edges, all such J edges were inserted in the list x->$jedges$ when D2b rule was applied to these edges. To ensure constant time operation for inserting, we insert the J edge $y \rightarrow z$ at the head of the list x->$jedges$.

Procedure Delayed2b($y \rightarrow z$)
{

70: x = **cTop**$_{y \rightarrow z}$;
71: Insert node $y \rightarrow z$ in the list x ->$jedges$.
72: Delete the edge $y \rightarrow z$;
73: **if**($x \rightarrow z$ exists) **then**
74: $z.indegree = z.indegree - 1$;
75: **else**
76: Insert a new J edge $x \rightarrow z$;
77: **endif**

}

The function **GetJedge**(w) removes the first J edge from node w's J edges list w ->$jedges$ (if one exists) and returns it. This function is invoked by **Delayed1**() and **Delayed2a**(). For reasons explained later (Section 6.3), we will operate the list w->$jedges$ as a *queue* structure. Therefore we always return J edges from the tail of the list.

Function GetJedge(w)
{

78: **if** the list w->$jedges$ is not empty **then**
79: Remove the first edge from the tail of list and return it.
80: **else**
81: **return** $NULL$
82: **endif**

}

5.6 Top-Down Propagation

The top-down propagation phase in the delayed elimination method is similar to the propagation phase in the eager elimination method, except that we propagate the data flow information on the compressed dominator tree. When Algorithm 5.1 terminates, every node's equation depends only on the flow variable of its parent in the compressed dominator tree. In other words, the flow equation at each node y is a linear function of O_x, where $x = Parent(y)$ in the compressed dominator tree. We will prove this formally later in Section 6. To propagate the data flow information on the compressed dominator tree, we invoke **cDomTDPropagate()** at step **7** instead of **DomTDPropagate()**. The procedure **cDomTDPropagate()** is defined below:

cDomTDPropagate()
{

83: Solve the data flow equation of the START node.

84: Propagate the solution on the compressed dominator tree in a top-down fashion, using the solution at node $x = Parent(y)$ to substitute variable O_x on the RHS of equation $O_y = f(O_x)$, to compute the solution at y.

}

5.7 An Example

Here we illustrate the delayed elimination method for our example DJ graph. The complete DJ graph reduction process is shown in Figure 5.

1. For the DJ graph in Figure 5(a) we apply D2b rule, and so we invoke **Delayed2b**($7 \rightarrow 9$). Since variables are not eliminated during D2b reduction, no flow equation changes in response to the graph reduction from Figure 5(a) to Figure 5(c). But we insert the J edge $7 \rightarrow 9$ in the list 3->$jedges$ (since node 3 is the **cTop** of the J edge $7 \rightarrow 9$). For the DJ graph in Figure 5(b) we again apply D2b rule, invoking **Delayed2b**($4 \rightarrow 9$). Once again we insert the J edge $4 \rightarrow 9$ in the list 3->$jedges$).

2. For the DJ graph in Figure 5(c) we apply D2a rule, invoking **Delayed2a**($4 \rightarrow 6$). At node 4, we have 4->$jedges = \emptyset$; therefore the procedure **GetJedge**(4) would return $NULL$ (step **60**). At step **64** we eliminate O_4 on the RHS of H_6 by replacing it with a linear function of O_3. Once this is done, the flow equation at node 6 becomes

$$O_6 = P_6(P_4O_3 + G_4 + O_5) + G_6$$

3. For the DJ graph in Figure 5(d) we apply D2a rule, invoking **Delayed2a**($5 \rightarrow 6$). At node 5, we again have 5->$jedges = \emptyset$; therefore the procedure **GetJedge**(5) would return $NULL$ (step **60**). At step **64** we eliminate O_5 on the RHS of H_6 by replacing it with a linear function of O_3. Once this is done, the flow equation at node 6 becomes

$$\begin{aligned} O_6 &= P_6(P_4O_3 + G_4 + P_5O_3 + G_5) + G_6 \\ &= (P_6P_4 + P_6P_5)O_3 + P_6G_4 + P_6G_5 + G_6 \end{aligned}$$

   Notice that at this point O_6 depends only on O_3 (where node 3 is a parent of 6 on the dominator tree).

4. Next we apply D2b rule to the edge $6 \rightarrow 2$, transforming the DJ graph of Figure 5(e) to Figure 5(f). We invoke **Delayed2b**($6 \rightarrow 2$), in which we store $6 \rightarrow 2$ in 2->$jedges$ (step **71**).

5. Next we apply D2a rule to the edge $3 \rightarrow 9$, transforming the DJ graph of Figure 5(f) to Figure 5(g). We invoke **Delayed2a**$(3 \rightarrow 9)$. At this point 3->*jedges* contains edges $4 \rightarrow 9$ and $7 \rightarrow 9$. So at step **60** the procedure **GetJedge**(y) returns these two edges, one after another. Let $4 \rightarrow 9$ be the first edge returned, and so at step **61** we invoke **CompPath**$(3, 4)$. In the procedure **CompPath**() we express the flow equation at every node on the path $3 \xrightarrow{+} 4$, excluding 3, as a function of O_3. Since 4 is the only node on this path, and its equation is already in the required form we return from the procedure **CompPath**$(3, 4)$. At step **50** we replace O_4 on the RHS of H_9 as a linear function of O_3. The resulting equation of node 9 is

$$O_9 \quad = \quad P_9(P_4O_3 + G_4 + O_7 + O_8) + G_9$$

Next **GetJedge**$(()y)$ would return the J edge $7 \rightarrow 9$ from the list 3->*jedges*. Therefore we invoke **CompPath**$(3, 7)$. Within the procedure **CompPath**() we first express the flow equation of every node on the path $3 \xrightarrow{+} 7$, excluding 3, as a function O_3 in a top-down fashion. The path $3 \xrightarrow{+} 7$ contains nodes 6 and 7 (excluding 3). The equation at node 6 is already expressed as a function of O_3. But the equation at node 7 is expressed in terms of O_6. Therefore we replace O_6 in H_7 by a linear function of O_3. By doing so the new equation at node 7 becomes

$$\begin{aligned} O_7 &= P_7O_6 + G_7 \\ &= P_7((P_6P_4 + P_6P_5)O_3 + P_6G_4 + P_6G_5 + G_6) + G_7 \\ &= (P_7P_6P_4 + P_7P_6P_5)O_3 + P_7P_6G_4 + P_7P_6G_5 + P_7G_6 + G_7 \end{aligned}$$

At step **52** we also make node 7 a child of node 3, resulting in the compressed dominator tree shown in Figure 5(g).

Next, at step **62** we eliminate O_7 on the RHS of H_9 by replacing it with a linear function O_3. By doing so the new equation at node 9 becomes

$$\begin{aligned} O_9 &= P_9(P_4O_3 + G_4 + O_7 + O_8) + G_9 \\ &= P_9P_4O_3 + P_9O_7 + P_9O_8 + P_9G_4 + G_9 \\ &= P_9P_4O_3 + P_9((P_7P_6P_4 + P_7P_6P_5)O_3 + P_7P_6G_4 + \\ &\quad P_7P_6G_5 + P_7G_6 + G_7) + P_9O_8 + P_9G_4 + G_9 \\ &= P_9P_4O_3 + (P_9P_7P_6P_4 + P_9P_7P_6P_5)O_3 + P_9P_7P_6G_4 + \\ &\quad P_9P_7P_6G_5 + P_9P_7G_6 + P_9G_7 + P_9O_8 + P_9G_4 + G_9 \\ &= (P_9P_4 + P_9P_7P_6P_4 + P_9P_7P_6P_5)O_3 + P_9O_8 + P_9P_7P_6G_4 + \\ &\quad P_9P_7P_6G_5 + P_9P_7G_6 + P_9G_7 + P_9G_4 + G_9 \end{aligned}$$

Finally at step **62** we eliminate O_3 on the RHS of H_9 by replacing it with a linear function of O_2. By doing so the equation at node 9 becomes

$$\begin{aligned} &= (P_9P_4 + P_9P_7P_6P_4 + P_9P_7P_6P_5)(P_3O_2 + G_3) + P_9O_8 + P_9P_7P_6G_4 + \\ &\quad P_9P_7P_6G_5 + P_9P_7G_6 + P_9G_7 + P_9G_4 + G_9 \end{aligned}$$

6. We can continue to eliminate variables as described above at other nodes. We encourage interested readers to do so.

6 Correctness and Complexity of Delayed Elimination Method

In this section we prove the correctness of Algorithm 5.1 and analyze its complexity. We use some of the results of the eager elimination method.

6.1 Correctness

We prove the correctness of the delayed elimination method along the same line as in the eager elimination method. We will focus on showing (1) that $\mathcal{D}$-*rules* when applied in a bottom-up fashion eliminate all the J edges (Lemma 6.4), and (2) that variable elimination is properly performed in the bottom-up phase (Lemma 6.5). First note that Lemma 4.1 also holds for the delayed elimination method. As in the eager elimination method we always apply $\mathcal{D}$-*rules* to a non-join node.

In the following lemma, we prove that the application of $\mathcal{D}$-*rules* will not introduce irreducibility into a reducible DJ graph.

Lemma 6.1 *Let $\mathcal{G}$ be a reducible DJ graph. Let one of $\mathcal{D}$-rules be applied to $\mathcal{G}$, resulting in $\mathcal{G}^1$. Then $\mathcal{G}^1$ is also reducible.*

Proof: Recall that, in terms of graph reduction, E2b rule is the only difference between $\mathcal{E}$-*rules* and $\mathcal{D}$-*rules*. When E2b rule is applied to an edge $y \rightarrow z$, we first delete $y \rightarrow z$ and then insert $x \rightarrow z$, where $x = \mathbf{cTop}_{y \rightarrow z}$. We use the results from Lemma 4.2 and will only need to prove that the insertion of $x \rightarrow z$ will not introduce irreducibility. We complete our proof is by case analysis.

Case 1: $y \rightarrow z$ is a back edge. So z will dominate y. By definition of **cTop**, x dominates y and $x.level = z.level$. Therefore x is the same as z, and $x \rightarrow z$ is a self-loop, which is a back edge.

Case 2: $y \rightarrow z$ is a forward edge. By definition of **cTop**, x dominates y and $x.level = z.level$. But in this case $x \neq z$; otherwise, $y \rightarrow z$ would be a back edge. Given the fact that $y \rightarrow z$ is a forward edge and x dominates y, we conclude that $x \rightarrow z$ is also a forward edge.

■

To show that $\mathcal{D}$-*rules* eliminate all the J edges in a bottom-up fashion, we will first prove a result similar to Lemma 4.3.

Lemma 6.2 *In delayed elimination,*

1. *at the time when* **ReduceLevel**(i) *is invoked at step* **2**, *all the J edges whose source nodes are at levels greater than i are either eliminated or properly deposited in their corresponding* **cTop** *nodes.*

2. *at the time when the call to* **ReduceLevel**(i) *terminates, eliminated are all the J edges whose source nodes are at level i, and those that were previously deposited in the corresponding* **cTop** *nodes at level i.*

Proof: We will prove this lemma by induction on the loop index i at step **1**.

Base case: ($i = NumLevel - 1$). The first assertion is trivial since there are no nodes at level $NumLevel$ or beyond. For the second assertion, it is also obvious to see that **Delayed2b**() rule will properly deposit J edges in their corresponding **cTop** nodes, and the other two procedures (**Delayed1**() and **Delayed2a**()) will properly eliminate J edges whose source and destination nodes both at level $NumLevel - 1$.

Induction hypothesis: Assume that both assertions of the lemma are true for $i = k + 1$.

Induction step: We will show that both assertions are true for $i = k$. The first assertion follows immediately from the induction hypothesis. Now we are to show that the second assertion holds. Let e be a J edge with its source and/or destination node at level k. There are two cases:

Case 1: The source node of e is at this level. Then this edge will be either deposited in its corresponding **cTop** node by applying D2b rule, or eliminated by applying D1 or D2a rule.

Case 2: The destination node of e is at this level, and e was previously deposited in the corresponding **cTop** at this level. D1 or D2a rule will discard e and eliminate its derived edge.

■

Notice the difference between Lemma 4.3 and Lemma 6.2. In Lemma 6.2, we do not claim the following: If the flow equation of any node at a level less than i contains, on its RHS, the flow variable of a node at level i, this variable will be eliminated and substituted by the flow

variable of its parent (as we did in Lemma 4.3). This is because we are delaying the substitution of some variables in the delayed elimination method.

Next, we will prove the correctness of delayed variable elimination (Lemma 6.3). First of all observe that variable eliminations are delayed only by D2b rule. In other words, when D2b rule in applied to $u \rightarrow w$, the elimination of variable O_u in equation H_w is delayed until D1 or D2a rule is applied to the derived edge $y \rightarrow w$, where $y = \mathbf{cTop}_{u \rightarrow w}$. But we will apply the D1 or D2a rule only when we are processing nodes at level $w.level$ (i.e., loop index $i = w.level$ in the **for** loop at step $\boxed{1}$).

Lemma 6.3 *At the time when* **ReduceLevel**(i) *terminates, all the variables O_u that are associated with nodes u at level greater than i but still exist in the equation H_w for nodes w at level i are eliminated. Furthermore, any of these equations H_w will become dependent only on variable $O_{parent(w)}$.*

Proof: We prove this by induction on the loop index i of step $\boxed{1}$.

Base case: Loop index $i = NumLevel - 1$. Our claim is obviously true in this case since there are no nodes at levels greater than $NumLevel - 1$.

Induction hypothesis: Assume that the assertion is true for $i = k + 1$.

Induction step: Loop index $i = k$. From our induction hypothesis we know that for $i > k$, the assertion of the lemma is true. This means that all the variables O_u, with $u.level > k$, that exists in equations H_w at levels $i > k$ are eliminated, and these equations become dependent only on the output variable of their parent node. Now we have to show that the assertion of the lemma is true for $i = k$.

Now let O_u be a variable such that (1) it exists in equation H_w, (2) $u.level > k$, and (3) $w.level = k$. Then **Delayed2b**() must have deposited $u \rightarrow w$ in y->$jedges$. This edge will now be processed by either **Delayed1**() or **Delayed2a**(). In either case, **CompPath**(y, u) will be invoked. When **CompPath**(y, u) terminates, the equation H_x for every node x on the path $y \stackrel{+}{\rightarrow} u$ (excluding y) will depend only on variable O_y. In particular, H_u now depends only on O_y. By substituting variable O_u on the RHS of equation H_w with the current RHS of H_u, we eliminate O_u from H_w, and so the RHS of H_w will contain O_y. Now if $w = y$ (i.e., **Delayed1**() is invoked), a closure operation will be performed. Otherwise (i.e., **Delayed2a**() is invoked), O_y is further eliminated from H_w by substituting it with the RHS of H_w. In either case, equation H_w will now depend only on variable $O_{parent(w)}$. Hence the validity of the lemma.

∎

The next two lemmas state that at the end of the bottom-up elimination phase, (1) all the J edges are eliminated, and (2) the flow equation of every node (except for the root) depends only on the variable associated with the node's parent. These two lemmas are corresponding to Lemmas 4.4 and 4.5, respectively.

Lemma 6.4 *When* Algorithm 5.1 *begins at step* **7** *all the J edges have been eliminated.*

Proof: This lemma follows from (1) Lemma 6.2 and (2) the fact that the call to **ReduceLevel**(1) has completed before step **7**. ∎

Lemma 6.5 *When* **cDomTDPropagate()** *is invoked at step* **7**, *the flow equation at each node depends only on the flow variable of its parent on the final compressed dominator tree.*

Proof: Follows from Lemma 6.3 since **cDomTDPropagate()** is invoked only after **ReduceLevel**(0) has completed. ∎

Theorem 6.1 Algorithm 5.1 *correctly computes the solutions to a set of data flow equations for a reducible flowgraph.*

Proof: Follows from Lemmas 6.1, 6.4, and 6.5. ∎

6.2 Complexity

In this section we will analyze the complexity of the delayed elimination method. Recall that in the eager elimination method, E2b rule was the bottleneck. Here we will first show that we can reduce the DJ graph to a compressed dominator tree in linear time using $\mathcal{D}$-*rules*. Next we will analyze the total cost of variable elimination.

Lemma 6.6 *The $\mathcal{D}$-rules will be applied at most $O(|E|)$ times.*

Proof: First of all observe that each application of D1 or D2a rule will eliminate one edge. Since there are $|E|$ edges, these two rules can applied at most $O(|E|)$ times.

Now when we apply D2b rule to an edge $y \rightarrow z$ we remove this edge and introduce $x \rightarrow z$, where x is the $\mathbf{cTop}_{y \rightarrow z}$ of $y \rightarrow z$. Once this is done we will never apply D2b rule to the derived edge $x \rightarrow z$, since $idom(x) = idom(z)$ and so we could only apply D1 or D2a rule for this edge. Also, for every J edge there is a unique closed **Top** node. Since there are $|E|$ edges in the original DJ graph, we will apply D2b rule at most $O(|E|)$ times. ∎

Even though it takes only linear time to reduce a DJ graph into a compressed dominator tree, the total cost of variable elimination is worse than linear because of path compressions performed during the invocation of procedures **Delayed1**() and **Delayed2b**(). First observe that the number of path compressions, denoted e, performed in the entire elimination phase is equal to the number of D2b rules applied. In the following we will show that the total cost for e path compressions is bounded by $O(e \times \log|N|)$. We mainly use the results from Lucas [Luc90] and Tarjan and van Leeuwen [TvL84].

If a node x is a **cTop** in a path compression, then x is called the *root* of the compression. We define a sequence of path compressions on an initial tree $T = T^0$ to be a sequence $(C_1, \ldots, C_e)$, such that it transforms T into the final compressed tree T^e (with C_i transforming T_{i-1} into T_i). Lucas introduces the notion of *Rising Roots Condition* as follows:

Definition 6.1 *A sequence of path compressions* $(C_1, \ldots, C_m)$ *satisfies the* Rising Roots Condition (RRC) *if and only if for every node* x*, if* x *appears as a non-root node in any compression* C_i*, then for every* $j > i$*,* x *appears as a non-root node in* C_j *if* C_j *is a compression from* y *and* y *is a descendant of* x *in* T^{j-1}*.*

Lucas shows that a sequence of path compressions on a tree that satisfies the RRC corresponds to some sequence of intermixed *union* and *find* operations used in the disjoint set union problem; and conversely, a sequence of intermixed *union* and *find* corresponds to some sequence of path compressions satisfying the RRC.

Tarjan and van Leeuwen, in a previous work [TvL84], showed that the time complexity for a sequence of e intermixed *union* and *find* operations is Consequently, if we can show that the sequence of path compressions does satisfy the RCC, it immediately follows that the total cost of our e path compressions is also $O(e \times \log|N|)$.

Lemma 6.7 *The sequence of path compressions performed during our bottom-up reduction satisfies the RRC.*

Proof: Our path compressions are ordered by the levels of their roots (i.e., the **cTop** nodes). Therefore, once a node has been a non-root node in a compression, it will never be a root node in any future compression. ∎

Theorem 6.2 *The time complexity of the delayed elimination method is* $O(|E| \times \log(|N|))$.

Proof: The number e of path compressions is bounded by $|E|$. ∎

Buchsbaum et al. have recently established the lower bound $\Theta(e \times \log(n))$ for a sequence of e (order-preserving) path compressions satisfying the RRC on an initial tree of n nodes [BST95]. For most flowgraphs, $|E| = O(|N|)$, so the above bound is tight.

6.3 Discussion

Before closing this section, we want to point out some interesting features of our delayed elimination method.

- Our D1 rule is similar to Hecht-Ullman's T1 rule and Graham-Wegman's T1′ rule. Our D2a rule is similar to Hecht-Ullman's T2 rule and Graham-Wegman's T2′ rule. However, our D2b rule is unique. We exploit the properties of DJ graphs during the preprocessing step to identify the **cTop** nodes. We can think of the **cTop** nodes as marking points where the appropriate delayed variables are to be eliminated.

- Although not detailed in our algorithm descriptions, we will use a queue to store the list of J edges deposited in a **cTop** node. Consequently, when we start a sequence of path compressions all with the node as the root, we will compress longer paths before shorter ones. This does not change the time complexity of our method, but it is a pragmatic choice.

- Another interesting property to note is that the number of dominance frontier interval paths that can pass a node is less than or equal to the number of edges in its dominance frontier set. Therefore the total length of the dominance frontier interval paths can be no more than the size of the dominance frontier relation. Cytron et al. have shown that the size of dominance frontiers is linear in practice, and so we expect the time complexity of the delayed elimination method also to be linear in practice. In Section 8 we provide measurements which support this claim.

- We showed that the time complexity of our delayed approach is $O(|E| \times \log(|N|)$. An interesting open problem to pose here is: Is there a linear time algorithm for data flow analysis? We conjecture that it is possible to find a linear time algorithm for data flow analysis, at least for reducible flowgraphs.

7 Handling Irreducibility

In this section we will show how to handle irreducibility in flowgraphs with delayed elimination.[5] Recall that a flowgraph is irreducible if we cannot partition the edges into forward edges and back edges such that the destination nodes of the back edges dominate the source nodes. Figure 6(a) gives an example of an irreducible flowgraph, and its DJ graph is given in Figure 6(b). In our elimination method we detect irreducibility if during the bottom-up reduction **GetNode**(i) returns $NULL$ but there are more J edges to be processed at this level. At step **3** if $PriorityList[i].head$ is not $NULL$ then there are more nodes to be processed at this level, but none of them are non-join nodes. This condition is sufficient to signal irreducibility, and we invoke the procedure **CollapseIrreducible**(i) to handle irreducibility at level i.

[5] We can similarly handle irreducibility with eager elimination.

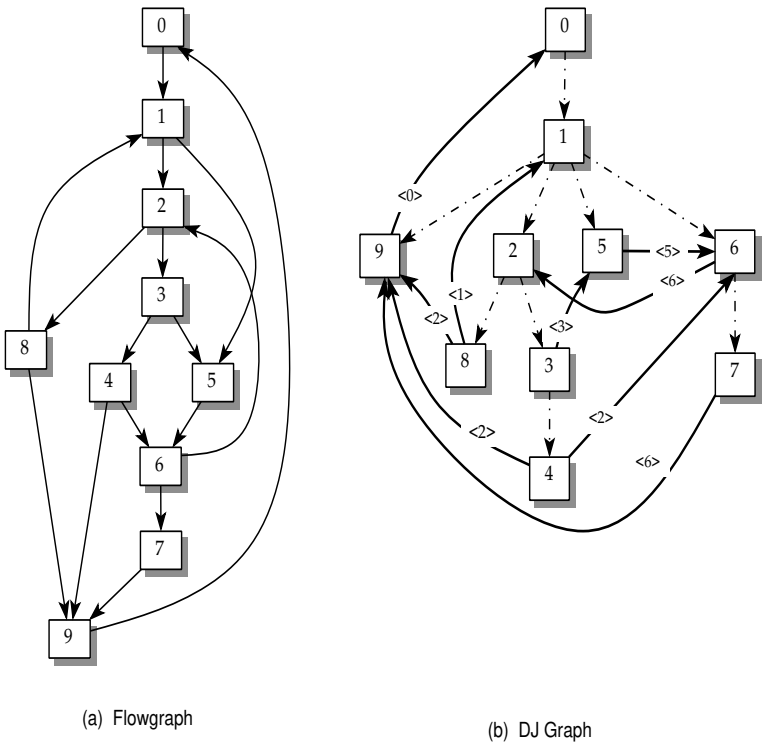

Figure 6: An irreducible flowgraph and its DJ graph.

The first step in handling irreducible graph is to apply D2b rule to all J edges whose source nodes are at level i and destination nodes are at levels less than i (step **88**). This will eliminate all such J edges that leave level i. The next step is to compress the dominator tree; for each node y at level i, and for each J edge $u \rightarrow w$ in y->$jedges$ we invoke **CompPath**(y, u) (step **92**).

The next step is to evaluate the equations of all nodes in level i. For this we first apply Tarjan's Strongly Connected Component (SCC) algorithm on nodes at level i (step **96**). This will generate dag(s) of SCCs. It is important to remember there can be more than one disjoint dag at level i. We process each SCC in each dag in topological order (step **97**). We compute closure of equations of all nodes in S, if needed (step **98**). We also express the flow equations at all nodes at level i in terms of flow variable of their immediate dominator node (step **99**). This is possible because we are eliminating the flow variables in the topological order of the SCC. Finally, we eliminate all the J edges at level i (step **100**).

Procedure CollapseIrreducible(i)

{ `/* See also the description in the main text */`

85: **foreach** node y at level i **do**

86: **foreach** $z \in Succ(y)$ **do**
87: **if**$((y \rightarrow z == Jedge)$ **and** $(z.level < i))$
88: **Delayed2b**$(y \rightarrow z)$.
89: **endif**
90: **endfor**
91: **while**$(((u \rightarrow w =$ **GetJedge**$(y)) \neq NULL))$ **do**
92: **CompPath**(y,u) ;
93: Eliminate O_u on the right-hand side of O_w by replacing it with the function of O_y computed in the previous step.
94: **endwhile**
95: **endfor**
96: Determine the SCCs of the nodes at level i, and construct dag(s) of SCCs;
97: Process each SCC S in each dag in the topological order as follows:
98: If the nodes in S induces a cycle then compute the closure of all equations of the nodes in the cycle. Also express the equation at each node in terms of its immediate dominator node.
99: If a node $s \in S$ has an edge $s \rightarrow t$ to node $t \notin S$ then replace O_s on the right hand side of O_t by the linear function of the input to O_s. Remember that O_s is expressed in terms of only the flow variable of its immediate dominator

```
/* After processing each SCC, the flow equation of  */
/*  each node at level i are expressed in terms of  */
/* its immediate dominator.    */
```

100: Finally remove all J edges at level i.

}

The complete trace of the DJ graph reduction for the irreducible DJ graph is shown in Figure 7. During the reduction when we reach to the DJ graph shown in Figure 7(g), we can notice that all nodes at level 2 are join nodes, and so we cannot apply any of the $\mathcal{D}$-$rules$. At this point we invoke **CollapseIrreducible**(2). In the procedure **CollapseIrreducible**() we first apply D2b rule by invoking **Delayed2b**$(9 \rightarrow 0)$. Next we compress the path $2 \rightarrow 3 \rightarrow 4$, expressing the flow equations at these nodes in terms of 2, and finally making all the nodes on this path children of 2.

Next we determine the SCC and process the nodes in the topological order. The processing steps consists of computing the closure whenever we have cycles, and expressing the flow equations of all the nodes in terms of their immediate dominator. Finally we will eliminate all the J edges at this level. After the call to **CollapseIrreducible**(i) terminates we get the DJ graph shown in Figure 7(h).

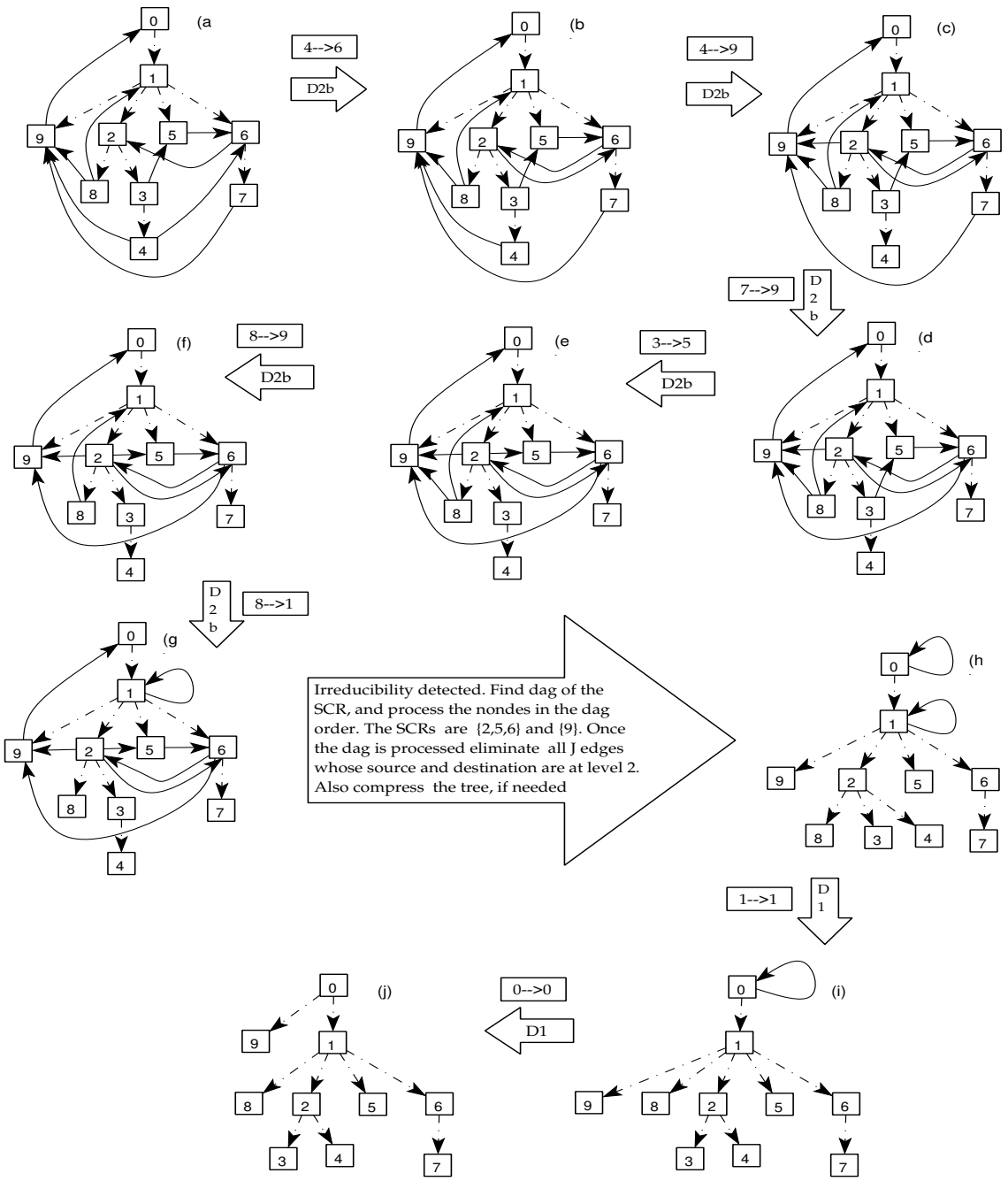

Figure 7: A trace of DJ graph reduction for the irreducible flowgraph.

8 Experiments and Empirical Results

In this section we present empirical results for both eager and delayed elimination methods. To demonstrate the effectiveness of our approach, we implemented both the methods for solving the intraprocedural reaching definitions problem. For the purpose of comparison, we also implemented the iterative method that uses a reverse postorder for iteration. Hecht and Ullman show that this ordering can be very efficient, especially for reducible flowgraphs, and can exhibit linear time complexity in practice [HU77].

For efficient set manipulation we implemented the Briggs-Torczon sparse sets [BT93]. We found that this representation to be more time efficient than the bit-vector representation. For this chapter we carried out our experiments on a SPARC-20 workstation.

We will first summarize the major results of our experiments.

- Of the 40 procedures we tested, five procedures have irreducible loops. Three of these five procedures have only one irreducible loop.
- The maximum size of SCC, found when Tarjan's SCC algorithm is applied during **CollapseIrreducible**(), is 4 (found in procedure `coef`). This suggests that our approach is indeed very efficient in practice for handling irreducible loops.
- As pointed out by Cytron et al. [CFR+91] the size of the dominance frontier relation is linear in practice. We found the average ratio $\frac{|DF_e|}{|E_f|} = 1.1$. This ratio suggests that the average size of dominance frontiers (represented as a set of edges) is about $\frac{1.1-1}{1.1} = 9.1\%$ more than the size of the flowgraph. This confirms to the claim made by Cytron et al. that size of dominance frontiers are linear in practice.
- As expected, the number of $\mathcal{E}$-*rules* rules applied is bounded by the size of the dominance frontier relation. We found the average ratio $\frac{nE}{|DF_e|}$ of the number of $\mathcal{E}$-*rules* applied nE to the size of dominance frontiers $|DF_e|$ to be 0.76. This suggests that eager elimination is expected to behave linearly in practice.
- As expected the number of $\mathcal{D}$-*rules* rules nD is less than the size of the flowgraph $|E_f|$. The average ratio $\frac{nD}{|E_f|} = 0.52$.
- We found that the average ratio of the total length of the dominance frontier interval path without path compression C' to the total length of the dominance frontier interval path with path compression C to be 1.3 (i.e, $\frac{C'}{C} = 1.3$). This ratio indicates that all the *dominance frontier interval paths* have about $\frac{1.3-1.0}{1.3} = 23.1\%$ of their edges overlapped.
- The value of $\frac{C'}{C} = 1.3$ also suggests that, in an ideal situation, delayed elimination method can be about 1.3 times faster than eager elimination method. In practice, delayed method incurs overhead from bookkeeping and nonprofitable path compressions.

This is evidenced by the data reported in Table 2, where we can see that, on average, delayed elimination method is about 1.1 times faster than its eager counterpart.

- All three algorithms (iterative, eager, and delayed) are very efficient in practice. The average number of iterations (performed during fixed-point calculation) in iterative method is 4, suggesting that iterative method is indeed very efficient (at least for solving reaching definitions problem).

- We find that delayed elimination method is on average about 1.4 times faster than iterative method, and eager elimination method is on average 1.3 times faster than iterative method.

- Finally, we find that eager elimination is almost as fast as the delayed elimination method. On average delayed method is only about 1.1 times faster than the eager elimination method.

In the following subsections, we will further elaborate on these results in two aspects: (1) the structural characteristics of our approach, and (2) the execution performance of our approach. Table 1 and Table 2 give a summary of our empirical results. The notation used in these tables are given below:

Notation used in Table 1 and Table 2

Name	Names of procedures.
$\|N\|$	Number of flowgraph nodes.
$\|E_f\|$	Number of flowgraph edges.
$\|G\|$	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
C'	Total length of the dominance frontier interval paths w/o path compression
C	Total length of the dominance frontier interval paths with path compression
$\frac{C'}{C}$	Ratio of C' to C
$\|DF_e\|$	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
$S_{i/e}$	$\frac{T_i}{T_e}$
$S_{i/d}$	$\frac{T_i}{T_d}$
$S_{d/e}$	$\frac{T_e}{T_d}$

8.1 Structural Characteristics

Table 1 shows the structural characteristics of our elimination methods for our test procedures. The second and third columns give the number of nodes (basic blocks) and the number of edges in the flowgraph for each procedure, respectively. The column $|G|$ gives the total number of downward exposed definitions in a procedure.[6]

The column Ir shows the number of times the procedure **CollapseIrreducible**() is invoked. Recall that **CollapseIrreducible**() is invoked only if irreducibility is detected at a particular level. The column S shows the maximum size of nontrivial strongly connected components detected and processed by the procedure **CollapseIrreducible**(). This column quantifies the number of nodes whose data flow equations are involved in fixed-point iteration. Our results indicate that the maximum size of SCCs is 4 for one procedure (`coef`), 3 for two procedures (`card` and `comlr`), and 2 for two procedures (`dcdcmp` and `readin`). This suggests that our approach is very efficient in practice for handling irreducible flowgraphs.[7] Previous approaches perform iteration over a normally much larger region when an irreducible region is encountered [Bur90].

The columns $ED1$ and $ED2a$ indicate the number of times E1 (D1) and E2a (D2a) rules were invoked during the eager (the delayed) elimination method. The number of E1 (D1) rules give the number of single-entry loops in a procedure. The column $E2b$ shows the number of E2b rules applied in eager elimination, whereas $D2b$ shows the number of D2b rules applied in delayed elimination. The column nE represents the sum of the three columns $ED1$, $ED2a$, and $E2b$, and nD represents the sum of $ED1$, $ED2a$, and $D2b$. Figure 8 and Figure 9, respectively, gives a profile of the number of $\mathcal{E}$-*rules* and $\mathcal{D}$-*rules* applied during the DJ graph reduction. We can see from these plots that the number of E2b rules applied dominates in the eager elimination method, that the number of D2a and D2b rules applied are not very different.

The column C' gives the total length of all the *dominance frontier interval paths* when no path compressions are performed. In comparison, the column C gives a similar number, but when path compressions are performed. Recall that the length of a dominance frontier interval path can progressively decrease after its overlapped paths are compressed. As we can see from the table that there is not much difference between C' and C. The ratio $\frac{C'}{C}$ gives an indication on how much improvement delayed elimination can achieve over eager elimination. The average ratio is 1.3, indicating that delayed elimination method should, ideally, be faster than eager elimination by a factor of 1.3 (if we ignore the overhead of bookkeeping and nonprofitable path compressions in delayed elimination).

In Figure 10 we plotted the size of dominance frontiers $|DF_e|$ along with C and C'. From the plot and the table we can that the length of the dominance frontier interval without

[6] We only consider scalar variables in our experiments.

[7] For backward flow problems, such as live uses of variables, we expect to see much more irreducible regions, since the analysis is performed on the reverse flowgraph.

Name	$\|N\|$	$\|E_f\|$	$\|G\|$	Ir	S	$ED1$	$ED2a$	$E2b$	nE	$D2b$	nD	C'	C	$\frac{C'}{C}$	$\|DF_e\|$
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.

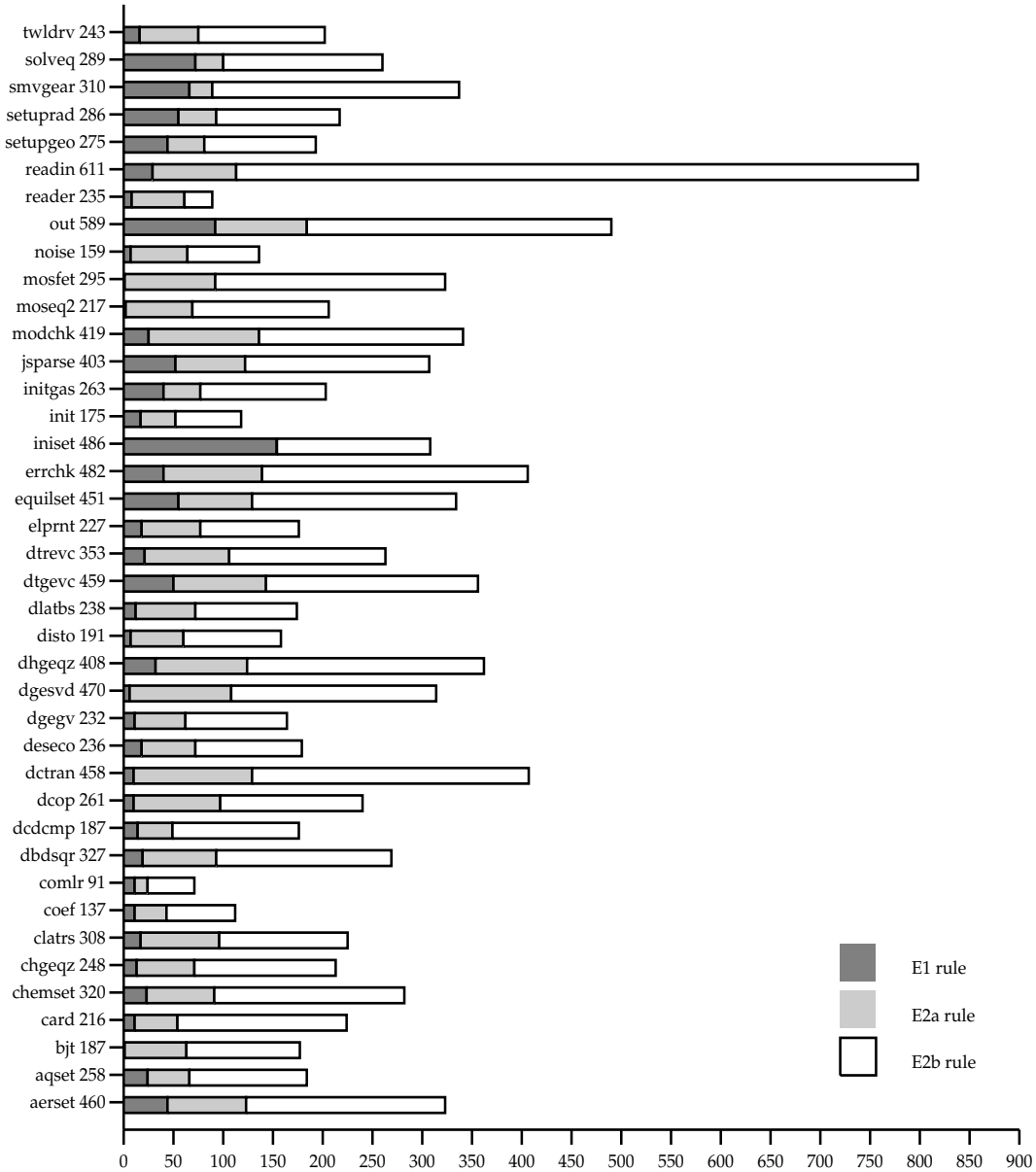

Figure 8: A profile of the number of $\mathcal{E}$-*rules* applied.

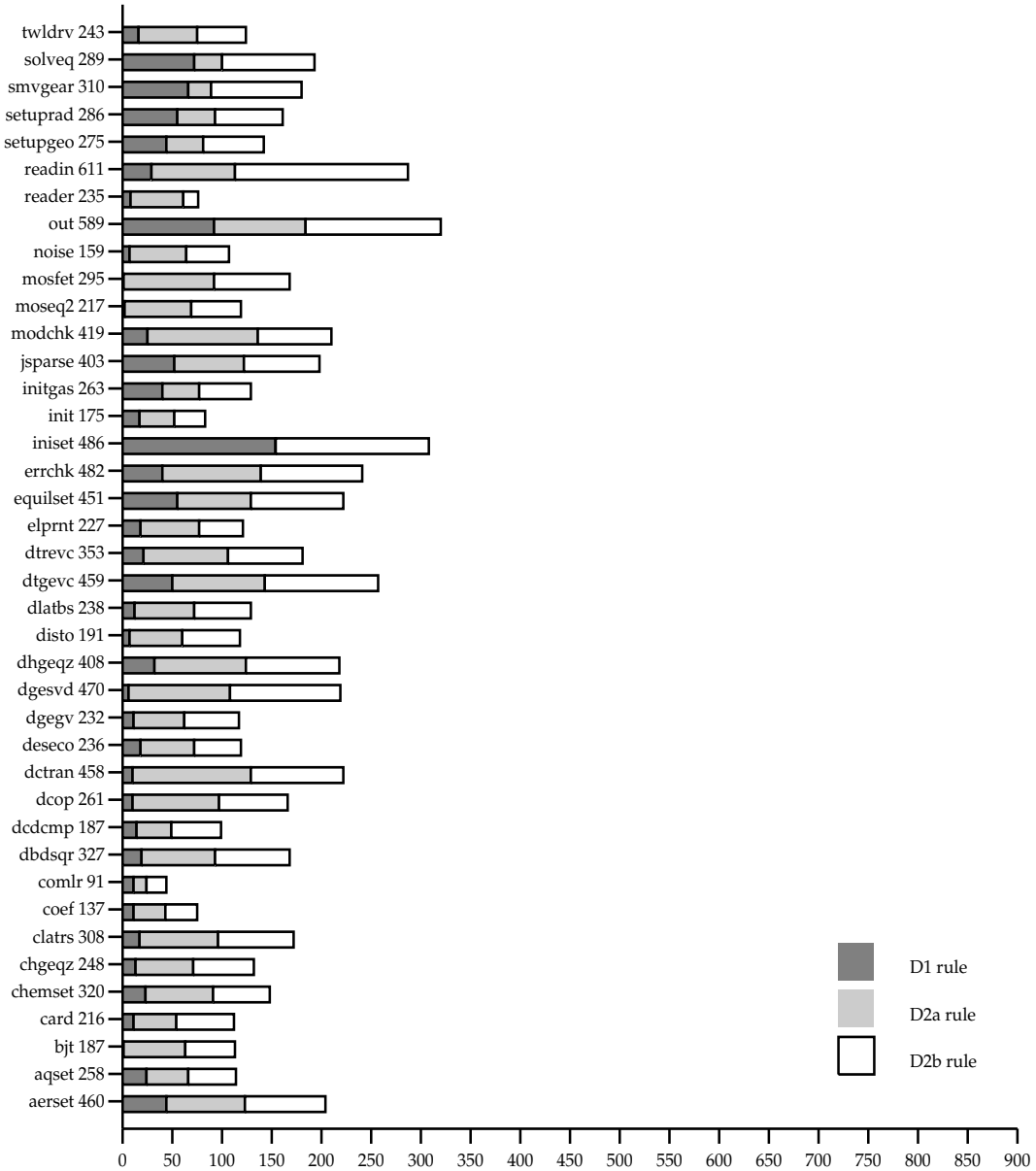

Figure 9: A profile of the number of $\mathcal{D}$-$rules$ applied.

compression, C', is less than the size of the dominance frontier relation, $|DF_e|$. In the plot we have also shown the sizes of dominance frontiers $|DF_e|$. As we can see from Table 1 and from the plot shown in Figure 10, the lengths C and C' are less than $|DF_e|$. From this result we can conclude that the time complexity of delayed elimination can be expected to be linear in practice. Also, from the table we can see that the number of $\mathcal{E}$-$rules$ applied (i.e., nE) is less than $|DF_e|$. From this, we can again conclude that the time complexity of eager elimination should also be linear in practice (since the size of dominance frontier is linear in practice). Finally, as expected, we can see that the total number of $\mathcal{D}$-$rules$ nD applied is bounded by the number of flowgraph edges $|E_f|$.

In delayed elimination, although each D2b rule takes only constant time, the cost of applying D1 or D2a rule includes the cost of the path compression incurred. The total number of path compressions performed is equal to the number of D2b rules applied, but each compression can take $O(\log(N))$ time (in the worst case). Therefore delayed elimination can suffer from the $\log(N)$ overhead factor due to a path compression. For the benefit of path compressions to be fully exploited, there must be enough overlapping paths, so that future path compressions could take less time. The number of dominance frontier interval paths passing through a node is bounded by the number of nodes in its dominance frontier. In other words, a node can participate in path compressions only as many times as the size of its dominance frontier set, which is a small constant in practice (we can use use $|DF_e|$ and $|E_f|$ from Table 1 to calculate $\frac{|DF_e|}{|E_f|}$ as an estimate). Therefore, we believe the benefit of path compressions will normally not be fully utilized for real programs.

As a final remark, we want to emphasize that the above observations are restricted to the context of intraprocedural analysis. We did not empirically investigate our approach for interprocedural analysis to quantitatively argue its behavior in practice.

8.2 Execution Performance

Table 2 gives the timing data from our experiments. The columns T_i, T_e, and T_d give the execution times in seconds for the iterative method, the eager elimination method, and the delayed elimination method, respectively. Figure 11 shows the execution behavior plotted against the number of flowgraph edges. As can be seen from the trend shown in the plot that all three procedures behave linearly for our test procedures. The columns $S_{e/i}$ and $S_{d/i}$ gives the speedups of eager method over iterative method and delayed method over iterative method, respectively.

The $S_{d/e}$ column gives the speedups of delayed over eager elimination method. Interestingly, the execution times of eager elimination method are competitive when compared with those of delayed elimination method. Eager elimination can even out perform delayed elimination in some cases. This is because there are not many overlapping paths in our test programs for delayed method to benefit from path compressions. Recall that each path compression takes time proportional to the path length. This cost is unnecessarily spent if there are no overlapping paths.

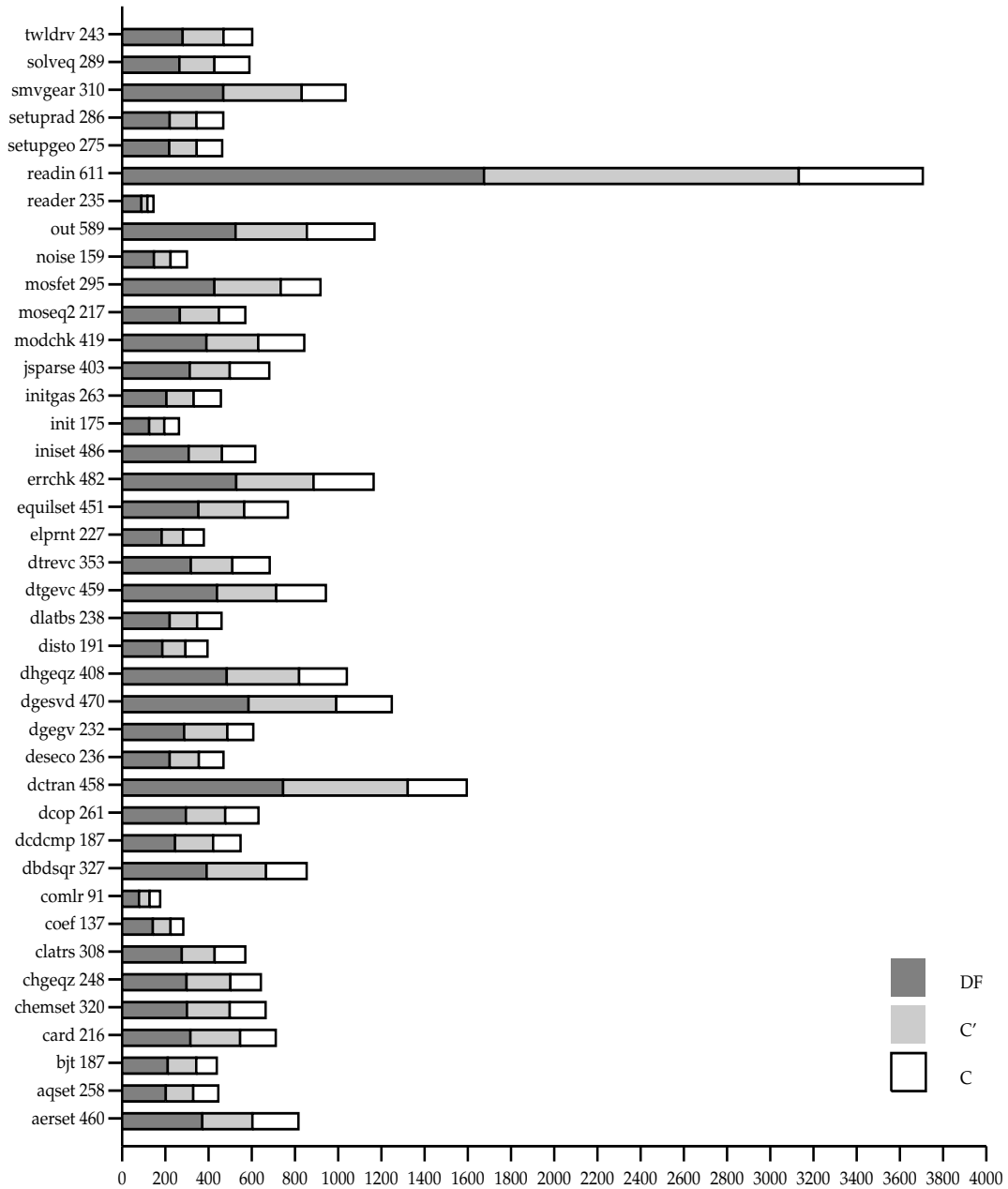

Figure 10: A comparison of $|DF_e|$ with C and C'.

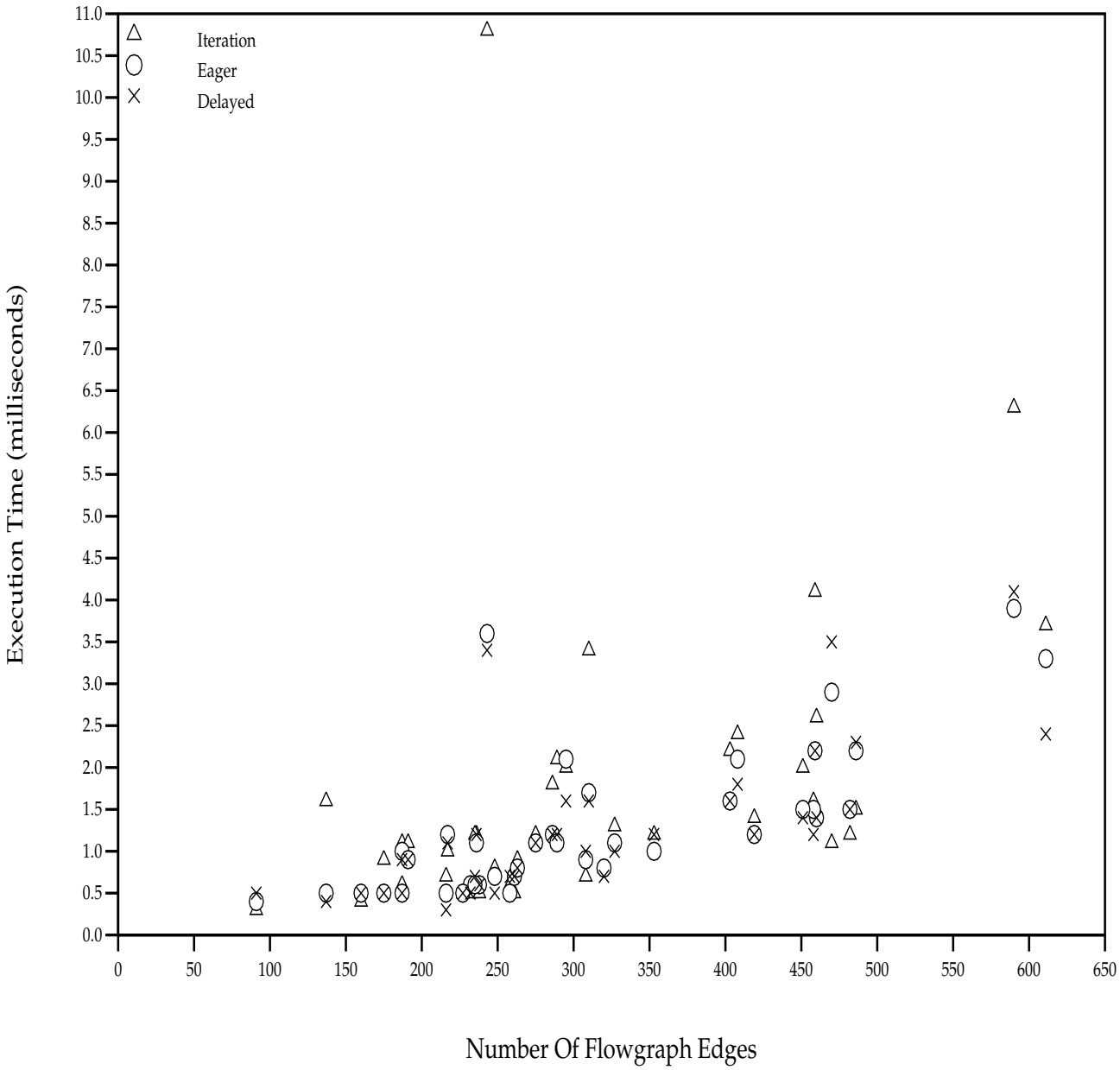

Figure 11: Execution profile of the three algorithms when executed on our test suites.

Theoretically speaking, eager elimination method is worse than delayed elimination method in terms of time complexity. However, our empirical results demonstrate that eager method is very competitive when compared with delayed method. From a pragmatic point view, we thus recommend that one implement eager method. Not only is it simple and easy to implement, but also it is amenable to incremental data flow analysis [SG95a].

We found several "outstanding" procedures when we examined the data reported in Tables 1 and 2. The first one is the procedure `iniset`: The sum of its $ED1$ and $ED2b$ numbers is equal to its $|DF_e|$ number. For this procedure 154 E1 (D1) rules were applied, suggesting that `iniset` contains 154 loops. We examined the procedure and found this to be true: It consists of 154 simple DO loops for initializing arrays. From Table 2, we can see that iterative method takes only two iterations to converge. By contrast, both eager and delayed elimination methods perform poorly for this procedure. Another interesting aspect for this procedure is that the ratio $\frac{C'}{C}$ is one, suggesting that **CompPath**() is never called for this procedure.

At the other extreme is procedure `twldrv`. This procedure is well-known for its complex control flow (although it does not have irreducible loops). As we can see, iterative method takes 9 iterations for solving the data flow equations. Both eager and delayed elimination methods perform much better than iterative method (by a factor of 1.3 on average).

Another interesting procedure is `readin`. This procedure also has complex control structure and, as expected, both eager and delayed elimination methods perform better than iterative method. The size of dominance frontiers is quite large compared to the number of flowgraph nodes or edges. The ratio $\frac{C'}{C}$ for this procedure is the largest among all the procedure, suggesting that delayed elimination method should perform better than eager elimination method. From the speedup measure, we can see that this is indeed true.

9 Discussion and Related Work

In this section we compare our work with other related work. Our work is related to all of the four classical elimination methods (the Allen-Cocke method, the Hecht-Ullman method, the Graham-Wegman method, and the Tarjan method), but with a number of significant differences. In [RP86] Ryder and Paull present a unified model to characterize a family of data flow analysis algorithms—elimination methods. The model is based on systems of data flow equations. They compare and contrast the four elimination methods. We encourage readers to consult this article for a comprehensive treatment of elimination methods.

Except for Graham-Wegman's analysis, 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 replicates certain nodes in a flow graph and generates an "equivalent," reducible flow graph [Hec77]. The other approach is to form improper regions to accommodate irreducibility and use fixed-point iteration in those regions [Bur90]. Our elimination method

Name	$\|E_f\|$	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.

takes the second approach but can 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.

Allen-Cocke's interval analysis was the first elimination method [AC76]. The Allen-Cocke method for forward data flow problems has two phases: elimination and propagation. In the elimination phase, it partitions the flow graph into intervals, summarizes data flow effects local to each interval on the global data flow solution, and collapses each interval into a single node.[8] It repeats the process until there is only one node left, and then easily solves the data flow problem on this node. In the propagation phase, it expands a node into an interval, and propagates global data flow information from the head node to internal nodes in the interval.

Any Allen-Cocke interval is a single-entry region; its head node dominates all the internal nodes in the interval. Thus, the data flow solution at each internal node can be expressed solely in terms of the solution at its head node. The worst-case time complexity for the Allen-Cocke method can be quadratic, and so is our eager substitution method.

In Tarjan's interval analysis [Tar74], an interval is a single-entry, strongly connected subgraph; by contrast, an Allen-Cocke interval need not be strongly connected. The Tarjan intervals, therefore, can reflect the loop structure of a program. His method carefully orders variable substitutions in a system of data flow equations. It delays some substitutions until a later time when common factors can be detected, calculated only once, and used. In [Tar81] Tarjan proposes two implementations for his approach. An almost linear time algorithm needs to use the balanced, path compressed trees. By contrast, our delayed substitution method is linear and performs compression on the dominator tree.

Hecht-Ullman's T1-T2 analysis uses single-entry regions to direct its elimination phase [Hec77]. Two transformations, T1 and T2, are repeatedly applied to a reducible flow graph until it is collapsed into one single node. The sequence of T1 and T2 operations applied is called a parse of the flow graph. The Hecht-Ullman method uses this parse to guide the elimination phase. Data flow information is summarized in some region at each step of the parse. In the propagation phase, the reverse order of the parse is used and data flow information is propagated within some region.

Our D1 rule is exactly the same as T1; our D2 rules are similar to T2. However, our D2b rule is equivalent to a sequence of T2 rules. To achieve the $O(|E|\log(|N|))$ time bound, the Hecht-Ullman method uses a height-balanced 2-3 tree to assist delayed substitutions of variables in data flow equations. This data structure is more complicated than our compressed dominator tree. In addition, the Hecht-Ullman method needs an explicit parse to guide its elimination and propagation. By contrast, we do not need to keep track of the order in which our reduction rules are applied, since our approach does its propagation on the dominator tree (which may be compressed) in a top-down manner.

[8] For backward data flow problems, the flow graph partitioning will be performed on the reverse flow graph.

Graham-Wegman's analysis uses graph reduction rules similar to those in the Hecht-Ullman method, whereas its groupings of data flow equations are similar to those in Tarjan's interval analysis [GW76]. It partitions the flow graph nodes into non-disjoint sets called S-sets, which are analogous to the Tarjan intervals. However, 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 substitutions are remembered in a reduced flow graph. Our path compression is comparable to that in Graham-Wegman method. During their T2′ rule they have to inspect which outgoing edges of a node are within the current S-set (in Graham-Wegman method, the S-set is a strongly connected region). Therefore the time complexity of their T2′ rule depends on the number of loop exist and so is non-linear. In our case, D2b rule eliminates such edges so we never need to 'inspect' any J edges during path compression (in Graham-Wegman method T2′ does path compression on the depth first spanning tree).

For reducible flow graphs, the Graham-Wegman method uses graph reduction rules S1 and S2 to collapse nodes within an S-set. If necessary, S3 is used to reduce the final graph to one node. S1 is the same as our D1; S2 is similar to our D2 rules; and S3 is similar to our D3. These operations can be generalized to GS1, GS2, and GS3 to handle irreducibility caused by multiple-entry regions. For reducible flow graphs, the worst-case time complexity of Graham-Wegman's analysis is $O(|E|\log(|E|))$.

Elimination methods are general-purpose data flow solution procedures [Mar89, Bur90, Ros81, Tar81]. All the above classical elimination methods are formulated and discussed with fast problems. In [Bur90] Burke reformulates Tarjan's interval analysis so that it can be applied to any monotone data flow problem. The loop-breaking rule used by Ryder and Paull in [RP86] is only valid for fast problems. 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 D2'b rule. We suspect that this coincidence may partially explain why our method can handle irreducibility gracefully.

10 Conclusion

In this paper we have presented a new framework for elimination-based data flow analysis using DJ graphs. By utilizing the properties of DJ graphs, we can handle irreducible flowgraphs efficiently. There are two variations of our approach: (1) eager elimination method and (2)

delayed elimination method. We proved their correctness and analyzed their complexity. We also implemented both methods and experimented with them on 40 FORTRAN procedures from various sources. Although eager elimination method is theoretically worse than its delayed counterpart, our preliminary empirical results suggest that it is competitive in running time when compared to delayed elimination method. Not only is it simple and easy to implement, but also it is amenable to incremental data flow analysis [Sre95].

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 support. The first author would like to thank Shamir Merali for listening and participating in many insightful discussions on path compression. Finally, we thank Adam Buchsbaum for his pointing out an error in one of our original proofs and directing us to some key results on path compression.

References

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

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

[BST95] Buchsbaum, A. L., Sundar, R., and Tarjan, R. E. Data structural bootstrapping, linear path compression, and catenable heap ordered double ended queues (*to appear*). *SIAM Journal of Computing*, 42(5), October 1995.

[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.

[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.

[HU74] Hecht, M. S., and Ullman, J. D. Characterizations of reducible flow graphs. *Journal of the ACM*, 21(3):367–375, July 1974.

[HU77] Hecht, M. S., and Ullman, J. D. A simple algorithm for global data flow analysis problems. *SIAM Journal of Computing*, 4(4):519–532, December 1977.

[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.

[PB95] Pingali, K., and Bilardi, G. $\mathcal{APT}$: A data structure for optimal control dependence computation. In *Proceedings of the SIGPLAN'95 Conference on Programming Language Design and Implementation*, 1995.

[PGH+91] Polychronopoulos, C., Girkar, M. B., Haghighat, M. R., Lee, C. L., Leung, B. P., and Schouten, D. A. The structure of Parafrase-2: An advanced parallelizing compiler for C and FORTRAN. In Nicolau, A., Gelernter, D., Gross, T., and Padua, D., Eds., *Advances in Languages and Compilers for Parallel Processing*, pages 423–453, London, England, and Cambridge, Massachusetts, 1991. Pitman and the MIT Press. Selected papers from the Third Workshop on Languages and Compilers for Parallel Computing, Irvine, California, August 1–3, 1990.

[Ros80] Rosen, B. K. Monoids for rapid data flow analysis. *SIAM Journal of Computing*, 9(1):159–196, February 1980.

[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.

[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.

[SGL95] 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.

[SS79] Schwartz, J. T., and Sharir, M. A design for optimizations of the bitvectoring class. Technical report, Courant Institute of Mathematical Sciences, New York University, September 1979. Courant Computer Science Report No. 17.

[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

In this section we introduce some backgraound material and notation that will be used in the paper.

A.1 Flowgraph Terminology

A flowgraph is a connected directed graph $G = (N, E, \text{START}, \text{END})$, where N is the set of nodes, E is the set of edges, START $\in N$ is a distinguished start node, and END $\in N$ is a distinguished end node. We represent program texts using flowgraphs, where nodes represent basic blocks and edges flow of control. For forward problems (such as Reaching Definitions) flowgraph is the forward control flow graph with START representing program entry point and END representing program exit point. For backward problems (such as Live Uses of Variables) flowgraph is the reverse control flow graph, with START representing program exit point and END representing program entry point. Figure 1(a) shows an example flowgraph.

In a flowgraph, if $x \rightarrow y \in E$, then x is called the *source* node and y is called the *destination* node of the edge. We define $Succ(x)$, the successor nodes of x, to be all nodes y such that $x \rightarrow y \in E$. Similarly we define $Pred(x)$, the predecessor nodes of x, to be all nodes y such that $y \in x \in E$.

A *path* of length n is a sequence of edges $(x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_n)$, where each $x_i \rightarrow x_{i+1} \in E$. We will use $P : x_0 \xrightarrow{+} x_n$ to denote a path of length one or more. We will also assume that for every node $x \in N$ there exists a path from START to x, and similarly there is a path from x to END.

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

In a flowgraph, a node x **dominates** another node y iff all paths from START to y pass through x. We write x **dom** y to indicate that x dominates y. The dominance relation is reflexive and transitive, and can be represented by a tree, called the dominator tree. If x is a parent node of y in the dominator tree, then x **immediately dominates** y, and we write $idom(y)$ to denote the immediate dominator of y. Figure 1(b) shows the dominator tree for the flowgraph shown in Figure 1(a). We associate a *level number* with each dominator tree node. The level number of a node x, $x.level$, is defined to be its depth from the root.

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

A.2 Data Flow Analysis

Data flow analysis is a process of collecting facts about a program from its intermediate representation. These facts or data flow information can be modeled by elements of a semi-lattice L. Associated with each node x is a flow function f_x that maps input information to output information [Kil73, Mar89].[9]

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

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

$$O_x = f_x(I_x)$$

We can rewrite this equation to be as follows:

$$O_x = f_x(I_x) = I_x \cap P_x \cup G_x \tag{3}$$

where $P_x, G_x \in L$.[10] We can interprete the above equation as follows: The Output flow information at a node's exit is either (1) what is Generated within the node or (2) what arrives at its Input and is Preserved through the node. For convenience, in this paper we will use the following notation instead of the notation in Equation 3.

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

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 $y \rightarrow x$ exists. The input information I_x is the merge of all the output information O_y of nodes y in $Pred(x)$; i.e.,

$$I_x = \bigwedge_{y \in Pred(x)} O_y \tag{5}$$

$\bigwedge$ is usually a union or intersection operation, depending upon the data flow problem being solved.

Combining Equations 4 and 5, we obtain the following equation, denoted H_x, for each $x \in N$.

$$\begin{aligned} H_x : O_x &= f_x(\bigwedge_{y \in Pred(x)} O_y) \\ H_x : O_x &= P_x(\bigwedge_{y \in Pred(x)} O_y) + G_x \end{aligned} \tag{6}$$

Since there is one equation for each node, we have a total of $|N|$ equations.

Equation 6 has two variations: The first variation is more general than the second, 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

[10] In general G_x may not be a constant. 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.

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 compute the solution of O_x. When we examine the initial set of equations, H_x depending on H_y also indicates that there exists an edge $y \rightarrow x$ in the flowgraph.

Now we will discuss the concept of *variable elimination*, which is fundamental to all elimination methods. Consider the following two equations for example:

$$\begin{aligned} H_y : O_y &= P_y(\bigwedge_{z \in Pred(y)} O_z) + G_y \\ H_x : O_x &= P_x O_y + G_x \end{aligned}$$

In this example, equation H_x depends only on H_y. This also means that x has only one incoming edge $y \rightarrow x$. To eliminate the variable O_y from the RHS of H_x, we can replace it with the RHS of H_y. The resulting H_x equation thus becomes:

$$H_x : O_x = P_x(P_y(\bigwedge_{z \in Pred(y)} O_z) + G_y) + G_x$$

Here we have eliminated the dependence of H_x on H_y but introduced dependences from H_z to H_x for each predecessor of y. In the corresponding flowgraph we also eliminate the edge $y \rightarrow x$ and introduce an edge $z \rightarrow x$ for each $z \in Pred(y)$.

For the more general variation in Equation 6, variable elimination corresponds to function composition. To illustrate this, let us consider the following equations:

$$\begin{aligned} H_y : O_y &= f_y(\bigwedge_{z \in Pred(y)} O_z) \\ H_x : O_x &= f_x(O_y) \end{aligned}$$

After eliminating O_y in H_x we get

$$H_x : O_x = f_x(f_y(\bigwedge_{z \in Pred(y)} O_z))$$

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

$$H_y : O_y = mO_y + k = f(O_y), \tag{7}$$

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

$$H_y : O_y = f^*(O_y) \tag{8}$$

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 semi-lattice does not contain an infinite descending chain [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].

Example: Consider the flowgraph shown in Figure 1. The data flow equation for each node is summarized in the table below.

x	f_x	O_x	$I_x = \wedge_{y \in Pred(x)} O_y$	$f_x(I_x) = O_x = P_x I_x + G_x$
0	f_0	O_0	O_9	$P_0 O_9 + G_0$
1	f_1	O_1	$O_0 \wedge O_8$	$P_1(O_0 \wedge O_8) + G_1$
2	f_2	O_2	$O_1 \wedge O_6$	$P_2(O_1 \wedge O_6) + G_2$
3	f_3	O_3	O_2	$P_3 O_2 + G_3$
4	f_4	O_4	O_3	$P_4 O_3 + G_4$
5	f_5	O_5	O_3	$P_5 O_3 + G_5$
6	f_6	O_6	$O_4 \wedge O_5$	$P_6(O_4 \wedge O_5) + G_6$
7	f_7	O_7	O_6	$P_7 O_6 + G_7$
8	f_8	O_8	O_2	$P_8 O_2 + G_8$
9	f_9	O_9	$(O_4 \wedge O_7 \wedge O_8)$	$P_9(O_4 \wedge O_7 \wedge O_8) + G_9$

Table 3: Initial data flow equation at each node.

B A Linear Algorithm for Computing Top Nodes

Algorithm B.1 below computes the **Top** nodes for all J-edges. The algorithm works on the DJ graph. We will use the following notation and data structure to simplify the presentation.

- $Children(u)$ denotes the children of u on the dominator tree.
- Each node in the DJ graph has the following attributes:
  - curChild `/* Current child node visited in the depth first traversal */`

– cand /* Candidate **Top** node for this **Bottom** node in the closed interval $[thisbottom, cand]$. */

Algorithm B.1 *The following algorithm determines the* **Top** *nodes for a given set of open interval* $[y, w)$.

♣ **Input:** The DJ graph.

♣ **Output:** The **Top** node x of the closed interval $[y, x]$, stored in $y.cand$.

MainTop()
{

101: **foreach** $c \in Children$(START)
102: START.$curChild = c$
103: **TopDFS**(c).
104: **endfor**

}

TopDFS(y)
{

105: **foreach** outgoing J-edge $y \rightarrow z$ **do**
106: $y.cand = idom(z).curChild$;
107: **endfor**
108: **foreach** $c \in Children(y)$
109: $y.curChild = c$ /* possibly a **Top** node of some closed interval */
110: **TopDFS**(c).
111: **endfor**

}