National Library of Canada
Acquisitions and Bibliographic Services Branch
395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale du Canada
Direction des acquisitions et des services bibliographiques
395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

Canada

EFFICIENT PROGRAM ANALYSIS USING DJ GRAPHS

by

Vugranam C. Sreedhar

School of Computer Science
McGill University, Montréal
Québec, Canada
September 1995

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

ISBN 0-612-08158-3

Abstract

Program analysis is a process of estimating properties of a program statically. Program analyses have many applications, including compiler optimizations, software maintenance and testing, and program verification. In this dissertation we present a new framework for efficient program analysis. At the heart of our approach is a new program representation called the **DJ Graph**. Using DJ graphs we present several new algorithms for solving problems encountered in program analysis. The problems that we have solved range from a simple loop identification problem to sophisticated exhaustive and incremental data flow analysis, including the construction of Sparse Evaluation Graphs. The algorithms presented here are simple, more general, and/or more efficient than existing methods for solving similar problems. To study the effectiveness of our algorithms on real programs we implemented many of them, and experimented on a number of FORTRAN procedures taken from standard benchmark suites. Our results indicate that the algorithms presented here perform well in practice.

Résumé

L'analyse de programmes est un processus utilisé pour déterminer de façon statique les propriétés d'un programme. Les analyses de programmes ont de nombreuses applications, e.g., optimisations dans un compilateur, entretien et tests des logiciels, vérification de programmes. etc. Dans cette thèse, nous présentons une nouvelle approche pour obtenir des analyses de programmes performantes. Au coeur de notre approche est l'utilisation d'une nouvelle représentation des programmes appelée **Graphes DJ**. Nous présentons plusieurs nouveaux algorithmes utilisant les graphes DJ et permettant de résoudre de nombreux problèmes rencontrés dans le cadre d'analyses de programmes. Les problèmes que nous avons résolus vont d'un simple problème d'identification des boucles au problème plus complexe d'analyse incrémentale du flux des données, y compris la construction de graphes creux d'évaluation (*Sparse Evaluation Graphs*). Les algorithmes présentés sont simples, plus généraux et/ou plus efficaces que les méthodes généralement utilisées pour résoudre des problèmes similaires. Dans le but d'étudier l'efficacité de nos algorithmes sur des programmes réels, plusieurs d'entre eux ont été mis en oeuvre et exécutés sur un ensemble de procédures FORTRAN provenant de programmes tests standards. Nos résultats indiquent que les algorithmes présentés, en pratique, fonctionnent efficacement.

Acknowledgements

When a man thinks of objects, attachement from them arises. From attachment arises desire; from desire arises wrath. From wrath arises delusion; from delusion, failure of memory; from failure of memory, loss of conscience; from loss of conscience he is utterly ruined

—Bhagavad Gita

Even if you have knowledge, do not disturb the childlike faith of the ignorant. On the other hand, go down to their level and gradually bring them up.

—Bhagavad Gita

First of all I sincerely thank my father whose constant encouragement and spiritual support helped me to withstand the trials and tribulations of my Ph.D. research. It was through his spiritual guidance and blessings that I was able to find my research problem and solve it. I wish he was here to see my success. I dedicate this dissertation in memory of my father and my mother.

Next, I thank my lovely wife Lakshmi, who was instrumental to my well-being, especially after my father left me an orphan. I married her immediately after my comprehensive exam, and she has suffered the pain that I brought her during my research phase. Her constant encouragement and support gave me strength to finish my Ph.D. I also dedicate this dissertation for her love and support.

I also thank my in-laws for constantly praying for our well-being and hoping that one day I will provide a comfortable life for their daughter. I also dedicate this dissertation for their love and prayers. Next I thank my brothers and my sister for their support and encouragement.

Next I express my deepest debt of gratitude to my special friend Maryam Emami who constantly encouraged me throughout my stay in Montreal. Without

her help I would not have passed the comprehensive exam. She is one of the best friend I have in Montreal.

I wish to thank Peggy Gao for many parties she hosted and whose constant presence enlightens the ACAPS lab. I wish to thank my friends Shashank, Ravi, Govind, Bhama, and Justiani for many good times we had. I also thank Shashank also for listening to my woes during my depressing times. Next I wish to thank a special couple Russel and Yoshiko for the good times we had, and especially Yoshiko's presence which lit the whole atmosphere of the ACAPS lab. The lab was never the same after they left. Next I wish to thank my other friends of the lab like Erik Altman, Nasser Elmasri, Luis Lozano, Dhrubajyoti Goswami, Andres Marquez, Xinan Tang, Zhu Yingchun, Kevin Theobald, Shamir Merali, and Qi Ning with whom I had many interesting discussions, and Kevin Theobald for maintaining the ACAPS documents which helped me to channel my work professionally. I also thank Russel Olsen and Shamir Merali for carefully proofreading my dissertation and providing comments that greatly improved the presentation. I wish to thank Guy Tremblay for translating the abstract to French version, and also for commenting on a paper, which eventually became part of this dissertation. I wish to thank Priyadarshan Kolte (of OGI) for providing me with some experimental results which have been included in this disseration for the purpose of comparison.

Next I wish to thank our department staff members: Lise, Franca, Vicki, and Lorraine, for their administrative help throughout my stay at McGill. I also wish to thank our systems people: Luc, Kent, Ramesh, Matthew, Maryam (Alavi), and other older generation staff members for their patience and help throughout my stay at McGill University.

I wish to thank Prof. Laurie Hendren for her partial financial support during the early phase of my research. I also thank Prof. Nathan Friedman, Prof. Laurie Hendren, and Prof. Monty Newborn for serving on my proposal committee, and directing me in the right direction towards the completion of my dissertation. Next, I wish to thank my Ph.D. oral defense committee members: Prof. Luc Devroy, Prof. Tim Merrett, Prof. Nathan Friedman, Prof. Laurie Hendren, Prof. Ted Symanski, and Prof. John Trischuk, for their comments and suggestions that improved the overall presentation of the dissertation. I also wish to thank Prof. Keshav Pingali for serving as my external examiner and critically

evaluating my dissertation, and also for recommending me for Dean's honor.

Next I wish to thank Yong-fong Lee for his help and support. He was the one who first believed the importance of my work. I was honored by his willingness to collaborate and dedicate his precious time to help me in my research. I a' ɔ thank him for being very patient during my frustrating moments. I was also lucky to be helped and advised by many other peers. In particular, I thank G. Ramalingam, Richard Johnson, Bjarne Steensgaard, Adam L. Buchsbaum, Prof. Rajiv Gupta, Prof. Keshav Pingali, and Dr. Barry Rosen for commenting on various drafts of the papers which eventually became part of this dissertation.

A Ph.D. research can never be completed without help and support from your advisor. I sincerely thank my advisor Dr. Guang Gao. His moral, social, and financial support helped me to finish up my Ph.D. I also thank him for giving me the opportunity to work independently. One thing that encouraged me to work hard is seeing my advisor work. I have never seen any one (except my father) that dedicated to his work and research. It is an honor to be his student—an honor which I will proudly cherish throughout my entire life. I also dedicate this dissertation for his hard work and dedication to research, and for supporting me during my research.

Next I wish to thank my sponsoring agencies: the National Sciences and Engineering Research Council (NSERC) and the Canadian Centers of Excellence (IRIS) for financially supporting me through my Ph.D. research.

Finally, I thank the Lord for all the blessings that gave me the strength to move forward in my life.

Dedication

Dedicated to the memory of my

beloved father and mother

Also dedicated to my

loving wife Lakshmi

Contributions

Each work has to pass through these stages— ridicule, opposition, and then acceptance.

—Swami Vivekananda

The major contributions of this dissertation are as follows:

- We introduce a new representation called the **DJ graph** for efficient program analysis. We also explore some of the properties of DJ graphs that simplify many proofs (Chapter 3).
- We present simple linear time algorithms for computing dominance frontiers and related sets (Chapter 4).
- We present an efficient and a low polynomial time algorithm for computing the multiple node immediate dominance relation for an arbitrary flowgraph (Chapter 5).
- We present a simple and an efficient algorithm for detecting both reducible and irreducible loops in a flowgraph (Chapter 6).
- We present a simple linear time algorithm for computing iterated dominance frontiers (Chapter 7).
- We present a simple and an efficient algorithm for incrementally updating the dominator tree of an arbitrary flowgraph, when the flowgraph is subjected arbitrary incremental changes, including those that introduce irreducibility (Chapter 8).
- We present a simple and an efficient algorithm for incrementally updating the dominance frontier relation of an arbitrary flowgraph, when the flowgraph is subjected to arbitrary incremental changes, including those that

introduce irreducibility(Chapter 9).

- We present a new framework for elimination-based data flow analysis. Within this framework we present two methods for data flow analysis: (1) an eager elimination method, and (2) a delayed elimination method. The two methods presented are simple, efficient, and can handle arbitrary flowgraphs (Chapter 10).

- We present a simple and an efficient algorithm for incrementally updating data flow solutions, when the corresponding flowgraph is subjected to incremental changes, including those that introduce irreducibility (Chapter 11).

- Finally, to demonstrate the effectiveness of our algorithms we implemented most of them, and tested them on real programs. We present empirical results of our experiments and give their analysis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Great spirits have always found violent opposition from mediocrities. The latter cannot understand it when a man does not thoughtlessly submit to hereditary prejudices but honestly and courageously uses his intelligence.

—Albert Einstein

My grandfather once told me that there are two kinds of people: those who work and those who take the credit. He told me to try to be in the first group; there was less competition there.

—Indira Gandhi

This dissertation is about *efficient program analysis* using a new representation called the **DJ graph**. Using DJ graphs we have solved a number of problems encountered in program analysis. Our solution methods are simple, efficient, and/or more general (i.e., can handle both reducible and irreducible flowgraphs) than existing methods. We begin the dissertation by introducing the concept of program analysis in Section 1.1. Then, in Section 1.2, we briefly introduce DJ graphs. A more thorough presentation on DJ graphs is given in Chapter 3. In Section 1.3, we highlight the major contributions of the dissertation, and also briefly discuss some of the related work. Finally, in Section 1.4, we give the overall organization of the dissertation.

1.1 Program Analysis

1.1.1 Motivation

Program analysis is a process of estimating properties of programs at each program point [ASU86]. The information provided by a program analysis is useful in compiler optimization, code generation, program verification, testing and debugging, and parallelization [ASU86]. To understand the concept of program analysis consider the following program written in a Pascal like language.

```
Program FOO();
   var
      i,  x, y, z, w: integer ;
      sum: array[0..100] of integer;

0:  begin
1:    read(y, z) ;
2:    x := 2 ;
3:    if y > 0 then
4:    begin
5:      i := 1 ;
6:      while (i < 100) do
7:      begin
8:        if((i mod 2) = 2)
9:          sum[i] :=  (z div y) + w ;
10:       else
11:         sum[i] := (z div y) - 2 ;
12:       i : = i + 1 ;
13:     end
14:     write(`sum initialized`) ;
15:   end
16:   else
17:     writeln(`Program Error: y should be > 0`) ;
18:   writeln(`Program terminated`) ;
19: end.
```

In the above program we can observe the following:

1. The variable w at statement 9 is used before being defined. This may cause a run-time error. This problem can easily be detected at compile time using a classical program analysis technique, called the reaching definitions analysis [ASU86].[1] In this analysis, all variables at the beginning of a program are initialized to an 'undefined' value. Then we check whether an undefined value of a variable reaches its use. If it does, then it is an error. For the above program, we see that an undefined value reaches the use of w at statement 9, and therefore is an error.

2. In statement 2, the definition of the variable x is never used by any other statement. In the literature statement 2 is called a *dead* statement, and should be eliminated at compile-time. By eliminating dead statements (*á la dead-code elimination* [ASU86]), we can speed up the overall execution of a program.

3. The expression `z div y` is computed each time inside the loop. The value of this expression does not change as the `while` loop iterates, and so is invariant inside the loop. Hoisting invariant expressions outside a loop decreases the execution time of the program. In the literature this is called *loop-invariant removal* [ASU86].

4. In the above program, the value of the predicate `y>0` at statement 3 will determine whether statements from 5 to 14 will be subsequently executed or not. In other words, these statements are said to be "control dependent" on the condition at statement 3 [FOW87]. The control dependence relation can again be derived at compile-time via program analysis.

In each of above cases we are interested in certain properties of the given program. These properties are useful in program optimization, testing and debugging, code generation, etc. In general, program analysis can be divided into two types: (1) *control flow analysis*, and (2) *data flow analysis* [ASU86]. Control flow analysis is concerned with estimating properties related to program or control flow structure, for example, computing the control dependence relation. On the other hand, data flow analysis is concerned with estimating properties related

[1] A *definition* in a program is a statement or an instruction that assigns or may assign a *value* to a variable (or memory location) [ASU86].

to program data or variables, for example, computing reaching definitions set or removing loop-invariants.

1.1.2 Program Representation, Analysis Methods, and Solution Techniques

Both control flow analysis and data flow analysis are usually performed on a graph representation of a program called the Control Flow Graph (CFG).[2] The nodes in a CFG represent basic blocks or statements, while edges of the graph represent flow of control from one basic block to another.[3] Figure 1.1 illustrates the CFG for the program `FOO()`.

In the literature, control flow problems are typically solved using concepts from graph theory, whereas data flow problems are typically solved using concepts from set theory (or more precisely, lattice theory) [ASU86]. An example of a control flow analysis is computing the dominance relation. Given a CFG, a node R is said to dominate another node S if all paths from START to node S always pass through node R. One classical approach for computing the dominance relation is due to Lengauer and Tarjan [LT79]. Their algorithm is based on a depth first search of the CFG. The algorithm computes the dominance relation by searching for the nearest common ancestor of all the predecessor nodes of a node in the depth-first spanning tree of the given CFG [LT79].

An example of a data flow analysis is the reaching definitions problem. In reaching definitions the problem is to determine which definitions in a program reach a given point. The first step in solving the problem is to represent each definition as an element of a set S.[4] At the beginning of the program the set S is empty, meaning that no definitions reach the start of the program. This set is passed through every statement (or basic block) in the program, and whenever we pass through a definition d we first remove (kill) all previous definitions of the same variable from the set S, and then add the new definition d to the set. We

[2]For structured programs one can perform data analysis on the Abstract Syntax Tree representation of the program [ASU86].

[3]Many modern optimizing compilers analyze programs at two levels: (1) within a procedure, called the *intraprocedural analysis*; and (2) across procedures, called the *interprocedural analysis* [ASU86]. CFGs are used for representing statements (or basic blocks) and flow of control within a single procedure, whereas *call graphs* are used for representing procedure calling sequence. Whenever we use the term flowgraph of a program we mean flowgraph of a procedure.

[4]More precisely as an element of a *lattice*.

repeat this process until no more definitions are added or removed from the set. We also store a copy of the set S at each program point as we propagate through them. The set S is generally called as the *data flow information*, and the effect of a statement (or basic block) on S is called as the *data flow function*.

We can generalize the above discussion and represent any data flow problem within a framework called the *data flow framework*, first proposed by Kildall, and subsequently extended by others [Kil73, Hec77, Mar89]. Within this framework we represent data flow information as elements of a lattice, and the effect of a node (a statement or a basic block) as a data flow function. The input-output effect of a node can be represented as a data flow equation, and so we can set up a *system of data flow equations*, one equation per node, whose consistent solution gives the desired estimate of the program property [Kil73, Hec77, Mar89].

The methods for solving the system of equations can be broadly classified into three categories [ASU86]: (1) iteration methods, (2) elimination methods, and (3) syntax-directed methods. Iteration methods are very general and are applicable to all types CFG structure [ASU86, KU76, Kil73]. In iteration methods, we first initialize the flow information at each node to some *safe* value (i.e., a *safe* initial guess [Mar89], page 29), and then iterate through each node, applying the flow function, until a fixed-point is reached. This method for solving the system of equations is not very efficient, but is simple to implement. Elimination methods, first proposed by Allen and Cocke [AC76], are derived from the Gaussian elimination method for solving simultaneous equations. The key idea in all elimination methods is to reduce the system of equations to a "reduced" system of equations, and then solve the reduced system of equations. Elimination methods are asymptotically faster than iteration methods, but are more complex to implement than iteration methods. Finally, syntax-directed methods are applicable to programs that contain no arbitrary `goto` statements.[5] Both elimination based methods and iteration based methods use CFGs for propagating data flow information, whereas syntax-directed methods use Abstract Syntax Tree for propagating data flow information [ASU86].

Each of the above solution methods can also be characterized as (1) exhaustive, (2) incremental, or (3) demand-driven [MR90a, Mar89]. In exhaustive methods,

[5] If the source program contains `goto` statements, they should be eliminated [Amm92, Ero95].

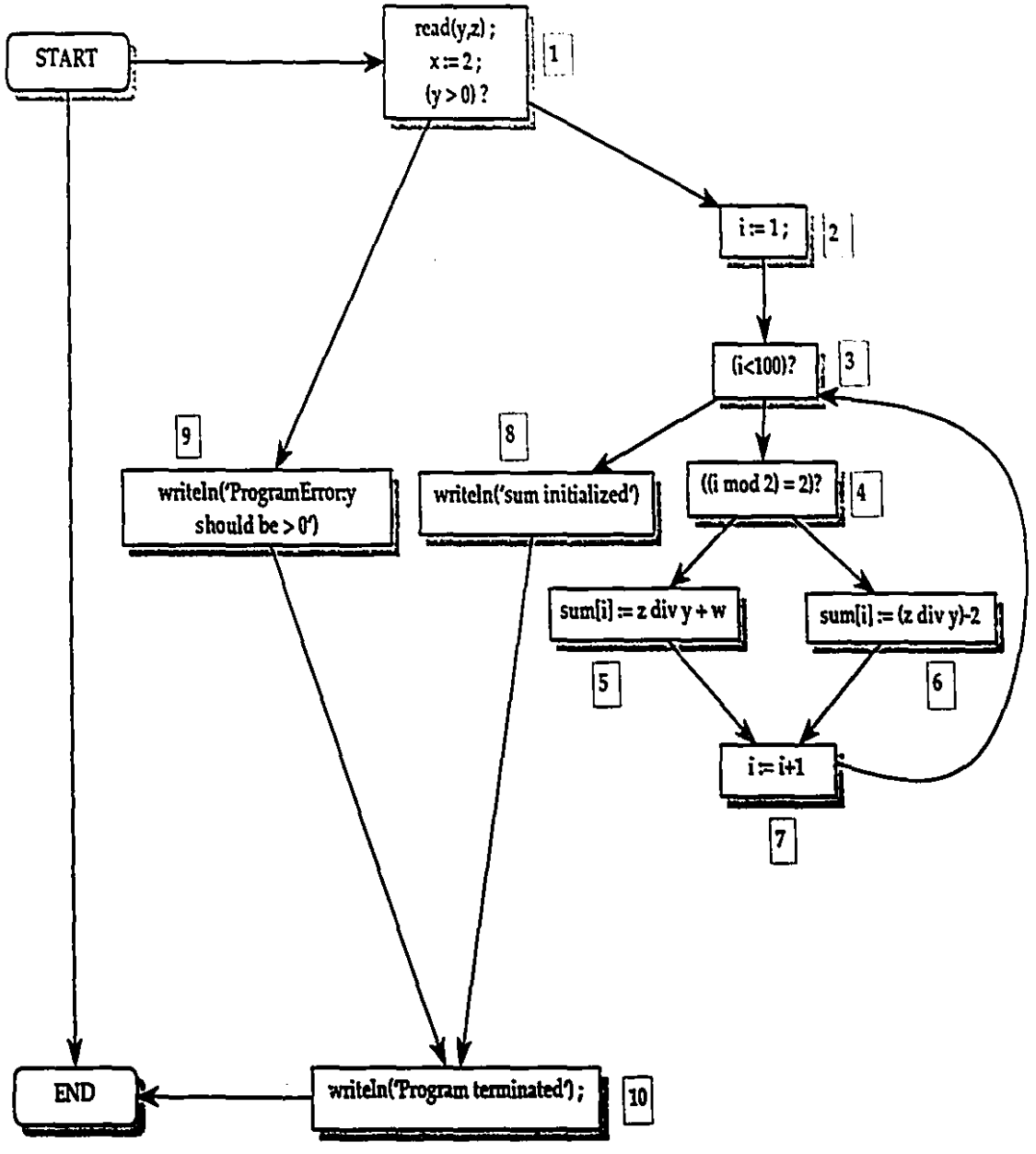


Figure 1.1: The control flow graph representation for the program `FOO()`.

the data flow information is computed from 'scratch' each time there is an incremental change in the program (either during program development or during optimization). In incremental methods, the data flow information is recomputed (ideally) only for those parts of the program that are affected because of an incremental change. Typically, information from other unaffected parts is used to recompute information in the affected parts. It is important to note that all incremental algorithms depend on having correct information or solutions at all points prior to incremental changes.[6]

In the demand-driven approach, flow information is computed on demand [DGS95]. That is, if a certain property needs to be verified or derived at a program point, a demand-driven 'engine' is invoked to do that job.

Finally, to improve the efficiency of certain classes of data flow problems, sparse evaluation techniques have been proposed in the literature [CFR+91, CCF91]. These techniques rely on constructing what are called Sparse Evaluation Graphs, on which data flow problems are solved. Intuitively, a sparse evaluation graph, for a particular data flow problem, is nothing but a subgraph of the control flow graph that contains only nodes that affect the information that is propagating through it [CCF91].

In this dissertation, we propose solutions to data flow problems that are based on iteration and elimination methods, and also based on exhaustive and incremental analysis.[7] We also give a new algorithm for constructing sparse evaluation graphs that is simple and efficient.

1.2 DJ Graphs: A New Representation for Solving Old Problems

At the heart of our work is a new representation called the **DJ Graph**. All other contributions revolve around this representation. What are DJ graphs? As motivation, consider the example program `FOO()`. Its CFG is shown in Figure 1.1. In the figure, we can see that nodes such as 3, 7, and 10, contain more than one

[6] It is important to note that control flow properties of a program can also be incrementally updated.

[7] We will not be concerned with syntax-directed approaches or demand-driven approaches in this dissertation.

predecessor. Such nodes are called *join* nodes, since they 'join' flow of control coming from different control paths. Now consider the incoming edges of one of these join nodes, for example, join node 3. The incoming edges to node 3 are 2→3 and 7→3. Consider the incoming edge 2 → 3, we can see that the source node 2 of the edge "strictly" dominates destination node 3, meaning that all paths from START of the CFG to node 3 always pass through node 2. Therefore the edge 2→3 is called as a *dominator edge*.[8] Now consider the incoming edge 7→3. Here the source node 7 does not strictly dominate node 3, meaning that there is an alternative path (START → 1 → 2 → 3) from START to node 3 that does not pass through 7. If this is the case we call the edge 7→3 a *join edge*. In general, an edge $x \rightarrow y$ in a CFG is a join edge if there exists an alternative path from START to y that does not "pass through" the edge $x \rightarrow y$. A DJ graph is a graph that represents both dominator edges and join edges in a single representation.[9] In this dissertation, we will demonstrate how this simple representation helps us in solving a number of program analysis problems very efficiently. The problems that we will address in this dissertation range from a simple problem of identifying loops to sophisticated data flow analysis based on elimination methods and incremental methods, including construction of sparse evaluation graphs.

1.3 Major Contributions of the Dissertation

In this section, we outline the major contributions of the dissertation, and discuss their importance in compiler optimizations and other program development tools. We will also discuss some of the related work along the way.

1.3.1 DJ Graphs and Their Properties

We introduce a new representation called the **DJ Graph** (Chapter 3). A DJ graph is nothing but the dominator tree of a program augmented with join edges.[10] An edge in a control flow graph is a join edge if the source node of the edge does not "strictly dominate" the destination node of the edge. Derived from a control

[8] More precisely we introduce a dominator edge between two nodes X and Y if X strictly dominates Y, and there is no other node Z not equal X strictly dominating Y (see Chapter 3).

[9] We will give a formal treatment of DJ graphs and their properties in Chapter 3.

[10] We have given the relevant background material and notation in Chapter 2.

flow graph, a DJ graph can be viewed as a refinement representing explicitly and precisely both the dominance relation between nodes and the potential program points where the flow information may be merged. Throughout this dissertation we will demonstrate how this simple representation enables us to design simpler and/or more efficient algorithms for performing sophisticated program analysis, including incremental analysis and sparse evaluations.

Contribution 1 *We introduce a new and a simple representation, called the* **DJ Graph**, *for solving program analysis problems. We also explore some of the properties of DJ graphs that will simplify many proofs in later chapters.*

1.3.2 Computing Dominance Frontiers and Related Sets

Dominance frontiers, control dependences, regions of control dependence etc., are control flow relations that are useful in many optimizing compilers [CFS90a]. In this dissertation, we will show how to efficiently compute some of these relations using DJ graphs (Chapter 4).

We will also show how to compute the **Top** node of a set of dominance frontier intervals in linear time (Chapter 4).[11] Our algorithm is very simple and does not require auxiliary data structures for keeping track of the intervals or annotating the dominator tree [CFS90b]; a DJ graph completely captures the interval information through join edges. The **Top** nodes are used in our elimination based data flow analysis (Chapter 10).

Contribution 2 *We propose new algorithms for computing dominance frontiers and related sets in linear time.*

1.3.3 A Fast Algorithm for Computing Multiple Node Immediate Dominators

Gupta introduced the notion of multiple-node immediate dominators for performing certain optimizations, like loop invariant removal and array bound checking, very aggressively [Gup92]. Intuitively, in a multiple-node dominance

[11] The dominance frontier interval is same as the control dependence interval on the reverse control flow graph [PB95, CFS90b].

relation a group of nodes dominate a node such that no subset of the group of nodes dominate the node [Gup92].

Gupta proposed a two-step process for computing the multiple-node dominance relation. In the first step, multiple-node *immediate* dominator nodes are computed, using which the complete dominance relation is computed. In multiple-node immediate dominators a group of predecessors of a node dominate the node. Gupta's algorithm for computing multiple-node immediate dominator has worst-case exponential time complexity of $O(|N|^p)$, where $|N|$ is the number of flowgraph nodes and p is the maximum number of predecessors of a node. In this dissertation, we propose a new algorithm for computing the same set in time $O(|E|^2)$, where $|E|$ is the number of edges in the DJ graph (Chapter 5).

Contribution 3 *We present a simple algorithm for computing multiple node immediate dominators in time $O(|E|^2)$, a considerable improvement compared to Gupta's algorithm, which is $O(|N|^p)$.*

1.3.4 Identifying Nested Reducible and Irreducible Loops

Loop identification is a necessary step in loop transformations for high-performance architectures [Wol89]. *Tarjan's intervals* are single-entry, strongly connected subgraphs, so they closely reflect the loop structure of a program [Tar74]. They have been used for loop identification. The basic idea behind Tarjan's method is to repeatedly *collapse* each natural loop into a single node inside-out until the whole graph reduces to one node. This idea will work if the flowgraph is reducible. In this dissertation, we generalize the collapsability to irreducible loops (Chapter 6). We propose a new algorithm that works on a DJ graph in a bottom-up fashion. Using our algorithm we can easily detect irreducible portions and collapse them immediately using Tarjan's strongly connected component algorithm, and then continue to do a bottom-up reduction. Our method can be considered as a generalization of Tarjan's reducibility algorithm. A novel aspect of our approach is that, in the presence of irreducible flowgraphs, our method can detect reducible loops within irreducible loops of the flowgraph.

Contribution 4 *We propose a new and a simple algorithm for identifying nested reducible and irreducible loops. Our algorithm can be considered as a generalization of Tarjan's reducibility algorithm.*

1.3.5 Computing Iterated Dominance Frontiers in Linear Time

Data flow analysis frameworks based on Static Single Assignment (SSA) form and Sparse Evaluation Graphs (SEGs) demand fast computation of program points where data flow information must be merged, the so-called ϕ-nodes [CFR+91, CF93, CCF91]. To determine where to place ϕ-nodes requires the knowledge of iterated dominance frontiers [CFR+91, CF93, CCF91]. Iterated dominance frontiers have other applications such as computing guards [Wei92] and incremental analysis (Chapter 8). We present a surprisingly simple algorithm for computing iterated dominance frontiers for a (sub-)set of nodes of arbitrary flowgraphs (reducible or irreducible) that runs in *linear* time (Chapter 11). To the best of our knowledge, this is the first linear time algorithm for computing iterated dominance frontiers (at the time it was first published [SG94, SG95b]). A novel aspect of our algorithm is that it can easily be adapted on other representations, like *APT* [PB95]. Previous algorithms for this problem were either not linear (for example, the algorithm of Cytron and Ferrante [CFR+91, CF93, CCF91]) or not general (for example, the algorithm of Johnson and Pingali is restricted to SSA forms [JP93]).

Contribution 5 *We present a simple linear time algorithm for computing iterated dominance frontiers for a set of nodes of an arbitrary flowgraph.*

1.3.6 Incremental Computation of Dominator Trees

Data flow analysis based on an incremental approach may require that the dominator tree be correctly maintained at all times (Chapter 10). Previous solutions to the problem of incrementally maintaining dominator trees were restricted to reducible flowgraphs [RR94, CR88]. In this dissertation, we present a new algorithm for incrementally maintaining the dominator tree of an arbitrary flowgraph, either reducible or irreducible (Chapter 8). For the case where an edge is inserted, our algorithm is also faster than previous approaches (in the worst case). For the deletion case, our algorithm is likely to run fast on the average cases. Unlike the previous methods we use properties of DJ graphs and iterated dominance frontiers for updating dominator trees.

Contribution 6 *We present a new incremental algorithm for updating the dominator tree of a flowgraph.*

1.3.7 Incremental Computation of Dominance Frontiers

We present a simple algorithm for incrementally updating the dominance frontier relation of a flowgraph (Chapter 9). We are not aware of any published work for solving this problem. Our algorithm relies on having the incremental computation of dominator trees. Since dominance frontiers are same as control dependences on the reverse flowgraph, our algorithm can also be used for incrementally updating the control dependence relation. Finally, incremental computation of dominance frontiers is useful in incremental data flow analysis (Chapter 10).

Contribution 7 *We present a simple incremental algorithm for updating the dominance frontier relation of a flowgraph.*

1.3.8 A New Framework for Elimination-Based Data Flow Analysis: Exhaustive Analysis

Despite much ground research work that has been done in elimination methods, many researchers and practitioners prefer to use iterative methods for the following two reasons: (1) it is simple and easy to implement, and (2) can handle arbitrary flowgraphs, including irreducible flowgraphs. Elimination methods, on the other hand, are more efficient than iterative methods, but are more complex to implement. Also, some elimination methods cannot handle irreducible flowgraphs, and even if they do, they are not very efficient.

In this dissertation we propose a new framework for data flow analysis based on elimination methods. We will demonstrate that our approach is simple, easy to implement, practically efficient, able to handle irreducible flowgraphs, and amenable to incremental analysis. At the heart of our approach is the DJ graph representation. Within our framework we propose two algorithms for exhaustive data flow analysis, and one algorithm for incremental data flow analysis (Section 1.3.9).

We propose two variations of our approach for exhaustive data flow analysis: (1) the **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) the **delayed elimination method**, whose worst case time complexity is

$O(|E| \times log(|N|))$ (Chapter 10).[12] Rather than reducing a DJ graph into 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.

Although the time complexity of both eager and delayed eliminations are worse than linear, we will demonstrate that the two methods are expected to behaves linearly in practice. Another novel aspect of our approach (both eager and delayed eliminations) is that it can easily identify and handle *irreducibility* gracefully in the bottom-up reduction process. 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).

Contribution 8 *We propose a new framework for data flow analysis based on elimination methods. We propose two variations of our approach for exhaustive data flow analysis: (1) the eager elimination method, and (2) the delayed elimination method. A novel aspect of our approach is that it can handle irreducible loops very efficiently. Although the time complexity of eager and delayed eliminations are worse than linear, we will demonstrate that they are expected to behave linearly in practice.*

1.3.9 A New Framework for Elimination-Based Data Flow Analysis: Incremental Analysis

In this dissertation we present a new method for incremental data flow analysis based on elimination methods (Chapter 11). Our method is based on incrementalizing our eager elimination method. Unlike previous approaches [Bur90, CR88,

[12] 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, Mar89]. Please see Appendix A for relevant background on data flow framework.

RP88], our method can handle arbitrary non-structural and structural program changes, including irreducibility. Also, our method is novel in the sense that we use properties of dominance frontiers and iterated dominance frontiers for updating the data flow information.

Contribution 9 *We propose a new incremental data flow analysis based on elimination methods that can handle arbitrary incremental program changes, including irreducibility.*

1.3.10 Experiments and Empirical Results

In this dissertation, we have proposed a number of algorithms for solving problems encountered in program analysis. To demonstrate the effectiveness of our algorithms we implemented many of them using flowgraphs generated from the Parafrase2 compiler [PGH+91], and experimented on real FORTRAN procedures taken from SPEC92, LAPACK, GATOR, and RiCEPs. In Chapter 3 we give our experimental framework, and within each relevant chapters we provide the experimental results. This is the last contribution of the dissertation.

Contribution 10 *To demonstrate the effectiveness of our algorithms we implemented many of them using flowgraphs obtained from real FORTRAN programs. We used the Parafrase2 compiler for generating flowgraphs. We provide empirical results of our experiments, and give their analysis.*

1.4 Organization of the Dissertation

This dissertation is organized into a number of chapters. Figure 1.2 shows a "Road Map" of the major chapters in the dissertation, the edges in the figure represent dependences among the chapters. In Chapter 2, we give the relevant background material and notation that are necessary for understanding of the dissertation. In Chapter 3, we introduce DJ graphs and discuss some of their properties. Here we also give our experimental framework, and within each relevant chapter we provide experimental results. In Chapter 4, we give algorithms for computing dominance frontiers and related sets. In Chapter 5, we present a new algorithm for computing the multiple-node immediate dominance relation. In Chapter 6, we present a new algorithm for identifying reducible and irreducible loops. In

Chapter 7, we present a simple linear time algorithm for computing iterated dominance frontiers. In Chapter 8, we present a new incremental algorithm for updating the dominator tree of a flowgraph subjected to incremental changes. In Chapter 9, we present a new incremental algorithm for updating the dominance frontier set of a flowgraph subjected to incremental changes. In Chapter 10, we propose a new approach for elimination based data flow analysis. This chapter requires some knowledge on data flow frameworks. We refer readers to Appendix A for this background. In Chapter 11, we present a new incremental data flow analysis based on elimination methods. Finally, we will conclude in Chapter 12, projecting possible future work.

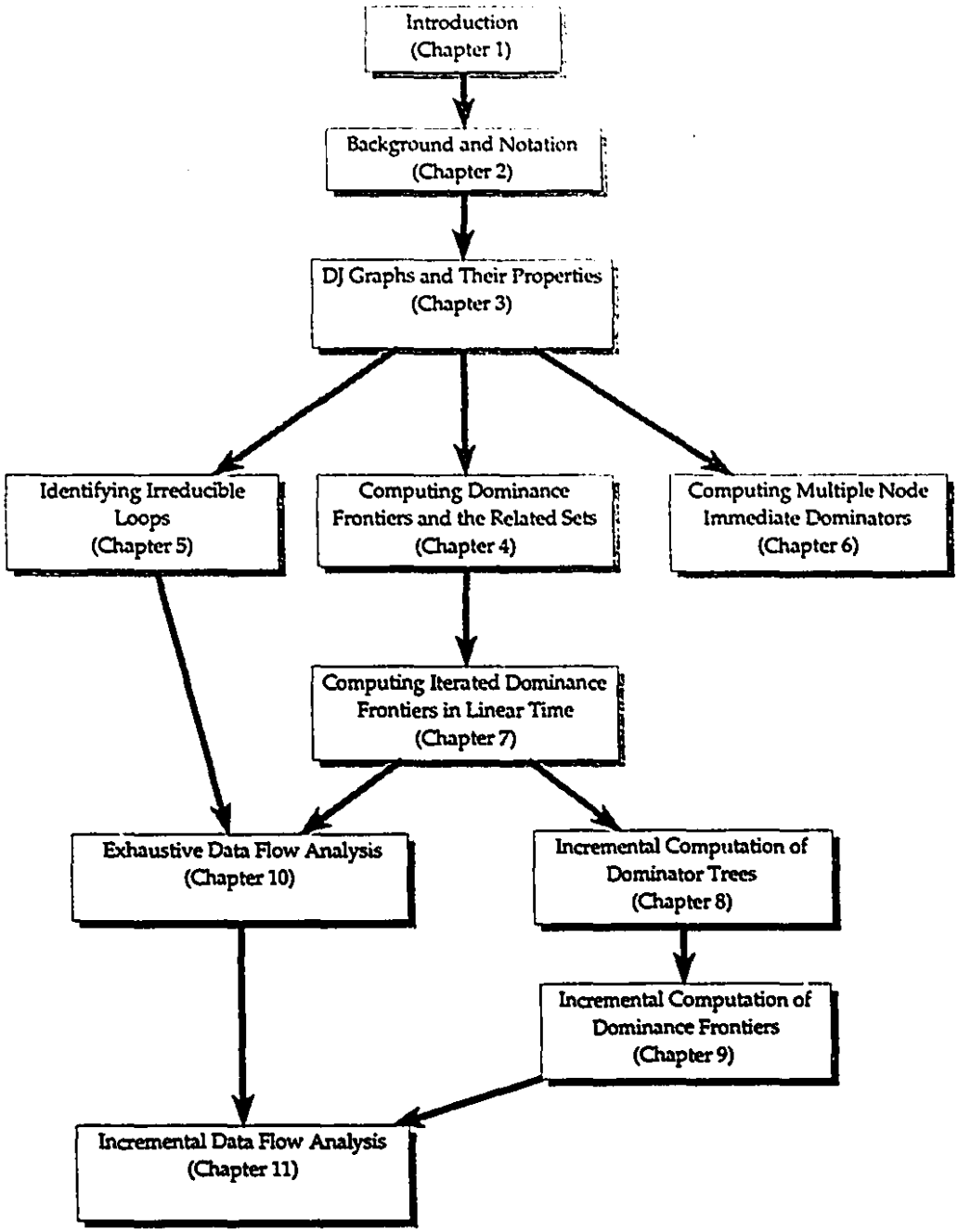


Figure 1.2: A 'Road Map' of the major chapters in this dissertation.

Chapter 2

Background and Notation

The past was great no doubt, but I sincerely believe that the future will be more glorious

—Swami Vivekananda

In this chapter we introduce the relevant background material and notation used in this dissertation. (More notation and terminology will be introduced in later chapters where they are used.)

Consider the program `FOO()` given in Chapter 1. The corresponding Control Flow Graph (CFG) is shown in Figure 2.1(a). Recall that a CFG is a basic structure on which the data flow information is propagated. Depending on the data flow problem, the information can be propagated either in the "forward" direction (from a node to all its successor nodes), or the "backward" direction (from a node to all its predecessor nodes). An example of a forward propagation problem is the reaching definitions problem, and an example of a backward propagation problem is live variable analysis.[1] For solving forward problems we use "forward" control flow graphs, and for solving backward problems we use "reverse" control flow graphs. In a reverse control flowgraph we reverse the orientation of the control flow edges.

In this dissertation we will use the term **flowgraph** to uniformly represent either the forward control flow graph or the reverse control flow graph, depending on the direction of the problem. We formally define a flowgraph as follows:

[1]Section 10.2 in Chapter 10 and Appendix A gives a brief to introduction data flow analysis.

Definition 2.1 *A* **flowgraph** *is a rooted directed graph* $G = (N, E, Root)$*, where* N *is the set of nodes,* E *is the set of edges, and* $Root \in N$ *is a distinguished root node with no incoming edges.*

For forward control flow graphs the *Root* node is the START node of the control flow graph, and for reverse control flow graphs the *Root* node is the END node of the control flow graph. Throughout this dissertation we will mostly deal with forward control flow graphs, and therefore we will use the notation START to denote the root of a flowgraph (unless otherwise explicitly specified that END is *Root*).

If $x \rightarrow y \in E$, then x is called the *source* node and y is called the *destination* node of the edge; and sometimes we will say that y is a *successor* of x, and x is a *predecessor* of y. The set of all successors of a node $x \in N$ is denoted by $Succ(x)$, while the set of all predecessors of x is denoted by $Pred(x)$.

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 the notation $P : x_0 \xrightarrow{*} x_n$ to represent a path of length zero or more, and $P : x_0 \xrightarrow{+} x_n$ to represent a path of length one or more. Given the flowgraph of a program, we define the reachable subgraph REACH(*Root*) to be a subgraph of G such that all nodes in REACH(*Root*) are reachable from *Root*.

Clarification 2.1

> *A flowgraph need not be connected, that is, some nodes may not be reachable from the root node. This is possible during program optimization, such as dead-code elimination, where a part of the flowgraph (the dead-code) is eliminated; or during program development, where a part of the flowgraph can be temporarily disconnected. In this dissertation we will assume that all program properties, such as the dominance relation, reaching definitions, etc., are defined only for the reachable subgraph* REACH(*Root*)*. Therefore, whenever we use the phrase "a flowgraph and its dominator tree," we are referring to "the reachable subgraph* REACH(*Root*) *and its dominator tree"; or when a node* x *dominates another node* y*, we implicitly assume that both* x *and* y *are in* REACH(*Root*)*. This convention applies to other properties as well; if a property is defined with respect to a node* x *or an edge* $x \rightarrow y$*, we will implicitly assume that* x *and* y *are in* REACH(*Root*)*.*

Let S be a set, we will use the notation $|S|$ to denote the number of elements in the set.

Next we introduce the dominance relation.

Definition 2.2 *In a flowgraph, a node x* **dominates** *another node y iff all paths from* START *to y always pass through x.*

We write x dom y to indicate that x dominates y, and write x !dom y if x does not dominate y. If x dom y and $x \neq y$, then x **strictly dominates** y. We write x stdom y to indicate that x strictly dominates y, and write x !stdom y if x does not strictly dominate y. A node x is said to *immediately dominate* another node y, denoted as $x = idom(y)$, if x stdom y and there is no other node $z \neq x$ and $z \neq y$ such that x stdom z stdom y. The dominance relation is reflexive and transitive, and can be represented by a tree, called the **dominator tree.** $x \rightarrow y$ is an edge in the dominator tree of a flowgraph iff $x = idom(y)$. Given a node x in the dominator tree, we define $SubTree(x)$ to be the dominator sub-tree rooted at x. Note that the nodes in $SubTree(x)$ are simply the set of all nodes dominated by x. For each node in the dominator tree we associate a *level number* that is the depth of the node from the root of the tree. We write $x.level$ to indicate the level number of a node x.

Example 2.1

> Figure 2.1(b) shows the dominator tree for the flowgraph shown in Figure 2.1(a). In the figure, we can see that node 3 dominates each node in $\{3,4,5,6,7,8\}$, and node 3 strictly dominates each node in $\{4,5,6,7,8\}$. Also, in the same figure, $idom(4) = 3$, and the nodes in $Subtree(4) = \{4,5,6,7\}$. Finally, the levels of the nodes are: START.$level = 0$, 1.$level = 1$, 2.$level = 2$, etc.

A number of algorithms have been proposed for computing the dominance relation [ASU86]. Lengauer and Tarjan proposed an algorithm that is almost linear, $O(|E| \times \alpha(|N|, |E|))$, where $\alpha()$ is the slowly-growing inverse Ackermann function [LT79]. More recently, Harel has given a *linear* algorithm for computing the dominator tree of a flowgraph [Har85]. We are not aware of any practical implementation of Harel's algorithm. In our research work we have implemented Lengauer and Tarjan's algorithm to construct DJ graphs (See Chapter 3).

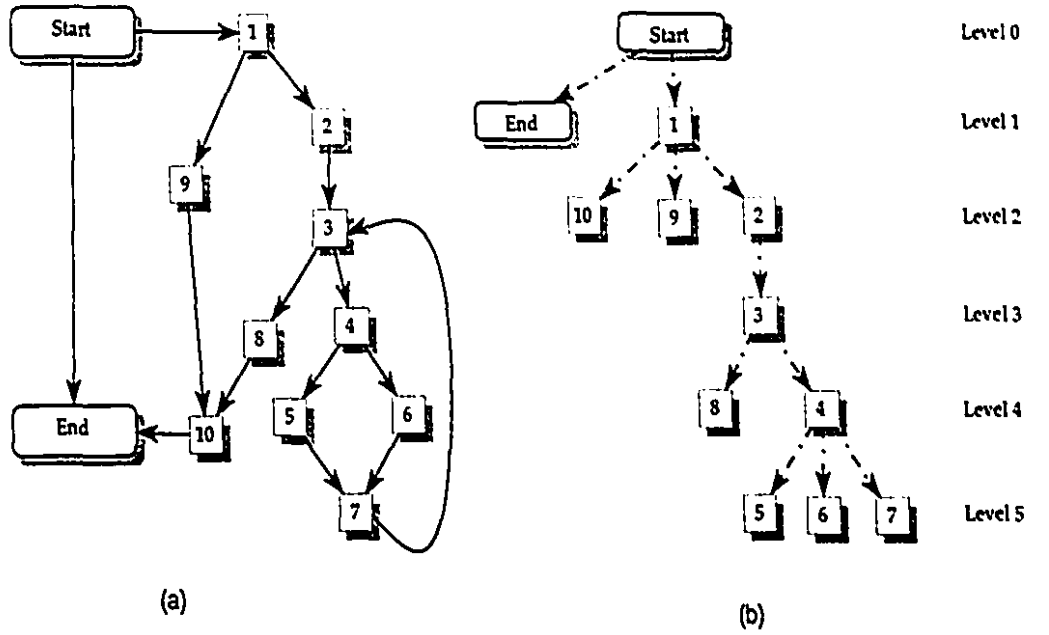


Figure 2.1: The flowgraph for program `FOO()` and its dominator tree.

Another concept that is important in this dissertation is the dominance frontier relation [CFR+91].

Definition 2.3 *The* **dominance frontier** $DF(x)$ *of a node* x *is the set of all* z *such that* x *dominates a predecessor of* z*, without strictly dominating* z*.*

Intuitively, one can visualize the dominance frontier of a node x as follows: to compute the dominance frontier of x, shine a *light* at START, and put a *shade* at all the output edges of node x, so that the light does not pass through them.[2] This partitions the flowgraph into dark and light regions. Consider all the dark edges that are incident on light nodes. The destination nodes of these edges form the dominance frontier of x.[3]

Sometimes it is convenient to think of dominance frontiers as a set of edges rather than as a set of nodes. If this is the case we will use the notation $DF_e(x)$ to denote this set, and define $DF_e(x)$ as follows:

[2]Think of nodes to be like crystal balls and edges as "directed" optic fibers.
[3]The above intuition is due to Barry Rosen [Ros94].

Definition 2.4 *The* **dominance frontier** $DF_e(x)$ *of a node* x *is the set of edges* $y \rightarrow z$ *such that* x *dominates* y*, without strictly dominating* z*.*

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

$$DF(S) = \bigcup_{x \in S} DF(x) \tag{2.1}$$

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

$$IDF_1(S) = DF(S), \tag{2.2}$$

$$IDF_{i+1}(S) = DF(S \cup IDF_i(S)) \tag{2.3}$$

Example 2.2

Consider the flowgraph shown in Figure 2.1(a). The dominance frontier for node 3 is $DF(3) = \{3, 10\}$. To see this, consider node 3 in $DF(3)$. We can see that node 3 dominates a predecessor of 3 (which is 7) and a predecessor of 10 (which is 8). But in both cases 3 does not strictly dominate nodes 3 and 10. Therefore, nodes 3 and 10 are in $DF(3)$.

Now let us compute $IDF(3)$. Using Equation (2.3) the iterated dominance frontiers is computed as follows:
$IDF_1(3) = \{3, 10\}$
$IDF_2(3) = DF(3 \cup \{3,10\}) = \{3,10,END\}$
$IDF_3(3) = DF(3 \cup \{3,10,END\}) = \{3,10,END\}$. We find no more changes in the iteration, and so $IDF(3) = \{3,10,END\}$.

Chapter 3

DJ Graphs and Their Properties

If I can't picture it, I can't understand it.

—Albert Einstein

As far as the laws of mathematics refer to reality, they are not certain, and as far as they are certain, they do not refer to reality.

—Albert Einstein

At the heart of this dissertation is a new representation called the **DJ Graph**. In this chapter we formally introduce DJ graphs (Section 3.1) and discuss some of the properties of DJ graphs that are relevant for our work (Section 3.2). To demonstrate the effectiveness of the algorithms based on DJ graphs, we have implemented them using flowgraphs generated from the Parafrase2 compiler [PGH+91]. In Section 3.3 we present our experimental framework based on DJ graphs. Here we also present some experimental results on the characteristics of DJ graphs for real programs. Finally, in Section 3.4, we discuss the related work.

3.1 The DJ Graph

In Chapter 1 we informally introduced DJ graphs. Recall that central to DJ graphs is the notion of *join* edges (J edges) and *dominator* edges (D edges). We defined an edge $x \rightarrow y$ in a flowgraph to be a join edge if there is an alternative path from START to y that does not pass through the edge $x \rightarrow y$. More formally, we define J edges as follows:

Definition 3.1 (Join Edge) *An edge $x \rightarrow y$ in a flowgraph is named a* **join edge** (J edge) *if $x \neq idom(y)$. Furthermore, we will call y as a* **join node.**

Now if $x = idom(y)$ then there is an edge from $x \rightarrow y$ in the dominator tree, called the **dominator edge** (or D edge). It is trivial to see that a node is a join node if it has more than one incoming edges; and that join edges are a subset of the incoming edges of a join node. Given the notion of D and J edges we define DJ graphs as follows:

Definition 3.2 (DJ Graph) *A* **DJ graph** $G_d = (N_d, E_d, Root_d)$ *is a rooted directed graph that consists of the same set of nodes as in its flowgraph, and two types of edges called* **D edges** *and* **J edges.** *D edges are dominator tree edges, and J edges are the join edges in the flowgraph.*

To construct the DJ graph of a flowgraph, we first construct the dominator tree of the given flowgraph, and then we insert the J edges into the dominator tree. The complete algorithm is give below.

Algorithm 3.1 *The following algorithm constructs the DJ graph of a flowgraph.*

```
MainDJG()
{
1:    Compute the immediate dominance relation using Lengauer and Tarjan's
      algorithm.
2:    Construct the dominator tree using the immediate dominance relation.
3:    foreach edge x → y in the flowgraph do
4:      if x → y is not an edge in the dominator tree then
5:        Insert an edge from x to y in the dominator tree and mark it as J edge
6:      else
7:        Mark the corresponding edge x → y in the dominator tree as D edge.
8:      endif
}
```

If we ignore the time complexity of Lengauer and Tarjan's algorithm (step [1]), we can see that the time complexity of the above algorithm is $O(|E_f| + |N_f|)$.[1]

[1] If we use Harel's algorithm for computing immediate dominance relation, the time complexity of step [1] would be $O(|E_f| + |N_f|)$.

Given a DJ graph we distinguish between two types of J edges: Back J (BJ) edges and Cross J (CJ) edges. A J edge $x \rightarrow y$ is a BJ edge if y dom x, otherwise it is a CJ edge.

Example 3.1

To see how a join edge is inserted in the dominator tree, consider the join node 3 of the example flowgraph shown in Figure 2.1. It has two incoming edges 7→3 and 2→3. Of the two edges only 7→3 is a J edge, and 2→3 is not a J edge since 2→3 is an edge in the dominator tree. We therefore insert the J edge 7→3 in the dominator tree. We can similarly insert other J edges. Figure 3.1 shows the complete DJ graph of the flowgraph. Throughout this dissertation we will use solid dark edges for representing J edges, and dash-and-dotted edges for representing D edges. Finally, an example of a BJ edge is 7→3 and an example of CJ edge is 6→7.

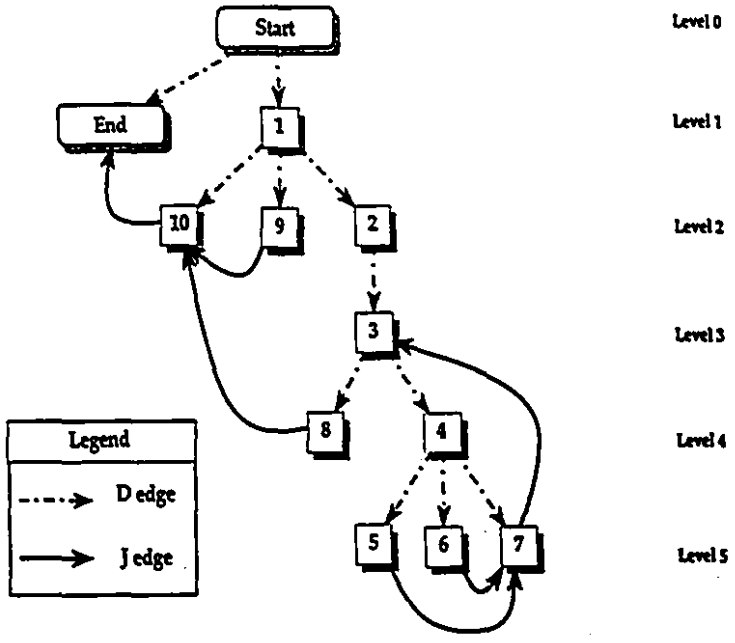


Figure 3.1: The DJ graph for the flowgraph shown in Figure 2.1.

Clarification 3.1

Since a flowgraph and its DJ graph contain the same set of nodes, whenever we use the term node x, *the node* x *may belong to either the flowgraph or the DJ graph, unless explicitly specified by using subscript notation with* f *for flowgraph and* d *for DJ graph. We will use the notation* $Succ_f(x)$ $(Pred_f(x))$ *to be the set of successor (predecessor) nodes on flowgraphs, while* $Succ_d(x)$ $(Pred_d(x))$ *to be the set of successor (predecessor) nodes on DJ graphs.*

Also, a DJ graph is made up of dominator tree edges and join edges. Therefore, whenever we use the term dominator tree, we also mean a DJ graph without J edges, and vice versa.

Finally, DJ graphs are defined (constructed) only for reachable subgraphs of flowgraphs.

3.2 Properties of DJ Graphs

In this section we discuss some of the properties of DJ graphs that are relevant to our discussion. We will subsequently use these properties for proving the correctness and analyzing the complexity of some of the major results in this dissertation. The first property gives us an upper bound on the size of a DJ graph with respect to the size of its flowgraph.

Theorem 3.1 *Let* $G_f = (N_f, E_f, \text{START}_f)$ *be a flowgraph, and let* $G_d = (N_d, E_d, \text{START}_d)$ *be the corresponding DJ graph. Then,* $|N_d| = |N_f|$ *and* $|E_d| < (|N_f| + |E_f|)$.

Proof:

The proof is based on the following observation: a DJ graph has the same set of nodes as its flowgraph, hence $|N_f| = |N_d|$.

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

From the above theorem we can easily derive the following corollary.

Corollary 3.1 *The size of a DJ graph is linear with respect to the size of its flowgraph.*

This is interesting because from now on we will argue the time complexity of our algorithms using either DJ graphs or flowgraphs.

Next we will discuss structural properties of DJ graphs. For every edge $x_f \rightarrow y_f$ in a flowgraph, there is a corresponding edge $x_d \rightarrow y_d$ in the DJ graph. But the reverse may not be true; that is, the DJ graph can have more edges than its flowgraph does. For example, consider the flowgraph given in Chapter 2 (Figure 2.1), whose DJ graph is shown in Figure 3.1. We can see that there is no edge from node 4 to 7 in the flowgraph, but there is an edge from 4 to 7 in the corresponding DJ graph.

Next we can see that for every path $P_f : x_f \stackrel{*}{\rightarrow} y_f$ in a flowgraph there is a corresponding path $P_d : x_d \stackrel{*}{\rightarrow} y_d$ in the DJ graph, and vice versa. This immediately follows from the construction of the DJ graph. Note that the nodes in the two paths may not be exactly the same. In other words, given any two nodes x and y, y is reachable from x in a flowgraph if and only if y is reachable from x in the corresponding DJ graph.[2]

Also recall that there is exactly one path from START to some node in the dominator tree of a flowgraph. Due to the presence of J edges, there may be more than one path from START to any other node in a DJ graph. How are J edges and D edges related? D edges and J edges are related in many ways. One such relation is given in the following lemma.

Lemma 3.1 *Let $x \rightarrow y$ be a J edge, then $idom(y)$* **stdom** *x.*

Proof:

Let $z = idom(y)$, and assume that z !stdom x. We can immediately see that there is a path from START to y that does not go through z, contradicting our initial assumption that z immediately dominates y.

■

Another relation between D and J edges is in terms of levels of the nodes in the dominator tree of the DJ graph. The following lemma establishes this relation.

[2]Another view of the DJ graph may give a better intuition behind these observations: A DJ graph can also be constructed by adding every missing *immediate dominance edge* $x \rightarrow y$ into its flowgraph if the edge is not already present in the flowgraph.

Lemma 3.2 *Let $x \rightarrow y$ be a J edge in a DJ graph, then $x.level \geq y.level$.*

Proof:

Suppose $x.level < y.level$. There are two cases:

1. There is a path from x to y in the dominator tree. This is impossible by definition (because x will dominate y).
2. There is no path from x to y in the dominator tree. Let z be a node such that $z.level = x.level$, and z dom y (actually, z strictly dominates y). Since $x \rightarrow y$ is in the DJ graph, there is a path from START to x to y in the CFG that does not pass through z (a contradiction, because we have assumed z dom y).

∎

Next let us see how D edges and J edges are related to the dominance frontier relation. Using the definition of dominance frontiers (Definition 2.3) we can easily see that, if $x \rightarrow y$ is a J edge, then $y \in DF(x)$. But in Lemma 3.2 we showed that if $x \rightarrow y$ is a J edge then $x.level \geq y.level$. We can generalize the above lemma as follows:

Theorem 3.2 *Let x be a node in a DJ graph, and let $DF(x)$ be the dominance frontier of x. Then, $x.level \geq y.level$ for each $y \in DF(x)$, and $x.level \geq y.level$ for each $y \in IDF(x)$.*

Proof:

Let $y \in DF(x)$ and let $u = idom(y)$. We will first show that u will strictly dominate x. Then using this result we will prove the validity of the theorem. Assume that u does not strictly dominate x. Then there is a path from START to x that does not pass through u. Since $y \in DF(x)$, there must be a predecessor node z strictly dominated by x. From this we can immediately see that there is a path from START$\ldots \rightarrow x\ldots \rightarrow \ldots z \rightarrow y$ in the DJ graph that does not pass through u, contradicting the assumption that $u = idom(y)$. Therefore u stdom x. From this we can easily see that

$$u.level < x.level$$

Again since $u = idom(y)$, $u.level = y.level - 1$. Substituting this in the above inequality, we get

$$y.level - 1 < x.level$$

or

$$y.level \leq x.level$$

Hence the result.

The property $x.level \geq y.level$ for each $y \in IDF(x)$, inductively follows from the definition of iterated dominance frontier and the property $x.level \geq y.level$ for each $y \in DF(x)$. ∎

Intuitively, Theorem 3.2 implies that given a node x, to determine its dominance frontier (or iterated dominance frontier) we only need to look at those nodes at the same level as x or above it, in the corresponding DJ graph. Nodes whose level number is greater than the level number of x in the dominator tree will never be in the dominance frontier of x. We will use this property in Chapter 4 and propose a new algorithm for computing the dominance frontier of a node. We will again use this result in Chapter 7, where we present a linear time algorithm for computing iterated dominance frontiers.

3.3 Experimental Framework and Empirical Evaluation

We implemented many of the algorithms presented in this dissertation using flowgraphs generated from the Parafrase2 compiler [PGH+91].[3] We implemented the Lengauer-Tarjan (LT) almost linear time algorithm for finding immediate dominators.[4] Using the immediate dominator information we next constructed DJ graphs. Using DJ graphs as the basis we implemented our algorithms. We chose a set of 40 FORTRAN procedures from SPEC92, LAPACK, GATOR, and RiCEP

[3]Parafrase2 is a research tool developed at the Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign.

[4]Due to the complex nature of Harel's linear time algorithm, we did not implement that algorithm.

programs.[5] We looked for programs that has larger control flow structure, some of them have complex flow of control and some them have simpler flow of control. Table 3.1 lists all the procedures used for our experiments in alphabetical order. Throughout this dissertation we will use these 40 procedures to quantitatively study characteristics of our algorithms. Within each relevant chapter we provide our experimental results and their analysis concerning that chapter.

In the remaining portion of this section we will study some interesting characteristics of flowgraphs, dominator trees, and DJ graphs for our test procedures. We will first summarize the main observations of our experiments for the procedures we tested.

- The number of flowgraph edges is approximately 42% more than the number of flowgraph nodes. This suggests that flowgraphs for practical programs are sparse [ASU86].

- The dominator tree structures are generally flat. The average depth (averaged over the depth of all the nodes) range from 9.0 to 90.0, whereas the maximum depths of dominator trees range from 14 to 179. This suggests that dominator trees are generally flat, when compared to the number of nodes in the corresponding flowgraphs.

- We measured the difference $|E_d| - |E_f|$, which gives the number of "extra" D edges that are in the dominator tree but not in the flowgraph. On average we found that the number of extra D edges to be 17, which is 5.4% more than the number of flowgraph edges. This suggests that the size of a DJ graph is very close to the size of its flowgraph.

- The average depth of J edges (measured as the difference between the depth of source and destination nodes of J edges, and averaged overall all J edges), range from 0.5 to 6.3, whereas the maximum depth of J edges range from 1 to 39. These two results suggest that J edges are generally flat.

- The average time for computing immediate dominance relation is 8.9 milliseconds. Now given the immediate dominance relation, the average time

[5]SPEC (Standard Performance Evaluation Corporation) is a standard suite of benchmark programs from SPEC Associates. LAPACK is a linear algebra package from Argonne National Lab. GATOR is a Gas, Aerosol Transport, and Radiation Model from University of California at Los Angele. Finally, RiCEP is a benchmark suite from Rice University.

for constructing DJ graphs is 1.2 milliseconds. This suggests that constructing DJ graphs is very fast in practice.

We will further elaborate on these results in the rest of this section. Table 3.1 and Table 3.2 gives a summary of our results. The notations used in Table 3.1 and Table 3.2 are given below:

Notation used in Table 3.1 and Table 3.2

Name	Procedure names.
Lines	Number of lines parsed (excluding comments).
$\|N\|$	Number of flowgraph nodes (where nodes are basic blocks).
$\|E_f\|$	Number of flowgraph edges.
$\|E_d\|$	Number of DJ graph edges.
L_{max}	Maximum depth of dominator trees
L_{ave}	Average depth of dominator trees (averaged over the depth of all nodes in the tree)
J_{max}	Maximum depth of J edges
J_{ave}	Average depth of J edges (averaged over the depth of all J edges)
$Frontend(s)$	Execution time for front-end parsing (in seconds)
$FG(ms)$	Execution time for constructing flowgraphs (in milliseconds)
$LT(ms)$	Execution time for computing immediate dominators using the LT algorithm (in milliseconds)
$DJG(ms)$	Execution time for constructing DJ graphs given the immediate dominator information (in milliseconds)

The second column of Table 3.1 gives the number of lines (*Lines*) of code, as computed by the Parafrase compiler, within each procedure. This does not include comments. The total number of lines of code for the 40 procedures is 20,599, and the average number of lines of code is 515. The third and the fourth columns of Table 3.1 give the number of flowgraph nodes $|N|$ and the number of flowgraph edges $|E_f|$, respectively. For our test procedures, the average number of flowgraph nodes is 219, and the average number of flowgraph edges is 312. From the table we can also see that the number of flowgraph edges is not much

larger than the number of flowgraph nodes, suggesting that the number of edges are linear with respect to the number of nodes.

We next measured depth (or depths) of dominator trees. From the table we can see that maximum depth L_{max} of dominator trees ranges from 14 to 179, and the average depth of dominator trees L_{ave} range from 9.0 to 90.0. We notice that for many procedures, the average and the maximum depth is small compared to number of flowgraph nodes, suggesting that dominator trees are generally flat for many programs. An exception to this is the dominator tree of the procedure `iniset`. For this procedure the average and the maximum depth are 90.0 and 179, respectively. A careful examination of this procedure reveals that it consists of 154 simple DO loops for initializing array varibles.

Next we measured the number of edges in DJ graphs (D edges + J edges). This is shown in the column $|E_d|$. The difference $|E_d| - |E_f|$ gives the number of "extra" D edges that are not in the corresponding flowgraph. We can see that the number of extra D edges range from 0 to 39, with 17 being the average. This suggests that the size of a DJ graph is almost the same as the size of its flowgraph.

Next we measured "depth" of J edges. We define the *depth* of a J edge $x \rightarrow y$ to be $x.level - y.level$. The columns J_{max} and J_{ave} give the maximum and the average depth of J edges, respectively. For our test procedures, maximum depth range from 1 to 39, and average depth of J edges ranges from 0.46 to 6.28. From these two results we can conclude that J edges are also quite flat for practical programs.

Finally, we measured execution times for front-end parsing (*Frontend*), for generating flowgraphs (*FG*), for computing immediate dominators using the LT algorithm (*LT*), and for constructing DJ graphs given the immediate dominator information (*DJG*). These measurements are given in Table 3.2. From the table we can see that the time for constructing DJ graphs is much smaller than the time for computing the immediate dominance relation using the LT algorithm.

3.4 Discussion and Related Work

In this chapter we introduced a new program representation called the DJ graph. Derived from a flowgraph, a DJ graph can be viewed as a refinement representing explicitly and precisely both the dominance relation between nodes (via D edges) and the potential program points where different control paths merge (via J edges).

Name	*Lines*	$\|N\|$	$\|E_f\|$	$\|E_d\|$	L_{max}	L_{ave}	J_{max}	J_{ave}
aerset	768	329	460	467	39	21.5	15	1.7
aqset	512	189	258	263	38	22.3	9	1.7
bjt	394	135	187	213	21	9.0	14	1.7
card	201	150	216	235	23	10.8	14	2.7
chemset	633	229	320	330	28	21.0	28	1.9
chgeqz	342	174	248	268	36	16.8	13	2.1
clatrs	408	214	308	337	21	13.3	5	1.2
coef	243	95	137	154	21	11.9	7	1.4
comlr	135	69	91	97	19	9.6	10	1.7
dbdsqr	542	228	327	343	31	18.7	15	2.4
dcdcmp	211	137	187	205	22	9.0	13	2.5
dcop	441	186	261	298	23	13.0	12	1.6
dctran	508	326	458	493	36	15.9	26	3.4
deseco	473	175	236	259	28	14.0	9	1.6
dgegv	290	160	232	246	34	19.1	18	2.3
dgesvd	1142	321	470	499	19	11.5	10	2.3
dhgeqz	631	285	408	433	39	22.3	25	2.2
disto	382	133	191	211	17	9.7	5	1.3
dlatbs	317	167	238	259	18	10.4	6	1.4
dtgevc	555	321	459	485	36	20.4	13	1.7
dtrevc	467	248	353	373	25	15.6	7	1.5
elprnt	296	162	227	245	24	10.0	9	1.2
equilset	782	327	451	467	58	32.3	9	1.5
errchk	462	346	482	515	45	24.6	37	2.1
iniset	456	333	486	486	179	90.0	1	1.0
init	265	122	175	176	23	13.9	10	1.3
initgas	511	189	263	267	34	17.9	21	1.6
jsparse	724	281	403	408	34	17.6	8	1.4
modchk	444	306	419	455	34	16.9	28	1.6
moseq2	348	161	217	246	22	10.6	12	2.1
mosfet	562	214	295	333	22	10.8	12	2.5
noise	310	115	160	184	14	8.2	4	1.1
out	1178	403	590	597	45	16.9	18	1.7
reader	697	182	235	242	67	30.4	5	0.5
readin	677	406	611	637	44	21.4	36	6.3
setupgeo	673	188	275	278	45	20.7	9	1.4
setuprad	698	195	286	290	34	21.3	18	1.3
smvgear	576	212	310	316	46	27.0	39	3.4
solveq	556	196	289	298	23	10.6	7	1.6
twldrv	759	168	243	258	53	29.7	18	2.1
Average	515	219	312	329	24	18.7	14	1.9

Table 3.1: Structural characteristics of DJ graphs and flowgraphs for the test procedures.

Name	$Frontend(s)$	$FG(ms)$	$LT(ms)$	$DJG(ms)$
aerset	1.8	34.2	12.2	1.6
aqset	1.4	20.1	7.7	0.9
bjt	0.7	17.0	5.7	1.3
card	0.3	14.0	6.2	0.8
chemset	1.6	25.9	9.1	1.1
chgeqz	0.5	18.3	7.1	0.9
clatrs	0.5	23.2	8.2	1.4
coef	0.4	10.8	4.6	0.5
comlr	0.2	6.9	4.1	0.3
dbdsqr	0.8	29.2	9.2	1.2
dcdcmp	0.3	13.0	5.7	0.7
dcop	0.9	19.4	7.7	1.1
dctran	0.8	30.2	11.8	2.0
deseco	0.9	18.0	7.4	0.9
dgegv	0.4	29.8	7.1	0.9
dgesvd	2.5	41.1	19.1	1.8
dhgeqz	1.0	30.2	11.4	1.5
disto	0.7	14.8	5.3	1.3
dlatbs	0.4	17.6	7.2	0.9
dtgevc	0.7	32.5	12.0	2.3
dtrevc	0.8	25.3	9.3	1.3
elprnt	0.5	21.9	6.7	0.9
equilset	1.7	33.5	12.9	1.7
errchk	0.7	33.0	12.0	2.4
iniset	0.6	28.3	12.7	1.6
init	0.4	13.3	5.5	0.7
initgas	1.4	19.7	7.6	1.3
jsparse	1.7	30.5	11.0	1.3
modchk	0.9	27.2	11.0	1.7
moseq2	0.5	15.7	7.0	0.9
mosfet	1.1	22.2	8.5	1.3
noise	0.6	11.8	5.1	0.7
out	2.3	43.7	15.1	2.0
reader	1.6	20.7	7.8	0.8
readin	1.1	39.3	15.1	2.3
setupgeo	1.7	25.1	7.8	0.9
setuprad	2.2	21.3	7.8	0.9
smvgear	1.5	22.1	8.6	1.2
solveq	1.6	20.2	7.8	1.0
twldrv	1.2	33.4	7.3	1.4
Average	1.0	23.9	8.9	1.2

Table 3.2: Execution time for front-end parsing, for constructing flowgraphs, for computing immediator dominators using the LT algorithm, and for constructing DJ graphs.

Previously relations similar to J edges have been proposed to indirectly capture control flow properties of a flowgraph. For example, the DF_{local} relation of Cytron et al. [CFR+91] are equivalent to J edges. Cytron et al. define $DF_{local}(x)$ to be the set of all successor nodes y of x such that x does not strictly dominate y. From this definition we can see that an edge $x \rightarrow y$ is a join edge iff $y \in DF_{local}(x)$. In the DJ graph we explicitly represent the DF_{local} relation as join edges.

In this chapter we also gave our experimental framework and quantitatively studied the structural characteristics of DJ graphs. From our study we can see that the size of DJ graphs is only about 5.4% more than the size of the corresponding flowgraph. We also noticed that DJ graphs are flat structures.

Chapter 4

Computing Dominance Frontiers and Related Sets

A 'No' uttered from deepest conviction is better and greater than a 'Yes' merely uttered to please, or what is worse, to avoid trouble.

—Mahatma Gandhi

In this chapter we propose new algorithms for computing dominance frontiers and related sets using DJ graphs. We will show how DJ graphs can concisely capture some of these relations via D and J edges. Since dominance frontiers are related to control dependences [CFR+91], our algorithms can also be used for computing the control dependence relation. In Section 4.1, we give a simple algorithm for computing dominance frontiers of a node using DJ graphs. In Section 4.2, we show how to compute dominance frontiers for a set of nodes in linear time. In Section 4.3, we will show to compute the full dominance frontier relation using DJ graphs. In Section 4.4, we show how to compute dominance frontier intervals in linear time using DJ graphs. Finally, we will compare our work with other related work in Section 4.5.

4.1 Computing Dominance Frontiers for a Node

In this section we give a simple algorithm for computing $DF(x)$ of a node x using properties of DJ graphs. In the next section, we will extend this algorithm for computing dominance frontiers for a set of nodes in linear time. The key intuition

behind our algorithm is based on Theorem 3.2. From this theorem we know that the level number of all nodes in $DF(x)$ are less than or equal to the level number of node x. Now, is it possible to compute the dominance frontier of a node using level information? The answer is yes. The following lemma establishes a relation between nodes in the dominance frontier of a node x and their levels.

Lemma 4.1 *A node $z \in DF(x)$ iff there exists a $y \in SubTree(x)$ with $y \rightarrow z$ as a J edge and $z.level \leq x.level$.*

Proof:

The "if" part Here we have to show that if $y \in SubTree(x)$ and $y \rightarrow z$ is a J edge such that $z.level \leq x.level$, then z is in the set $DF(x)$. There are two cases:

Case 1: z is in $SubTree(x)$. Since $z.level \leq x.level$, z must be x itself. Also, since $y \rightarrow z$ is a J edge, y must be a predecessor of z in the corresponding flow graph. Furthermore, y is in $SubTree(x)$, hence z must be in $DF(x)$ (from the definition of dominance frontier).

Case 2: z is not in $SubTree(x)$. In this case x does not dominate (and hence does not strictly dominate) z. Now since $y \rightarrow z$ is a J edge, y is a predecessor of z in the corresponding flowgraph, from the definition of dominance frontier z is in $DF(x)$.

The "only if" part Since z is in $DF(x)$, by definition of dominance frontier, x does not strictly dominate z, and also there must be a node y that is a predecessor of z in the corresponding flowgraph such that x **dom** y. Since x dom y, we have $y \in SubTree(x)$. Also, by definition, $y \rightarrow z$ is a J edge. Now, using Theorem 3.2 , it is easy to see that $z.level \leq x.level$.

∎

Using Lemma 4.1 we can easily devise a simple algorithm for computing the dominance frontier of a node as follows:

Algorithm 4.1 *The following algorithm computes dominance frontier $DF(x)$ of a node x using DJ graphs.*

DomFrontier(x)
{
9: $DF_x = \emptyset$
10: **foreach** $y \in SubTree(x)$ **do**
11: **if**($y \rightarrow z == Jedge$)
12: **if**($z.level \leq x.level$)
13: $DF_x = DF_x \cup z$
}

Notice that the time complexity of the above algorithm is $O(|N| + |E|)$, in the worst-case. This is because, to compute dominance frontiers for a node x we will potentially visit all the nodes and edges (D edges + J edges) in the $SubTree(x)$.[1]

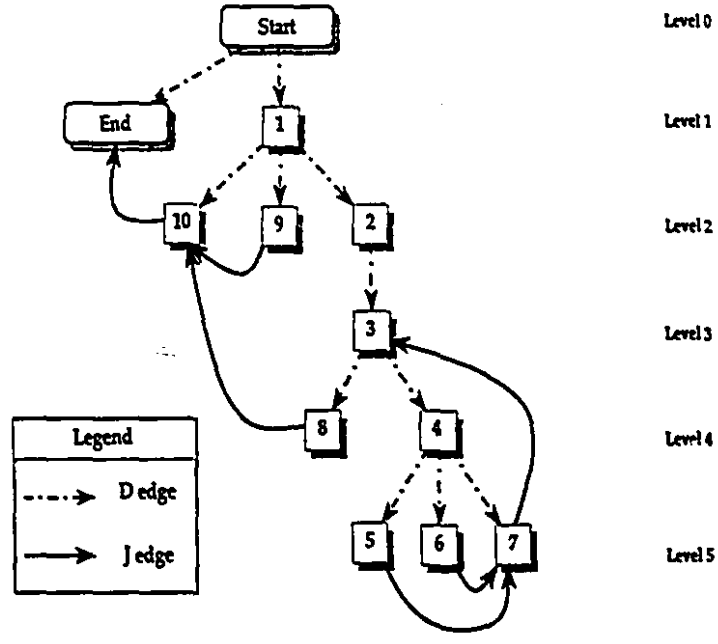


Figure 4.1: Figure 3.1 reproduced.

[1]Sometimes, for convenience, we will overload the notation $SubTree(x)$ to represent the subgraph of a DJ graph rooted at x that includes all the D edges and J edges induced by the nodes in the $SubTree(x)$.

Example 4.1

Consider the DJ graph shown in Figure 4.1. Let us compute $DF(3)$. To compute this, we simply walk down the DJ graph along D edges and look for J edges whose destination nodes are at levels $3.level$ or less. For the example DJ graph we can see that J edges $7 \rightarrow 3$ and $8 \rightarrow 10$ satisfies the level condition. Therefore, $DF(3) = \{3, 10\}$.

Given the above result next we will next show how to compute dominance frontiers for a set of nodes in linear time.

4.2 Computing Dominance Frontiers for a Set of Nodes

In the last section, we gave a simple linear time algorithm for computing the dominance frontier of a node. One way of computing dominance frontiers for a set S of nodes is to precompute the dominance frontier for every node in the flowgraph, and then use Equation (2.1) to compute dominance frontiers for the set S. This could give rise to a worst-case quadratic time behavior [CFR+91]. To illustrate this, consider the computation of $DF(\{2,4\})$ (Figure 3.1). From Equation (2.1), we know $DF(\{2,4\}) = DF(2) \cup DF(4)$. Let us therefore precompute $DF(2)$ and $DF(4)$. Using Algorithm 4.1 we get $DF(2) = \{10\}$ and $DF(4) = \{3\}$. So, $DF(\{2,4\}) = \{3,10\}$.[2] Notice in the above example that we visit the nodes in the $SubTree(4)$ twice—once during the computation of $DF(2)$, and once again during the computation of $DF(4)$. How can we avoid this redundant traversal of the nodes in the $SubTree(4)$? We can avoid this by first computing $DF(4)$ and marking node 4 as being processed. Now during the computation of $DF(2)$ we avoid visiting any node in $SubTree(4)$ (since node 4 is already processed, and is so marked) thereby avoiding redundant traversal. Notice here that we never need to precompute $DF(2)$ and $DF(4)$ in order to compute $DF(\{2,4\})$. Therefore, to compute $DF(\{2,4\})$, we first compute the $DF(4)$ using Algorithm 4.1, and also mark node 4 as being processed. Any candidate node that is generated on-the-fly is then added to the set $DF(\{2,4\})$. Now during the computation of $DF(2)$ we

[2]Cytron et al. have proposed a simple formula for computing the dominance frontiers for all nodes that is more efficient than our redundant traversal method. We will discuss this method in Section 4.5.

avoid visiting the nodes in $SubTree(4)$. Again we add any candidate node that is generated on-the-fly to $DF(\{2,4\})$. Based on this observation we can see that the ordering of nodes in the dominator tree is important to avoid redundant traversal of nodes during the computation of dominance frontiers. The complete algorithm for computing dominance for a set of node is given below

Algorithm 4.2 *Given a set S of nodes, the following algorithm computes $DF(S)$.*

♣ **Input:** DJ graph and the set S of nodes.
♣ **Output:** DF_S, the dominance frontier for a set S of nodes.
♣ **Initialization:** Order the nodes in S by their level numbers.

```
DFSet(S)
{
14:   DF_S = ∅
15:   foreach x ∈ S in a bottom-up fashion do
16:     Mark x as Visited.
17:     Walk down the SubTree(x) while avoiding nodes that have been previ-
      ously       marked Visited.
18:     Identify J edges y → z such that z.level ≤ x.level. Include all such z's
        in DF_S.
19:   endfor
}
```

We can see that the time complexity of the above algorithm is again $O(|N|+|E|)$. In Chapter 7 (Section 7.4) we will establish the correctness and complexity of a much stronger result, that of computing iterated dominance frontiers, which subsumes the correctness and the complexity of the above algorithm.

4.3 Computing the Full Dominance Frontier Relation

In previous sections we gave algorithms for computing dominance frontiers of a node and set of nodes, without precomputing the (full) dominance frontier relation for all the nodes in a flowgraph. In this section we will show how to compute the full dominance frontier relation using DJ graphs. Our method for computing the full dominance frontier relation is equivalent to the one proposed by Cytron et al. [CFR+91], except that we use level information instead of the dominance

relation. In this section we will also briefly discuss the recent result due to Pingali and Bilardi on the representation of the dominance frontier relation [PB95].

In [CFR+91], Cytron et al. gave a simple formula for precomputing dominance frontiers for all nodes [CFR+91]. The formula consists of two parts:

$$DF_{local}(x) = \{y \in Succ_J(x) | x \text{ !stdom } y\},$$

and

$$DF_{up}(z) = \{y \in DF(z) | idom(z) \text{ !stdom } y\}.$$

Now if $y \in Succ_J(x)$ and x !stdom y then $x \rightarrow y$ is a J edge (follows from the definition of DJ graph). Also if $y \in DF(z)$ and $idom(z)$!stdom y then $y.level \leq idom(z).level$ (follows from Theorem 3.2). Therefore we can rewrite the above formulas using level information.

$$DF_{local}(x) = \{y | x \rightarrow y \text{ is a J edge}\},$$

and

$$DF_{up}(z) = \{y \in DF(z) | y.level \leq idom(z).level\}.$$

Using DF_{local} and DF_{up} Cytron et al. proposed the following formula for computing dominance frontiers for each node x.

$$DF(x) = DF_{local}(x) \cup \bigcup_{z \in Children(x)} DF_{up}(z). \tag{4.1}$$

Example 4.2

Consider the DJ graph shown in Figure 3.1. Assume that the dominance frontier for nodes 5, 6, and 7 have been computed (i.e., $DF(5) = \{7\}$, $DF(6) = \{7\}$, and $DF(7) = \{3\}$). We will show how to compute $DF(4)$ using the above recursive formula. First notice that $DF_{local} = \emptyset$ (since there are no J edges coming out of node 4). The set $DF(4) = DF_{local}(4) \cup DF_{up}(5) \cup DF_{up}(6) \cup DF_{up}(7)$. Using the above formula we can see that $DF_{up}(5) = DF_{up}(6) = \emptyset$, whereas $DF_{up}(7) = \{3\}$. Therefore $DF(4) = \{3\}$.

Once the full dominance frontier relation is computed the next concern is how to store or represent it. This problem is generally called as the *factorization* problem [CFS90b]. As motivation, let us first see how to factorize the dominance relation.

The dominance relation can be represented in two ways: (1) at each node store a pointer to its immediate dominator node (except for the *Root* node where we store a pointer to itself), (2) at each node x store a list of all the nodes that strictly dominate x. Using either representation we can query the dominator of a node in time proportional to the size of its dominator set. But, the first factorization (or representation) is better since it occupies only linear space, whereas the second factorization occupies quadratic space.

Given the above intuition for the factorization problem, Cytron et al., in [CFS90b], posed the following open problem. Is there a factorization (or a representation scheme) for dominance frontiers that can be constructed in linear time, occupies linear space, and each query $DF(x)$ takes time proportional to the size of x's dominance frontier set. In that paper, the authors conjectured that it may not be possible to come up with such a factorization. Notice that the full dominance frontier relation, in the worst-case, occupies quadratic space (e.g., nested `repeat-until` loops), but querying takes time proportional to the size of the set.

DJ graphs can also be considered as a factorization for representing dominance frontiers. We can construct DJ graphs in linear time (Algorithm 3.1), its size is again linear with respect to the size of its flowgraph (Theorem 3.1), but querying the dominance frontier of a node takes $O(|E|)$ using Algorithm 4.1. Notice that our DJ graph has space optimality (since it occupies linear space) but not query-time optimality (since it takes $O(|E|)$ for a query), whereas the full dominance frontier relation has query-time optimality (since it takes $O(|DF(x)|)$ for querying the dominance frontier of node x), but not space optimality (since it occupies quadratic space, in the worst case).

Recently, Pingali and Bilardi solved this problem using a representation called *APT* [PB95]. To motivate the *APT* representation consider the DJ graph shown in Figure 4.1. Assume that we want to compute $DF(2)$. Using Algorithm 4.1, we would simply walk down the DJ graph along D edges looking for J edges whose destination node is at levels less than or equal to the level of node 2. We can see

that only $8 \rightarrow 10$ satisfies this level condition. Now rather than walking down the DJ subgraph root at 2 each time we query $DF(2)$ we can *cache* the J edge $8 \rightarrow 10$ at node 2. By doing so we improve the query time of $DF(2)$. Now the key question to ask is when and where to cache such J edges? Recently Pingali and Bilardi solved this problem in their $\mathcal{APT}$ representation. In a preprocessing step they show how to cache such J edges at certain nodes called *boundary nodes*. One can view the $\mathcal{APT}$ representation to be a *cached* DJ graph, where J edges are cached to improve query time. Pingali and Bilardi showed how to used a "tuner" to control how much caching is really needed so that space and time optimality of the representation is not sacrificed. One can think of $\mathcal{APT}$ to be a spectrum of dominance frontier factorizations with our DJ graph being at one end (with no caching) and the full dominance frontier representation being at the other end (with full caching).

4.4 Dominance Frontier Interval

In this section we give a simple algorithm for computing the **dominance frontier interval** in linear time using DJ graphs [CFS90b, PB95]. We define dominance interval as follows:

Definition 4.1 *Let $y \rightarrow z$ be a J edge and let $w = idom(z)$.*

- *The* **half-open dominance frontier interval** $[y, w)$ *of the J edge $y \rightarrow z$ is the set of all the nodes on the reverse dominator tree path from y to w, including w.*
- *The* **closed dominance frontier interval** $[y, x]$ *of the J edge $y \rightarrow z$ is the set of all the nodes on the half-open dominance interval $[y, w)$ but not including w.*

Given the above definition, we call y as **Bottom** node of the interval, w is called the **Top** node of the half-open interval, and x is called the **Top** node of the closed interval. Determining the set of nodes in $[y, w)$ or $[y, x]$ requires a simple tree walk on the reverse dominator tree path from y to w or x.

Our interest in this dissertation is in the **Top** node of a dominance frontier interval (see Chapter 10). For a half-open interval, computing **Top** node is trivial—the **Top** node of a J edge $y \rightarrow z$ is nothing but the immediate dominator of z. What about the **Top** node of a closed interval? A naive algorithm would require walking

up the reverse dominator tree path from y to $w = idom(z)$ and find the immediate dominee x of w on this path. The complexity of this would be $O(|N|)$ for a single J edge, and so for all J edges this would require $O(|E| \times |N|)$. We will next show how to compute the same set in time $O(|E|)$ using DJ graphs.[3]

Algorithm 4.3 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 4.3 *The following algorithm determines* **cTop** *nodes for all J egdes.*

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

```
MainTop()
{
20:   foreach c ∈ Children(START)
21:     START.curChild = c
22:     TopDFS(c).
23:   endfor
}
TopDFS(y)
{
24:   foreach outgoing J edge y → z do
25:     y.cand = idom(z).curChild;
26:   endfor
```

[3] Although the algorithm is very simple, we are not aware of any literature that gives a complete algorithm for determining the **Top** node of closed intervals. Pingali and Bilardi briefly mention this in their paper without providing a complete algorithm [PB95]. Subsequently, Pingali also proposed another algorithm for computing the same set [Pin95].

```
27:    foreach c ∈ Children(y)
28:      y.curChild = c /* a Top node of some closed interval */
29:      TopDFS(c).
30:    endfor
}
```

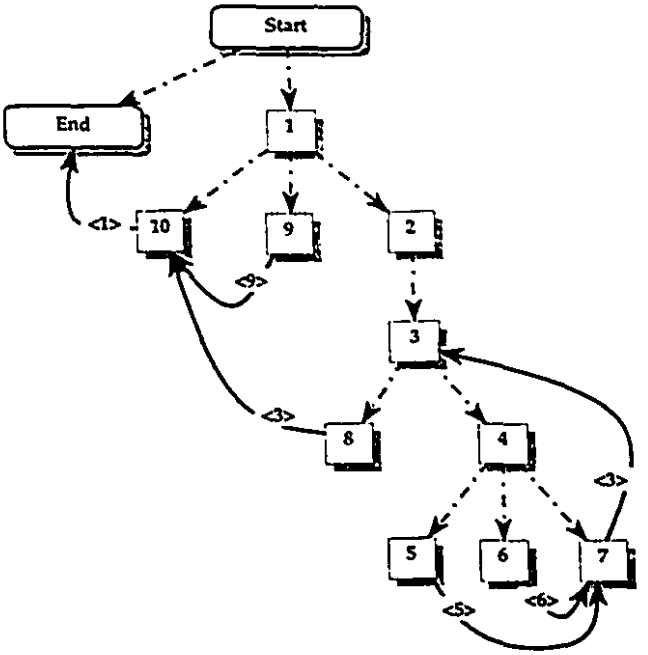


Figure 4.2: The DJ graph of Figure 3.1 annotated with **cTop** nodes.

Example 4.3

Consider the DJ graph shown in Figure 3.1. We will illustrate Algorithm 4.3 for this DJ graph. The DJ graph annotated with cTop nodes is shown in Figure 4.2. The top nodes are denoted in the figure as $< x >$. We perform a top-down depth-first search on the DJ graph via D edges looking for J edges. During the depth-first traversal (step 22 and step 29) we store a reference to the current child (through which visit the nodes in sub-tree rooted at the current child) in the parent node (step 21 and step 28). Subsequently, when we probe a J edge $x \rightarrow y$, the immediate dominator of y will contain the $\mathbf{cTop}_{x \rightarrow y}$ (step 25).

Consider for example the J edge $7 \rightarrow 3$, the immediate dominator of 3 is 2. The only dominator tree path from 2 to 7 is via node 3, and a reference to node 3 was previously stored in $2.curChild$. So the cTop_{7_3} is 3.

Theorem 4.1 Algorithm 4.3 *correctly computes* **cTop** *nodes for all J edges.*

Proof:

At step 22 and step 29 **TopDFS**(c) is invoked with the current child c. Prior to calling **TopDFS**(c), reference to the current child c is stored in its parent node. Since the path between any two nodes in a dominator tree is unique, when a J edge $y \rightarrow z$ is processed at step 25, $w = idom(z)$ will contain the reference to child node through which node y was previously visited from node w. This child is the **cTop** for $y \rightarrow z$. This is because there is a unique path between w and y on the dominator tree, and this path must pass through cTop_{y_z} ∎

Finally, one can easily show that the time complexity of the above algorithm is $O(|E_d|)$.

4.5 Discussion and Related Work

In this chapter we gave algorithms for computing dominance frontiers of a node and dominance frontiers for a set of nodes without precomputing the dominance frontiers for all nodes. We also gave an algorithm for computing the full dominance frontier relation that uses level information. One main contribution in this chapter is determining dominance frontiers using level information. This is important for us because it allowed us to devise a simple linear time algorithm for computing dominance frontiers for a set of nodes (Chapter 7).

In [CFR+91] Cytron et al., gave a simple formula for precomputing the dominance frontiers for all nodes. Cytron et al. proposed a factorization for storing the dominance frontiers for all nodes. This factorization occupies quadratic space, but querying the dominance frontier of a node takes time proportional to the size of its dominance frontier set. We can also think of DJ graphs to be a factorization of dominance frontiers. In our case we can construct a DJ graph in linear time and

linear space, but querying the dominance frontier of a node using Algorithm 4.1 could take $O(|N|)$ time. Recently, Pingali and Bilardi proposed a representation called $\mathcal{APT}$ for factorizing the dominance frontier relation. One can think of $\mathcal{APT}$ to be a spectrum of dominance frontier factorizations with our DJ graph being at one end and the complete dominance frontier representation being at the other end.

In this chapter, we also gave a simple algorithm for computing **Top** node of both closed and open dominance frontier intervals for all J edges. Although this algorithm is very simple, we have not seen any literature giving a description of an algorithm for this problem. We will use the concept of top nodes in Chapter 10.

Chapter 5

Computing Multiple Node Immediate Dominators

Consciously or unconsciously, every one of us does render some service or other. If we cultivate the habit of doing this service deliberately, our desire for service will steadily grow stronger, and will make, not only our own happiness, but that of the world at large.

—Mahatma Gandhi

Recently, Gupta introduced the concept of multiple-node immediate dominators for solving certain data flow problems such as array bound checking and loop invariant removals more aggressively than exiting methods [Gup92]. In his paper Gupta gave an $O(|N|^p)$ algorithm for computing multiple-node immediate dominators (where $|N|$ is the number of flowgraph nodes and p is the maximum number of predecessors of a node). In this chapter we present an $O(|E|^2)$ algorithm for computing the same result (where $|E|$ is the number of flowgraph edges) using DJ graphs. In the next section we introduce the concept of multiple node dominators, and also discuss one application in compiler optimization. In Section 5.2, we present our algorithm for computing the multiple-node immediate dominance relation using DJ graphs. In Section 5.3, we use an example to further illustrate our algorithm. In Section 5.4, we show the correctness and complexity of our algorithm. Finally, in Section 5.5, we discuss some related work and give our concluding remarks.

5.1 Introduction and Motivation

Recently, Gupta introduced a relation called multiple-node immediate dominators of a node [Gup92]. This relation can be used in automatic generation of compact test suites for program testing. It is also useful in program analysis and optimization. Using multiple-node immediate dominators, he showed how to perform loop-invariant removals and array bound checking more aggressively than existing methods [ASU86]. More recently, Gupta [Gup95], Bodik and Gupta [BG95], Appelbe et al. [AHM+95] have shown other applications of multiple-node dominance relation.

To illustrate one application of multiple node dominance relation, consider the flowgraph shown in Figure 5.1.[1] If one uses the traditional algorithm for loop invariant removal, it is impossible to move the expressions `i+1` (node 3 and node 4 in Figure 5.1) from the loop. This is because, single-node dominance relation prohibits such optimizations [ASU86]. To overcome this, Gupta introduced the notion of multiple-node dominance relation. Intuitively, in a multiple-node dominance relation a group of nodes dominate a single node such that no subset of the nodes dominates the node. For example, in Figure 5.1 nodes 3 and 4 together dominate node 5, but nodes 2 and 3 together do not dominate node 5 (there exists a subset of $\{2,3\}$, i.e., $\{2\}$, that dominates node 5).

Gupta proposed a two-step process for computing multiple-node dominance relation. In the first step *multiple-node immediate dominator nodes* are computed, using which, in the second step, the multiple-node dominance relation is computed. Gupta's algorithm for computing the multiple-node immediate dominance relation has a worst-case time complexity of $O(|N|^p)$, where p is the maximum number of predecessors of a node. We have improved the worst-case time complexity of the algorithm to $O(|E|^2)$.

Next we will formally introduce multiple-node dominance relation. The traditional definition of the dominance relation is called the single-node dominance relation, meaning every node except for the START node has exactly one immediate dominator. The generalized dominance relation captures both the single-node and the multiple-node dominance relation in a unified way. Gupta defines the generalized dominance relation as follows:

[1] This example is taken from [Gup92].

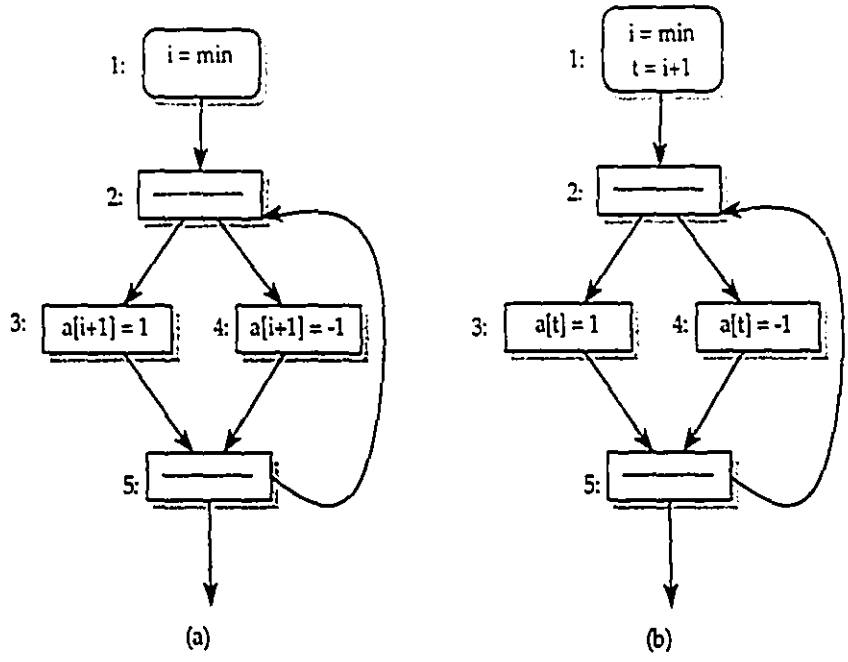


Figure 5.1: An example of loop invariants removal.

Definition 5.1 *A set of nodes S dominates node x if and only if:*

(1) *all paths from* START *to x contain some node $y \in S$; and*

(2) *for each $y \in S$, there is at least one path from* START *to x which contains y but does not contain any other node in S.*

Example 5.1

Consider the flowgraph shown in Figure 5.2. Let $S = \{2, 5, 6\}$. We can see that S dominates node 15. This is because all paths from START to 15 contain at least one node in S, thus S satisfies the first condition of Definition 5.1. Also, we can see that for each node $y \in S$, there is at least one path from START to 15 that passes through y, but not any other node in S, and so S also satisfies the second condition.

Now let $T = \{4, 8, 12\}$. We can see that T does not dominate node 15. This is because a subset of T, i.e., $\{4, 12\}$, dominates node 15. Thus T violates the second condition of Definition 5.1.

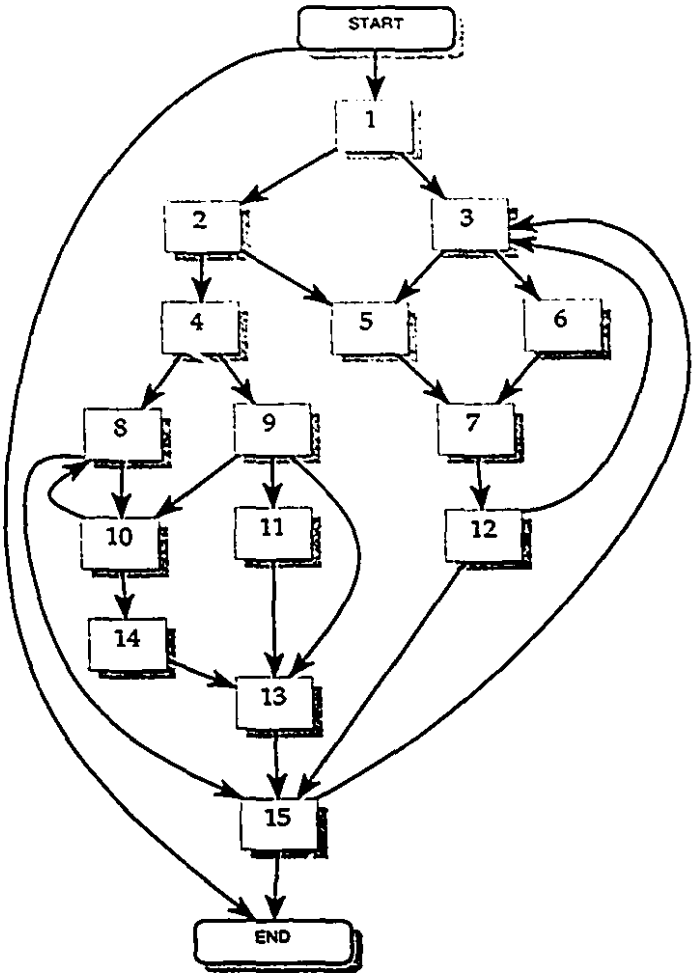


Figure 5.2: Another example of a flowgraph.

Gupta presented a two-step process for computing multiple-node dominance relation. First, multiple-node *immediate* dominator set is computed, using which the multiple-node dominance relation is computed. Gupta defines the multiple-node immediate dominance relation as follows. (Note that $Pred_f(x)$ represents the predecessor nodes of x in a flowgraph.)

Definition 5.2 *A multiple-node immediate dominator set $midom(x)$ of a node x is defined to be a subset of $Pred_f(x)$ which dominates x.*

Example 5.2

Consider the flowgraph shown in Figure 5.2. Let us compute $midom(13)$. First notice that $Pred_f(13) = \{9, 11, 14\}$. Using Definition 5.2, we can immediately see that $midom(13) = \{9, 14\}$.

In this chapter we will distinguish between single node immediate dominator ($sidom$) and multiple-node immediate dominator ($midom$) set. Every node has exactly one $sidom$ (except the START node which has none). The $midom$-set of a node, although unique, can be empty. The $midom$-set of a node is empty whenever the node satisfies the following property:

Lemma 5.1 *The midom-set of a node x ($x \neq$ START) is empty if and only if there is a flowgraph edge from $sidom(x)$ to x.*

Proof:

Easily follows from Definition 5.2. ∎

Therefore, we will compute the $midom$-set only for those nodes x such that $sidom(x) \rightarrow x$ is not a flowgraph edge. We will use $\mathcal{MIDOM}$ to denote the *relevant* set of nodes whose $midom$-sets are *not empty*.

In this chapter we give a simple algorithm for computing multiple-node immediate dominator with a better worst-case time complexity of $O(|E|^2)$. To compute the multiple-node immediate dominance relation we made the following key observation:

Observation 5.1 *A predecessor, y, of a node x in a flowgraph is in $midom(x)$ iff there exists at least one path from $sidom(x)$ to y that does not contain any other nodes in $Pred_f(x)$ (excluding y itself).*

This key observation follows from the definition of generalized dominator (Definition 5.1) and the restriction that $midom(x)$ must be a subset of $Pred_f(x)$. Therefore, in order to compute $midom(x)$ we check for each $y \in Pred_f(x)$ whether there is a path from $sidom(x)$ to y that does not contain any other nodes in $Pred_f(x)$. If such a path exists then we add y to $midom(x)$. Thus, in our approach, we start off with an empty $midom(x)$, and add nodes from $Pred_f(x)$ to $midom(x)$ if they satisfy the aforementioned property. This is in contrast to Gupta's method, where the $midom(x)$ set is initially assumed to be $Pred_f(x)$, and then nodes are removed from $midom(x)$ until the set satisfies certain properties (Lemma 2 in [Gup92]). Based on this observation, in the next section, we give a simple algorithm for computing the set of nodes in $midom(x)$.

5.2 Our Algorithm

In this section we present our algorithm for computing multiple-node immediate dominators. In the previous section we made the following key observation for computing multiple-node immediate dominators: If a node y is in $midom(x)$, then there is at least one path from $sidom(x)$ to y that does not contain any other nodes in $Pred_f(x)$ (excluding y itself). Therefore, in order to compute $midom(x)$ we check for each $y \in Pred_f(x)$ whether there is a path from $sidom(x)$ to y that does not contain any other nodes in $Pred_f(x)$.

In order to check if the above mentioned path exists, we do the following: For each node $y \in Pred_f(x)$ (assuming that we are computing $midom(x)$), we 'outblock' all the nodes in $Pred_f(x)$ except y. By 'outblock' we mean that all the outgoing edges from the node are conceptually cut. Next we check if x is reachable from $sidom(x)$. If so, we have found a path from $sidom(x)$ to x that does not pass through any other node except y in $Pred_f(x)$. This means that $y \in midom(x)$. We do the above process for all nodes in $Pred_f(x)$. The complete algorithm for computing $midom(x)$ for every node $x \in MIDOM$ is given below.

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

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

$struct\ NodeStructure\{$
$proc = \{Processed, NotProcessed\}$
$instack = \{InStack, NotInStack\}$
$outblocked = \{OutBlocked, NotOutBlocked\}$
$level = \{0 \ldots NumLevel - 1\}$
$\}$

- **PushNode**(x) inserts x into a *stack*. **PopNode**() retrieves a node from the stack. **ClearStack**() clears the stack.

The first step in the algorithm is to mark all the predecessor nodes $Pred_f(x)$ of x as $OutBlocked$ in the DJ graph (**for** loop at step 32). Then select one of the 'un-processed' nodes that is marked $OutBlocked$ and un-block it (step 37). Next call the function **VisitedX**($sidom(x), x$) (step 38). This function returns $True$ if a path exists from some node in $Succ_f(sidom(x))$ to x that does not pass through any of the outblocked nodes; otherwise, it returns $False$ (step 38). Add y to M_x (step 40) if the function **VisitedX**() returned $True$. At step 43 we again outblock the current node. This process is repeated for all un-processed nodes in $Pred_f(x)$. When the procedure terminates the *midom*-set for node x is stored in M_x. The **for** loop at step 46 un-blocks all outblocked nodes.

Algorithm 5.1 *The following algorithm computes the midom-set M_x for any $x \in$ MIDOM.*

♠ **Input:** A DJ graph $\mathcal{G} = (N, E, \text{START}, \text{END})$ and the set of relevant nodes *MIDOM*

♠ **Output:** The set M_x, the multiple-node immediate dominators for a node $x \in MIDOM$.

♠ **Initialization:**

$\forall x \in N\ (x.proc = NotProcessed\ ;$
$x.instack = NotInStack\ ;$
$x.outblocked = NotOutBlocked\ ;$
/ * Compute the level numbers * /
$x.level = Level(x)\ ;$

♠ **The Algorithm:**

```
Procedure MidomSet(x)
{
31:   M_x = ∅
32:   foreach y ∈ Pred_f(x)
33:     y.outblocked = OutBlocked
34:   endfor
35:   foreach y ∈ Pred_f(x)
36:     if(y.proc! = Processed) /* get next un-processed node */
37:       y.outblocked = NotOutBlocked
38:       status = VisitedX(sidom(x), x)
39:       if(status == True) /* found a path */
40:         M_x = M_x ∪ y /* insert in M_x */
41:       endif
42:       y.proc = Processed/* mark processed*/
43:       y.outblocked = OutBlocked
44:     endif
45:   endfor
46:   foreach y ∈ Pred_f(x)
47:     y.proc = NotProcessed /* recover the DJ graph*/
48:     y.outblocked = NotOutBlocked
49:   endfor

}
```

The function **VisitedX()** in conjunction with another function **Visit()** searches through the nodes below $sidom(x)$ to see if it can reach x from some node w in $Succ_f(sidom(x))$ without passing through any of the outblocked nodes. The procedure **VisitedX()** returns $True$ if such a path exists. Remember that we have unblocked only one predecessor of x, say y, and outblocked the rest. So if **VisitedX()** returns true, we have found a path from $sidom(x)$ to x that passes through y without passing through any other predecessors of x. Therefore, from our previous discussion y should be added to the set $mdiom(x)$.

The first step in function **VisitedX()** is to insert into a *stack* all the nodes in $Succ_f(sidom(x))$ that are at the same level as x and are not marked $OutBlocked$ (for loop at step 50). Then it picks a node z from the stack (step 56) and calls

another function **Visit**(z, x) (step 57). The function **Visit**() returns $True$ if the node x was visited during the invocation of **Visit**(z, x); otherwise, it returns $False$ (step 57). When the $True$ value is returned, further search through any remaining nodes in stack is aborted and the $True$ value is propagated back to the main procedure **MidomSet**() (step 60). Otherwise, it picks another node from the stack and continues with the search until there are no more nodes in the stack. Whenever **Visit**() returns $True$, the function **VisitedX**() clears the stack (by calling the procedure **ClearStack**() at step 59) before propagating the $True$ value back to the main procedure.

```
Function VisitedX(w, x)
{
50:   foreach z ∈ Succ_f(w)
51:     if((z.level == x.level) and
                (z.outblocked! = OutBlocked))
52:       PushNode(z) /* Push onto stack */
53:       z.instack = InStack
54:     endif
55:   endfor
56:   while((z = PopNode())! = NULL)
57:     status = Visit(z, x)
58:     if(status == True)
59:       ClearStack()
60:       return True /* found a path */
61:     endif
62:   endwhile
63:   return False /* not found a path */

}
```

The function **Visit**(z, x) basically walks down $SubTree(z)$ through D edges (step 66) and checks if there is a J edge from some node $v \in SubTree(z)$ to x (where z was previously put on the stack). If such a J edge is found (step 72), further search down the subtree is aborted and the function returns $True$ back to function **VisitedX**() (step 73). Otherwise, the function visits all the nodes in

$SubTree(z)$ through D edges. Note that the function **Visit()** never walks down an outblocked node (step 65). Whenever it finds a successor node to be outblocked, it avoids searching further down the tree through the outblocked node.

As it walks down the subtree, if it finds a J edge whose destination node, say u, is at the same level as x, and u was not previously put on the stack, it pushes u onto the stack (step 77). Note that it never pushes an outblocked node onto the stack.

```
Visit(z, x)
{
64:   foreach u ∈ Succ_d(z)
65:     if(u.outblocked! = OutBlocked)
66:       if(z → u == Dedge) /* walk down D edge */
67:         status = Visit(u, x)
68:         if(status == True)
69:           return True /* found a path, so stop searching
    further */
70:         endif
71:       else/* Jedge */
72:         if(u == x) /* found path to x */
73:           return True
74:         endif
75:         if(u.level == x.level) /* same level as x? */
76:           if(u.instack! = InStack)
77:             PushNode (u)
78:             u.instack = InStack
79:           endif
80:         endif
81:       endif
82:     endif
83:   endfor
84:   return False
}
```

5.3 An Example

Let us illustrate the working of Algorithm 5.1 for the flowgraph given in Figure 5.2(a). Its corresponding DJ graph is shown in Figure 5.3. Assume that we want to compute $midom(13)$, the multiple-node immediate dominator set for node 13. The first step is to outblock all the nodes in $Pred_f(13) = \{9, 11, 14\}$ in the DJ graph (**for** loop at step [32]). The resulting DJ graph is shown in Figure 5.4(a). In the figure all outblocked nodes are shaded and not shadowed. Next we unblock one of the previously blocked node, say 11. The resulting DJ graph is shown in Figure 5.4(b). Then we call the function VisitedX(4, 13) (step [37]).

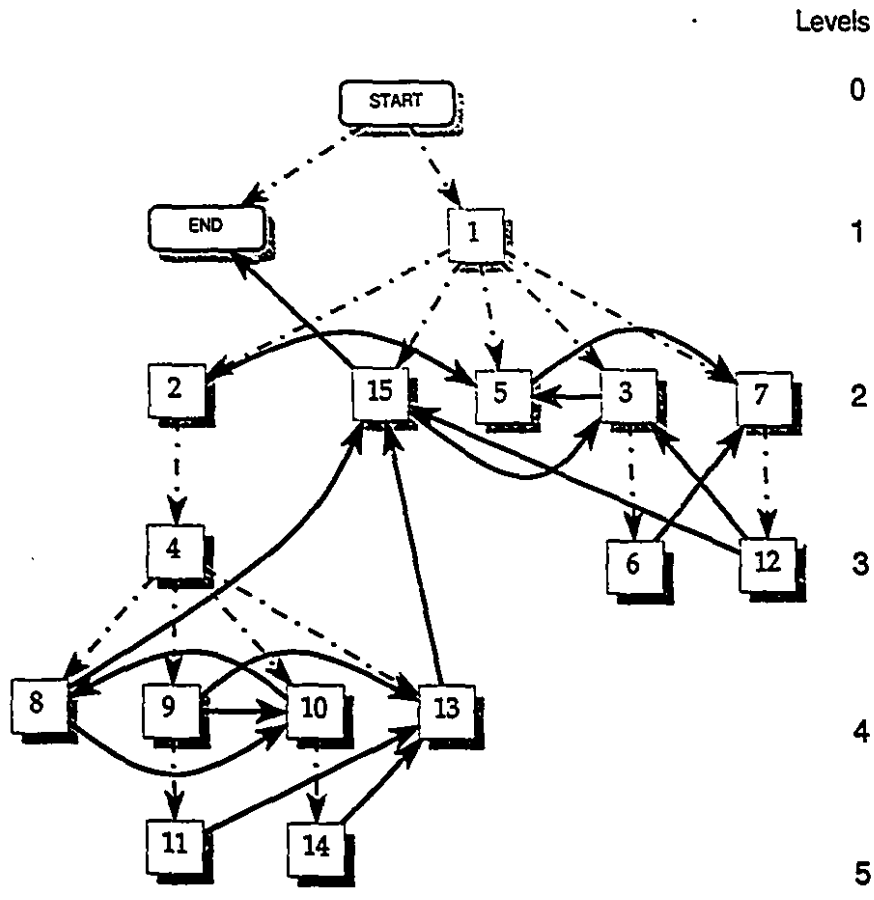
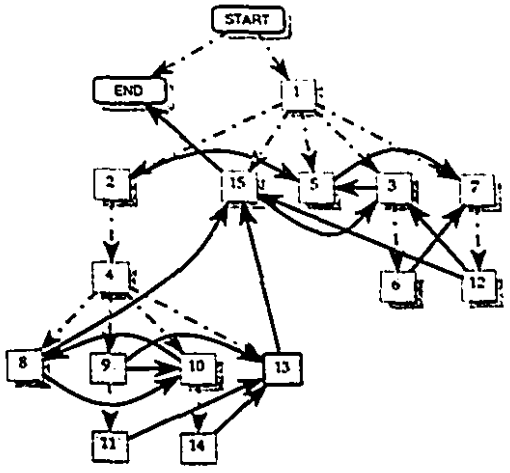
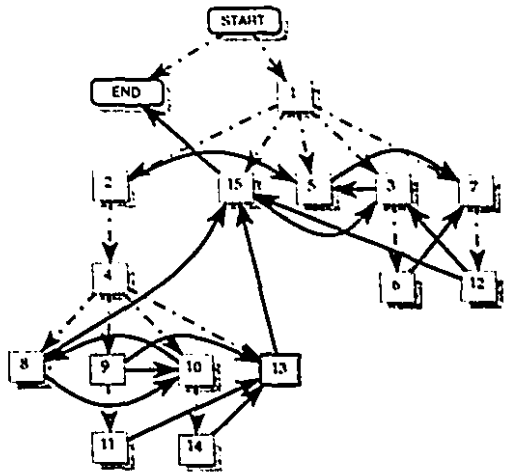


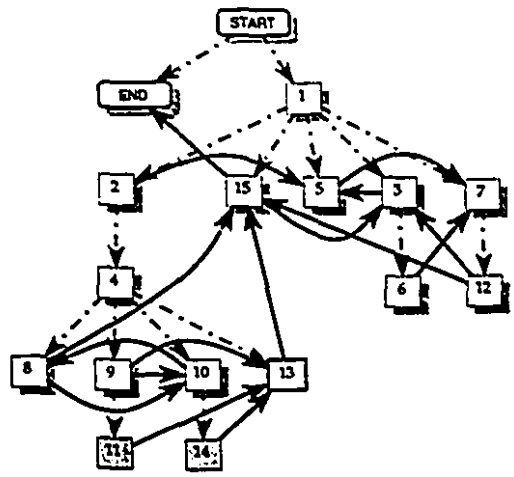
Figure 5.3: The DJ graph of the example flowgraph shown in Figure 5.2.



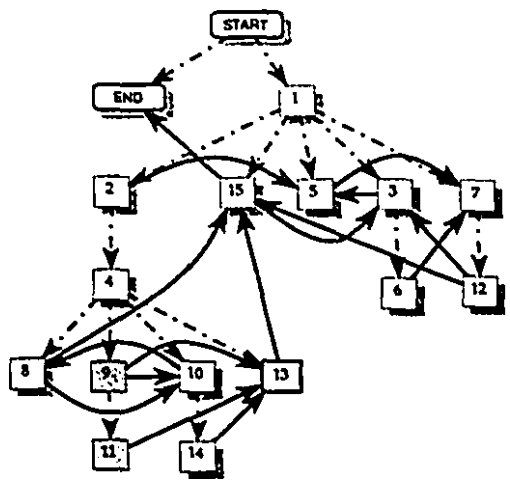
(a) All predecessor nodes are outblocked



(b) Only 9 and 14 are outblocked



(c) Only 14 and 11 are outblocked



(d) Only 9 and 11 are outblocked

Figure 5.4: Various stages of the DJ graph during the computation of *midom*(13). Outblocked nodes are shaded and not shadowed.

VisitedX(4, 13) first inserts into a stack all the nodes in $Succ_f(4)$ that are at the same level as node 13 and are not marked *OutBlocked* (**for** loop at step [50]). In this case only node 8 is pushed onto the stack, since node 9 is marked *OutBlocked*. Then we call **Visit**(8, 13). This function will in turn visit all the nodes in $SubTree(8)$ (step [64]), which contains only node 8. There are two J edges from 8: $8 \rightarrow 10$ and $8 \rightarrow 15$. The level number of node 15 is less than that of 13, so we do nothing. Node 10 is not outblocked, is at the same level as node 13, and was not previously put on the stack. So we push node 10 on the stack (step [77]). Since we have not yet reached our goal (of getting to node 13), **Visit**() returns $False$ back to its calling function **VisitedX**() (step [57]). Since node 10 is on the stack, at step [56] we pop the node and call **Visit**(10, 13) at step [57]. The only successor of node 10 is 14, and 14 is marked outblocked (step [65]). So **Visit**(10, 13) returns $False$, and so does **VisitedX**(4, 13) to the main procedure (since there are no more nodes on the stack). Consequently, we do not add node 11 to M_{13}.

Next, in the main procedure **MidomSet**(), we outblock node 11 and un-block node 9 (see Figure 5.4(c)). We again call the function **VisitedX**(4, 13). As before, **VisitedX**(4, 13) first pushed nodes 8 and 9 onto the stack (note that this time we have un-blocked node 9). Assume that node 9 was pushed earlier than node 8. At step [57] we call **Visit**(8, 13). The only successors of 8 are nodes 10 and 14. The scenario is the same as before—from node 10 we visit node 14 which is outblocked (the level number of node 14 is less than that of 13.) Therefore, we return with $False$ back to **VisitedX**(). Next the function **VisitedX**() picks node 9 from the stack (step [56]) and invokes **Visit**(9, 13). There are three successor nodes to node 9: nodes 11, 10, and 13. Node 11 is outblocked and node 10 was previously put on the stack. So we visit node 13, which is our goal state. Therefore we immediately return with $True$ (step [73]). This means we have found a path from 9, a node in $Succ_f(sidom(13))$, to node 13 that avoids any of the outblocked nodes. Consequently, we add node 9 to M_{13} (step [40]).

The only remaining un-processed node is 14. We un-block it and call the function **VisitedX**(4, 13) (see Figure 5.4(d)). Performing the computation process as before, we will see that this function returns $True$; therefore, we include node 14 in M_{13}. Thus, we eventually have $M_{13} = \{9, 14\}$.

5.4 Correctness and Complexity

In order to prove that Algorithm 5.1 correctly computes the *midom*-set for any node $x \in \mathcal{MIDOM}$, we must show that, when the algorithm terminates, the set M_x (at step 40) satisfies the definition of *midom*-set (Definition 5.2). In other words, we have to show that M_x satisfies the following conditions:

1. M_x is a subset of $Pred_f(x)$.

2. All paths from START to x contain some node $y \in M_x$, and for each $y \in M_x$, there is at least one path from START to x which contains y but does not contain any other node in M_x.

Notice that these two conditions follows from our key observation presented in Section 5.1 (Observation 5.1). Although we do not explicitly prove the key observation we will show that the above two conditions are indeed satisfied by our algorithm.

First of all notice that all paths from START to x must pass through $sidom(x)$. Also, all paths from START to any node y in $Pred_f(x)$ must again pass through $sidom(x)$ [PM72]. Therefore, it is sufficient for us to consider paths from $sidom(x)$ (rather than from START) to x when arguing that M_x indeed satisfies Definition 5.2. Before proceeding, we define a special set of nodes with respect to a node x:

Definition 5.3 (S_x set) *Given a node x, we define S_x to be a subset of $Succ_f(sidom(x))$ such that for every $y \in Succ_f(sidom(x))$, $y.level == x.level$.*

In order to show that M_x indeed satisfies the definition of *midom*-set, we will first examine what nodes are added to M_x when our algorithm terminates. A node is added to M_x at step 40 only if VisitedX() returns *True*. When does VisitedX() return *True*? In Lemma 5.5 we will show that Visited() returns *True* iff there exists a path from some node $y \in S_x$ to x in the DJ graph that does not pass through any outblocked node. What this means is that for a node y to be in M_x, there must exist a path from $sidom(x)$ to x that does not pass through any other node in $Pred_f(x)$.

We will use Theorem 5.1 to formally validate the correctness of our algorithm. The proof of the theorem is based on Lemma 5.5, whose proof in turn needs the results from the next three lemmas.

Lemma 5.2 *Let x and y be any two nodes such that $y.level < x.level$. Then every path from y to x must pass through $sidom(x)$.*

Proof:

Easily follows from properties of the DJ graph. ∎

In the following discussions, we assume that x is in the relevant set $MIDOM$. Lemma 5.3 shows that, in order to search for an outblock-free path from $sidom(x)$ to x, it is sufficient to start the search from nodes in S_x, thus reducing the search space.

Lemma 5.3 *All simple paths (in the flowgraph) from $sidom(x)$ to x must contain some node $y \in S_x$.*

Proof:

Assume P is a simple path (in the flowgraph) from $sidom(x)$ to x that does not contain any node in S_x. Notice that all paths from $sidom(x)$ to x must contain some node in $Succ_f(sidom(x))$. Let $u \in Succ_f(sidom(x))$ be some node such that u is in the path P. First of all observe that $u.level$ can never be greater than $x.level$. Now let us assume that $u.level$ is strictly less than $x.level$. But, from Lemma 5.2, all paths from u to x must pass through $sidom(x)$, and so does P. If this is the case, then P is no longer a simple path—a contradiction. So it must be the case that $u.level = x.level$, that is, $u \in S_x$. ∎

In an attempt to search for a path from $sidom(x)$ to x, we use Lemma 5.2 to guarantee that it is safe not to search any nodes whose levels are less than $x.level$. In other words, we can limit the search for such a path to only nodes below $sidom(x)$. Lemma 5.3 specifies exactly which subset of nodes of $Succ_f(idom(x))$ one needs to consider when determining if node x can be reached from $sidom(x)$. The needed nodes in $Succ_f(idom(x))$ are those at the same level as x. The next lemma specifies which nodes need to be pushed onto the stack (at step 52 and step 77). Those nodes are at the same level as x and are reachable through a outblock-free path from some node in $Succ_f(idom(x))$.

Lemma 5.4 *A node z is pushed onto the stack (at step 52 and step 77) only if there is a outblock-free path to z from some node in S_x, z is not outblocked, and $z.level = x.level$.*

Proof:

Nodes are put on the stack only at two places: step [52] and step [77]. At step [52] only nodes in S_x that are not outblocked are pushed onto the stack. At step [77] nodes are put on the stack when they are not outblocked, are siblings of x, and are reachable from some node $u \in SubTree(w)$, where w was previously put on the stack and is reachable from some node in S_x.

∎

Lemma 5.5 specifies that the function VisitedX($sidom(x), x$) does indeed find a path from some node in S_x to x that does not pass through any outblocked nodes, whenever such a path exists.

Lemma 5.5 *The function* VisitedX($sidom(x), x$) *returns* $True$ *if and only if there exists a path from some node* $y \in S_x$ *to* x *in the DJ graph that does not pass through any outblocked node.*

Proof:

The "if" part We first show that if VisitedX($sidom(x), x$) returns $True$, then there is outblock-free path from some node $y \in S_x$ to x. Note that we never visit any successor of an outblocked node in the process (step [65]). From Lemma 5.4 we know that a node is put on the stack only if it is not outblocked and there is a outblock-free path from some node y in S_x to this node. At step [73] we return $True$ only if x is visited through some J edge $z \rightarrow x$ (step [72]). Node z was previously put on the stack and is reachable from y. Therefore, if VisitedX() returns $True$, then there is an outblock-free path from y to x.

The "only if" part We next show that if there is a outblock-free path P from $y \in S_x$ to x, then VisitedX($sidom(x), x$) returns $True$. Note that our algorithm outblocks all the nodes in $Pred_f(x)$ except one node, say node z. So if P is a outblock-free path, then P must contain z, and so it is obvious to see that VisitedX($sidom(x), x$) will return true.

Finally, we prove our main result in Theorem 5.1. In Lemma 5.5 we have established that **VisitedX()** returns $True$ iff there is an outblocked-free path from some node in S_x to x. In Algorithm 5.1, a node is added to the set M_x only if **VisitedX()** returns $True$, meaning only if there is an outblock-free path from some node in S_x to x in the DJ graph. We will show in Theorem 5.1 that the set M_x, when the algorithm terminates, indeed satisfies the Definition 5.2.

Theorem 5.1 *Algorithm 5.1 correctly computes* $midom(x)$ *for any* $x \in \mathcal{MIDOM}$.

Proof:

First of all it is straightforward to see that M_x is a subset of $Pred_f(x)$. This is because each node y that is added to M_x belongs to $Pred_f(x)$ (step 35, and also M_x is initialized with the empty set at step 31).

Next we will show that all paths from $sidom(x)$ to x (in the flowgraph) contain some node in M_x. The proof is again easy. A node is added to M_x only if **VisitedX()** returns $True$. From Lemma 5.5 it is clear that **VisitedX()** returns $True$ if x is reachable from some node in S_x. Therefore all paths from $sidom(x)$ to x (in the flowgraph) contain some node in M_x.

Next we will show that for each $y \in M_x$ there is at least one path from $sidom(x)$ (in the flowgraph) that contains y but does not contain any other nodes in $Pred_f(x)$. Again the proof follows from Lemma 5.5. What we do in Algorithm 5.1 is to outblock all the nodes in $Pred_f(x)$ except node y, and then check if there is a outblock-free from some node in S_x. From Lemma 5.5 we know that **VisitedX()** returns $True$ if such a path is found. Also this path should contain y (since all other nodes in $Pred_f(x)$ are outblocked). Hence the result.

∎

Next we analyze the worst case time complexity of computing the *midom*-sets for all the nodes in $\mathcal{MIDOM}$. We first give the time complexity of Algorithm 5.1, which computes $midom(x)$ for any $x \in \mathcal{MIDOM}$.

Theorem 5.2 *The worst-case time complexity of Algorithm 5.1 is $O(|E| \times p)$, where $|E|$ is the number of edges in the DJ graph, and p is bounded by the maximum number of predecessor nodes of a node in the corresponding flowgraph.*

Proof:

For each node y in $Pred_f(x)$ we essentially traverse the DJ graph below $sidom(x)$ looking for a path to y that does not pass through any other nodes in $Pred_f(x) - \{y\}$. Also, we put a node on the stack only once (step 76). Finally, we only visit the nodes and edges of the DJ subgraph rooted at nodes that was previously put on the stack, *at most once*. Also, we perform the above process for each node in $Pred_f(x)$. With p being the maximum number of predecessors a node can have, the result easily follows. ∎

Let $x1, x2, \ldots, xk$ be the set of nodes in $\mathcal{MIDOM}$. The time complexity of computing the *midom* for k nodes is then bounded by $O(|E_d| \times (p_{x1} + p_{x2} \ldots + p_{xk}))$. Since the size k of the set $\mathcal{MIDOM}$ can be $O(|N|)$, the time complexity of computing the *midom* for all is bounded by $O(|E_d| \times |E_f|) \Rightarrow O(|E_f|^2)$ (since $p_{x1} + p_{x2} \ldots + p_{xk} = |E_f|$, for $k = |N|$).

5.5 Discussion and Related Work

We have shown how to use the DJ graph to facilitate computing the immediate multiple-node dominator of a node. Compared to Gupta's algorithm, ours has a better worst-case time complexity for computing the same set. We are currently not aware of any other work on generalized dominators. Finally, we notice that generalized dominators are related to the vertex cut-set problem. But computing *midom*-sets is different from general cut-set problem in the following ways. In the vertex (node) cut-set problem we are given an undirected graph, and the problem is to find a subset (usually a minimal subset) of nodes whose removal (along with all the edges incident on these nodes) will split the graph into two disconnected subgraphs. One could use any of the network flow algorithms to compute what is termed as min-cut set that will split the graph into two disconnected components by assuming all edges have the flow capacity of one. But in the *midom*-set problem for a flowgraph, our objective is to find a minimum subset of nodes from

$Pred_f(x)$ whose removal (along with all the edges incident on these nodes) will separate $sidom(x)$ and x. This does not necessarily separate the whole graph into disconnected components.

Chapter 6

Identifying Irreducible Loops

FORTRAN is not a flower but a weed – it is hardy, occasionally blooms, and grows in every computer.

— A.J. Perlis

Loop identification is a necessary step in loop transformations for high-performance architectures. Some compilers detect loop structures using syntactic constructs (e.g., `for`, `while`, etc.), while others detect loops using flowgraphs. The latter approach is more general in that it can detect loops in programs that even use `goto` statements or that are represented in low-level intermediate languages. In this chapter we follow this approach. One classical technique for detecting loops is using Tarjan's interval algorithm [Tar74]. The *Tarjan intervals* are *single* entry, strongly connected subgraphs [Tar74]. However, Tarjan's interval finding algorithm does not directly handle flowgraphs containing loops with more than one entry, i.e., loops with *multiple entries*. Such loops will be called as *irreducible loops* in this dissertation, whereas loops with single entry will be called *reducible loops* [Hec77]. There are extensions to Tarjan's algorithms that are listed in Section 6.3.

In this chapter we give a simple algorithm for identifying both reducible and irreducible loops using DJ graphs. As we will show in this chapter, our method can be considered as a generalization of Tarjan's interval algorithm (since we can identify nested loop intervals even in presence of irreducibility). Furthermore, we use level information in the DJ graph to detect finer irreducible regions, thus confining the effect of irreducibility to small regions.

We begin the chapter by motivating the notion of reducible and irreducible loops. Then, in Section 6.2, we give a simple algorithm for identifying loops in a flowgraph. Finally, in Section 6.3, we compare our work with other related work.

6.1 Introduction and Motivation

In the literature there are two kinds of flowgraphs: *reducible* flowgraphs, and *irreducible* flowgraphs (see Figure 6.1). Hecht and Ullman gave the following definition for reducibility of graphs [HU74, ASU86].

Definition 6.1 *A graph 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 initial node of G.*
2. *The back edges consists only of edges whose destination nodes dominate their source nodes.*

The above definition of reducibility applies equally to both DJ graphs and flowgraphs. In other words, we can easily see that a DJ graph is reducible if and only if the corresponding flowgraph is also reducible.

In a reducible flowgraph the destination node h of a back edge $x \rightarrow h$ is called the *loop header* or *loop entry* node. Reducibility of flowgraphs are related to reducibility of loops. If L is a loop with L_h as the entry node of the loop, then L_h will dominate every node in the loop. One of the classical approach for identifying loops is based on Tarjan's interval algorithm.

Tarjan's intervals are single-entry, strongly connected subgraphs, and they closely reflect the loop structures in programs [Tar74]. The basic idea behind Tarjan's method is to repeatedly *collapse* each loop into a single node inside-out until the whole graph reduces to one node. This idea will work if the flowgraph is reducible. Recall that a node h is a loop header if $x \rightarrow h$ is a back edge and h dominates x. Now as the reduction process proceeds in Tarjan's method, if we come across a back edge $x \rightarrow g$ such that g does not dominate n, further reduction

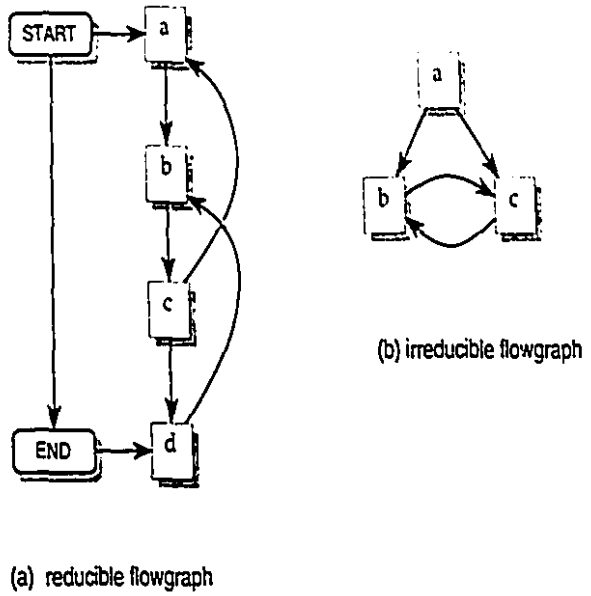


Figure 6.1: An example of a reducible flowgraph and an irreducible flowgraph

of the graph cannot be continued. This is because g is not a unique loop header node. A number of actions can be taken at this point; one is to split the header node, transforming the graph to a reducible graph; another action is to abandon the reduction process and warn the programmer that the graph is not reducible; yet another action would be to identify a single entry region which encloses the irreducible portion and collapse the single entry region as one node, and continue with the reduction.

In this chapter we take a different approach for reduction. Our approach uses DJ graphs. Translating the notion of reducibility on DJ graph, we can easily observe the following property.

Lemma 6.1 *A flowgraph is irreducible if and only if there exists a simple cycle in its DJ graph that does not contain a BJ edge (that is, the cycle is made of only D edges and CJ edges).*

Proof:

Follows from the definition of DJ graphs and Definiton 6.1. ∎

The interesting aspect of this lemma is that if we perform a depth-first search on such a DJ graph, we can always find a potential sp-back edge that is also a

CJ edge, and so every sp-back edge is not a BJ edge.[1] Using this key intuition, we can now perform depth-first search on the DJ graph, and identify all back edges, called sp-back edges. Once we identify the sp-back edges we can identify loops in a bottom-up fashion on the DJ graph. The complete algorithm is given in Algorithm 6.1.

Example 6.1

Consider the flowgraph shown in Figure 5.2 whose corresponding DJ graph shown in Figure 5.3 (Chapter 5). The flowgraph is irreducible because of a multiple-entry loop that has two entry nodes 3 and 5. The DJ graph contains a simple cycle $3 \rightarrow 6 \rightarrow 7 \rightarrow 12 \rightarrow 3$, in which there are no BJ edges.

Before presenting the complete algorithm we will introduce another key concept that is useful for understanding the algorithm. The following lemma states that all the "entry nodes" of an irreducible loop have the same immediate dominator.

Lemma 6.2 *All the entry nodes of an irreducible loop have the same immediate dominator.*

Proof:

Let $x_1, x_2, \ldots x_n$ be the set of loop entry nodes of an irreducible loop. By definition of loop, there exists a cycle C, such that $x_1, x_2, \ldots x_n$ are in C. Let $y = idom(x_i)$ and let $z = idom(x_j)$, for some $i \neq j \in \{1, 2, \ldots n\}$. We want to show $y = z$. Suppose that $y \neq z$. Then there is a path from START $\stackrel{*}{\rightarrow} z \stackrel{+}{\rightarrow} x_j \stackrel{+}{\rightarrow} x_i$ that does not pass through y, contradicting $y = idom(x_i)$. Therefore, y must be the same as z. ∎

What the above lemma implies is that when we are looking for candidate nodes that belong to a loop, look only at nodes that are at the same level as loop entry nodes and below it (i.e., whose level numbers are greater than the loop header's level). This is obvious for a reducible loop, since its unique entry node, the loop header, strictly dominates (or has a smaller level number than) any other nodes in the loop. For an irreducible loop all the entry nodes of the loop

[1] We say $x \rightarrow y$ is an sp-back edge iff y is x or is an ancestor of x in a depth-first spanning tree.

are at the same level (follows from Lemma 6.2). Therefore, we can identify an irreducible loop with entry nodes at a certain level by determining the Strongly Connected Component(SCC) while considering only nodes whose level number is equal to or greater than the current level. Once we find such a SCC, collapse the whole component into one node. It is also important to emphasize that using this technique we can identify reducible loops nested inside an irreducible loop.

6.2 Our Algorithm

The complete algorithm for identifying loops is given below.

Algorithm 6.1 *The following algorithm identifies both reducible and irreducible loops*

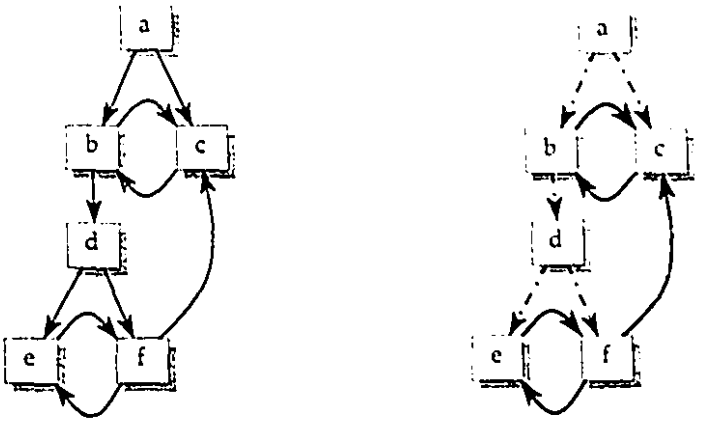
```
MainLoop()
{
85:   Perform a depth-first search on the DJ graph and identify sp-back edges;
86:   for(i = NumLevel − 1 downto 0) /* visit nodes in a bottom-up fashion */
87:     Irreducible = False;
88:     foreach node n with n.level = i do
89:       foreachedge m → n do
90:         if m → n is both a CJ edge and an sp-back edge then
91:           Irreducible = True; /* n is in an irreducible loop */
92:         endif
93:         if n is a destination node of a BJ edge then
94:           Find ReachUnder(n) for all the BJ edges m1 → n,...,mk → n
95:           Collapse the loop consisting of nodes {n} ∪ ReachUnder(n);
96:         endif
97:       endfor
98:     endfor
99:     if(Irreducible) /* there exists an irreducible loop */
100:      Identify SCCs for the subgraphs induced by nodes at level i and below;
101:      Collapse each non-trivial SCC to a single node.
102:    endif
103:  endfor
}
```

The algorithm visits all the nodes level by level in a bottom-up fashion. At every level, when there is a reducible loop whose header is at this level, we identify it by checking to see if that node is the destination node of any sp-back edge. If yes, we know a potential reducible loop is found. The algorithm goes to find the loop body with that node being the loop header, and collapses the loop – between step 93 and step 96.[2] The procedure $ReachUnder(n)$ will find all the nodes that can reach the source nodes of sp-back edges incident on the loop header n, without going through n. We collapse the nodes in $\{n\} \cup ReachUnder(n)$ using Tarjan's set-union data structure, and these nodes form the body of the loop.

Note that all the sp-back edges with the same destination node are sp-back edges for the same loop. The algorithm also checks to see if that node could be an entry node of an irreducible loop. If yes, turns on the *Irreducible* flag at step 91—indicating there are some irreducible loops that need to be handled later from step 99 to step 102. Once we are done with the above process for every node at the level, we check to see if there is any irreducible loop with its entry nodes at this level at step 99. Notice that the flag *Irreducible* is set at step 91 to *true* if irreducibility is detected. If yes, we use SCC's to identify the loop body for an irreducible loop and collapse it. Also note that at this point any reducible loop whose header is at this level or below, has been "collapsed", so has any irreducible loop whose entry nodes are below this level.

For a loop nest consisting of L_1, L_2, L_3 and L_4, where L_1 and L_3 are reducible while L_2 and L_4 are irreducible, we can identify the loop body for all four loops. However, when L_1 and L_4 are reducible while L_2 and L_3 are irreducible, we will only identify three loops by merging L_2 and L_3 into one larger irreducible loop. That is, *immediately nested* irreducible loops will not be distinguished. The advantage of our method is to be able to identify the bodies of nested (reducible and irreducible) loops, with the restriction that a sequence of consecutively nested irreducible loops will be collapsed in a single SCC region. It can expose maximally, in a loop nest, the nesting structure of each portion which is reducible separated by irreducible regions.

[2]By "collapse" we mean that a loop body is condensed and becomes a single node. Any edge incoming into the loop from outside the loop will become an edge incoming into the representative node. Any edge outgoing from the loop to outside the loop will become an edge outgoing from the representative node.



(a) flowgraph (b) DJ Graph

Figure 6.2: An irreducible flowgraph with two irreducible loops

Also, the level information allows us to detect finer irreducible regions. This is illustrated in Figure 6.2. For example, using our algorithm we can detect two irreducible loops instead of one for the flowgraph shown in Figure 6.2. The two loops are: $L1 = \{e, f\}$, and $L2 = \{b, c, d, L1\}$.

Next we will prove the correctness of our algorithm.

Lemma 6.3 Algorithm 6.1 *correctly identifies both reducible and irreducible loops in a program.*

Proof:

The proof is based on induction on levels of nodes in the DJ graph.

Base case: The level number is the maximum. It is trivial if sp-back edges are also BJ edges. But if the sp-back edge is also a CJ edge then *Irreducible* as true, and the nodes in a SCC will be collapsed at step 101.

Induction: Assume that the assertion is true for level $k + 1$. We will then show that the lemma is true for level k. At level k when we detect irreducibility (i.e., sp-back edge is the same as CJ edge), we mark *Irreducible* is *True*. The procedure for *ReachUnder*(n) will find all nodes that can reach the source node of BJ edges incident on node

n, without going through node n. At step 95 we will collapse all such nodes. If *Irreducible* is true then Tarjan's SCC algorithm will collapse nodes in the SCC. In both cases, bodies of loops whose headers are at level k are collapsed. Hence the result. ∎

Finally we analyze the complexity of the algorithm. When a flowgraph is reducible, our algorithm has the same time complexity $O(|E| \times \alpha(|E|, |N|))$ as Tarjan's approach [Tar81]. The complexity of Tarjan's algorithm is dominated by the time taken to collapse the nodes of a loop. Tarjan uses balanced path compression to maintain the set of nodes in a collapsed loop, and this takes $O(|E| \times \alpha(|E|, |N|))$, where $\alpha()$ is the inverse Ackermann function.

For irreducible flowgraphs, the worst-case time complexity of the algorithm occurs when it needs to find irreducible loops at every level. Assume k is the number of levels in the DJ graph. Then the time complexity of the algorithm is $O(|E| \times \alpha(|E|, |N|) + k \times |E|)$, since finding strongly connected components needs $O(|E|)$ time. We anticipate in practice, k is a constant, so the algorithm is almost linear.

6.3 Discussion and Related Work

Identifying loops has been a classical exercise in control flow analysis. We have shown how to use DJ graphs for identifying both reducible and irreducible loops by extending Tarjan's approach [Tar74]. One feature of our algorithm for identifying irreducible loops is that it utilizes level information in DJ graphs to discover the body of irreducible loops. In Chapter 10 we propose a new elimination-based data flow analysis that uses DJ graphs for reduction and variable elimination. Generally, elimination-based methods are applicable only to reducible flowgraphs. In Chapter 10 we have used some of the key results presented in this chapter for handling irreducible flowgraphs during the reduction and elimination process.

Many methods have been proposed in the context of elimination based data flow analysis for handling irreducibility such as node splitting [Hec77], identifying single-entry region that encloses the irreducible region [Bur90, SS79], etc. In a technical report, Steensgaard proposed a method for identifying nested loops, both reducible and irreducible [Ste93]. His method consists of first applying Tarjan's SCC algorithm to the whole graph and identifying each non-trivial

SCC [Ste93]. He then identifies, for each non-trivial component, what he calls as *generalized entry nodes*. A node y in a SCC S is a generalized entry node of S if, in the original flowgraph, there is an edge from $x \rightarrow y$ such that $x \notin S$. Now if $z \rightarrow y$ is an edge in the original flowgraph such that $z \in S$, then the calls $z \rightarrow y$ as a *generalized back edge*. Once he identifies generalized back edges in an SCC, he eliminates them from the SCC and applies Tarjan's SCC algorithm once again on the SCC and identifies "inner" SCCs. This way he identifies loops in an outside-in fashion of loop nests.

In our method, we apply Tarjan's SCC algorithm in an inside-out fashion of loop nests—our bottom-up reduction order will conform to this inside-out order of loop nests. Also, we will apply Tarjan's SCC algorithm only if we detect that there is an irreducible loop at a particular level. Therefore, the time complexity of our approach is expected to be better than Steensgaard's approach. The worst-case time of Steensgaard's algorithm can be quadratic in terms of the loop nesting.

Chapter 7

Computing Iterated Dominance Frontiers in Linear Time

I don't know the key to success, but the key to failure is to please everybody.
—Bill Cosby

I'd rather be a failure at something I like than a success at something I hate.
— George Burns

The first requisite for success is the ability to apply your physical and mental energies to one problem incessantly without growing weary.
—Thomas Edison

In this chapter we present a simple *linear* time algorithm for computing the Iterated Dominance Frontier (IDF) for a set of nodes using DJ graphs. A novel aspect of our algorithm is that it can be also used in conjunction with $\mathcal{APT}$ for computing iterated dominance frontiers [PB95]. Recall, from Chapter 4, that $\mathcal{APT}$ is a spectrum of dominance frontier representations, in which the DJ graph is at one end with no caching, and the full dominance relation is at the other end with full caching. In this chapter we will illustrate our algorithm on DJ graphs [SG95b].

Iterated dominance frontiers have many applications, such as for placing ϕ-nodes for arbitrary Sparse Evaluation Graphs (SEGs) and Static Single Assignment SSA form [CFR+91, CCF91], computing guards [Wei92], incremental computation of dominator trees (Chapter 8), and incremental data flow analysis (Chapter 11).

We begin the chapter by introducing and motivating the problem of computing IDF. Then, in Section 7.2, we give the complete algorithm and in Section 7.3 we give an example illustrating the algorithm. In Section 7.4, we prove its correctness and analyze its time complexity. In Section 7.5 we give our experimental results. Finally, in Section 7.6, we discuss the related work, and give our conclusion.

7.1 Introduction and Motivation

The Static Single Assignment (SSA) form [CFR+89, CFR+91] and the related Sparse Evaluation Graphs (SEGs) [CCF91], have been successfully used for efficient data flow analyses and program transformations [CLZ86, RWZ88, AWZ88, WZ85, WCES94, Bri92, CBC93]. The algorithms for constructing these two intermediate representations have one common intermediate step—computing program points where data flow information are potentially "merged", the so called ϕ-nodes [CFR+91, CCF91]. Given a flowgraph, the original algorithm for computing ϕ-nodes for an SEG consists of the following steps [CFR+91, CCF91]:

1. Precompute the dominance frontier $DF(x)$ for each node x (Chapter 4).

2. Determine the initial set of 'sparse' nodes N_α that represent non-identity transference in a data flow framework. For SSA, such nodes contain definitions of variables [CFR+91].

3. Compute the iterated dominance frontier $IDF(N_\alpha)$ for the initial set N_α. Cytron et al. have shown that the desired set of ϕ-nodes for an SEG is same as the iterated dominance frontier $IDF(N_\alpha)$ of the initial set [CFR+91].

The most time consuming step in the above algorithm is computing $IDF(N_\alpha)$, the iterated dominance frontier of the initial set of sparse nodes N_α. The time complexity of computing $IDF(N_\alpha)$ depends on the size of the dominance frontier relation. Although the size of the dominance frontier is linear for many programs (as was noted by Cytron et al.), there are cases in which the size of dominance frontiers is quadratic in terms of the number of nodes in a flowgraph, for example, nested `repeat-until` loops [CFR+91]. Note that, even though the size of the dominance frontier may be quadratic in terms of the number of nodes in the

flowgraph, the number of ϕ-nodes that is needed remains linear (for a particular SEG) [CFR+91]. As Cytron and Ferrante pointed out: "Since one reason for introducing ϕ-nodes is to eliminate potentially quadratic behavior when solving actual data flow problems, such worst case behavior during SEG or SSA construction could be problematic. Clearly, avoiding such behavior necessitates placing ϕ-nodes without computing or using dominance frontiers" [CF93].

To overcome the potential quadratic behavior of computing ϕ-nodes using dominance frontiers, Cytron and Ferrante proposed a new algorithm that has a better complexity than the original algorithm [CF93]. Instead of first precomputing the full dominance frontier relation and then using this relation for computing ϕ-nodes, Cytron and Ferrante use Tarjan's *balanced path compression* algorithm [Tar79], and combined with other properties that relate dominance relation and depth-first numbering of the flowgraph, gave an algorithm that has a time complexity of $O(E \times \alpha(E))$, where $\alpha()$ is the slowly growing inverse Ackermann function.

In this chapter, we present a *simple linear* time algorithm for computing the ϕ-nodes for a set of nodes without precomputing dominance frontiers for all the nodes. Given a set of initial nodes N_a, to compute the relevant set of ϕ-nodes, we made one key observation: Consider any two nodes x and y, where y is an ancestor of x in the dominator tree. If the dominance frontier of x, $DF(x)$, has been computed, then to compute the dominance frontier of y, $DF(y)$, we need not recompute $DF(x)$ (see Chapter 4). However, the reverse may not be true. Therefore, we order the nodes in the dominator tree in such a way that when the computation of $DF(y)$ is performed, the dominance frontier $DF(x)$ of any descendant node x, if it is essential for computing the desired set of ϕ-nodes for N_a, has already been computed and is so marked. As a result for any such x, the computation of $DF(y)$ do not require the traversal of the dominator sub-tree rooted at x. Recall that we used a similar trick for computing dominance frontiers of a set of nodes in linear time (see Chapter 4, Section 4.2). The algorithm presented here is an extension of that algorithm.

To perform the proper node ordering and marking, our algorithm uses DJ graphs. The levels of the nodes in the dominator tree are used to order the computation of dominance frontiers of those nodes, x, which are essential to compute the final set of ϕ-nodes, in a bottom-up fashion. Meanwhile, during

each computation of $DF(x)$, the descendant nodes of x in the dominator sub-tree rooted at x are visited in a top-down fashion guided by the D edges, while avoiding nodes which have already been marked. During this top-down visit, J edges are used to identify the candidate nodes that should be added into the final set of ϕ-nodes, and recursively explored further. It is important to observe (yet another key observation!) that each new candidate node that is generated on-the-fly always has a level number no greater than that of the node currently being processed (assuming that the nodes in the dominator tree are numbered such that all nodes have a level number equal to the depth of the node from the root of the tree). Therefore, a data structure, called the *OrderedBuckets* (Section 7.2), is used to keep the candidate nodes in the order of their respective levels, and no nodes are inserted into the *OrderedBuckets* more than once. We show that our algorithm visits each edge in the DJ graph at most once, and therefore the complexity is *linear*.[1]

7.2 Our Algorithm

In this section, we present our algorithm for computing iterated dominance frontiers. Let N_α be the initial set of sparse nodes, and let IDF be the desired of IDF for N_α. Recall that one way of computing IDF is to first precompute the dominance frontiers for all nodes and then use the inductive definition of IDF (Equation (2.3)) to compute the IDF for the set of nodes N_α. Cytron et al. have shown that this can lead to a quadratic time complexity [CFR+91]. Rather than precomputing the dominance frontiers for all nodes, our linear time algorithm is based on two key observations:

1. Let y be an ancestor node of a node x on the dominator tree. If $DF(x)$ has already been computed before the computation of $DF(y)$, $DF(x)$ need not be recomputed when computing $DF(y)$. However, the reverse may not be true; therefore the order of the computation is crucial.

2. When computing $DF(x)$ we only need to examine J edges $y \rightarrow z$, where y is a node in the dominator sub-tree rooted at x and z is a node whose level

[1] Recall from Theorem 3.1, the number of edges in the DJ graph is no more than $|N_f| + |E_f|$, where $|N_f|$ is the number of flowgraph nodes, and $|E_f|$ is the number of flowgraph edges.

is no greater than the level of x. Recall that we have previously made this observation in Lemma 4.1.

We use a data structure called the *OrderedBuckets* to keep the candidate nodes in the order of their respective levels.[2] Based on the above observations, levels of the nodes in the dominator tree will be used in a bottom-up fashion to order the computation of dominator frontiers of those nodes, x, which are essential to compute the final set IDF. Meanwhile, during each computation of $DF(x)$, the nodes in the $SubTree(x)$ are visited in a top-down fashion guided by D edges, while avoiding nodes which have already been marked. During this top-down visit, J edges are used to identify the candidate nodes which should be added to the set IDF, and those to be recursively explored further. Note that each new candidate generated on-the-fly always has a level number no greater than that of the node currently being processed, and we ensure that no nodes are inserted into the *OrderedBuckets* more than once. Therefore, intuitively, we visit each edge in the DJ graph at most once. This, and the structure of the *OrderedBuckets*, are the basis of the time linearity of our algorithm.

The *OrderedBuckets* is an array of list of nodes, with index (or bucket) i storing nodes of level i (See Figure 7.2). Associated with the *OrderedBuckets* are two procedures: **InsertNode()** and **GetNode()**. **InsertNode()** inserts a node in the *OrderedBuckets* at the index corresponding to the level number of the node. **GetNode()** returns a node whose level number is the maximum of all nodes currently stored in the *OrderedBuckets*. We first insert the initial set of nodes N_α into the *OrderedBuckets*. Then, we iteratively compute the dominance frontier of the nodes in the *OrderedBuckets* in the order that **GetNode()** returns them to obtain the iterated dominance frontier of the initial set of nodes N_α. It is important to note that a node is inserted into the *OrderedBuckets* if it is either in N_α or is in the iterated dominance frontier of some node in N_α.

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

- *NumLevel* is the total number of levels in the dominator tree embedded in the DJ graph.

[2]Previously we used the term *PiggyBank* for *OrderedBuckets* [SG95b]. *OrderedBuckets* can be considered as a connotation for an *indexed set of buckets*, with indices corresponding to levels in the DJ graph.

- Each node $x \in N$ has the following attributes:

  $struct\ NodeStructure\{$
  $visited = \{Visited, NotVisited\}$
  $alpha = \{Alpha, NotAlpha\}$ /* in N_α or not */
  $inidf = \{InIDF, NotInIDF\}$ /* in IDF or not */
  $level = \{0 \ldots NumLevel - 1\}$ /* levels of nodes */
  $\}$

- Each edge $x \rightarrow y \in E$ has an attribute that specifies the type of the edge: $\{Dedge, Jedge\}$.

- *OrderedBuckets* is an array of list of nodes. Its structure is defined as follows:

  $struct\ OrderedBucketsStructure\{$
  $NodeStructure\ *node$
  $OrderedBucketsStructure\ *next$
  /* list of nodes at the same level */
  $\}\ *OrderedBuckets[NumLevel]$

- *CurrentLevel* is initially $NumLevel - 1$, and subsequently has a value corresponding to the level number of the node that GetNode() returns.

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

The first step in the algorithm is to insert all the nodes in N_α into the *OrderedBuckets* (steps 104 to 107). We mark the nodes that are initially inserted into the *OrderedBuckets* as *Alpha* to indicate that they belong to the initial set N_α. This is needed to avoid re-inserting them into the *OrderedBuckets* again in the future (a condition that we check in the procedure Visit(), at step 119). We then iteratively invoke the procedure Visit() on the nodes that GetNode() returns to compute the iterated dominance frontier set IDF. At step 109, we assign the variable *CurrentRoot* to point to the node x that GetNode() returns in order to keep track of the current root of $SubTree(x)$. Before Visit(x) is invoked at step 111, the node x is marked *Visited* at step 110. This marking is crucial because

we never visit a node that has been marked *Visited*. We check for this condition in the procedure **Visit()** at step 125.

Algorithm 7.1 *The following algorithm computes* $IDF(N_\alpha)$.

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

♠ **Output:** The set $IDF = IDF(N_\alpha)$.

♠ **Initialization:**

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

♠ **The Algorithm:**

```
Main()
{
104:  foreach x ∈ N_α do
105:    x.alpha = Alpha
106:    InsertNode(x) /* Insert the nodes in the OrderedBuckets */
107:  endfor
108:  while((x = GetNode()) != NULL)
109:    CurrentRoot = x
110:    x.visited = Visited
111:    Visit(x) /* Find the dominance frontier of x */
112:  endwhile
}
```

The procedure **Visit()** called with *CurrentRoot* essentially traverses the *SubTree*(*CurrentRoot*) in a top-down fashion marking the nodes in the sub-tree as *Visited* if the nodes are not already marked *Visited* (a condition checked at step 125). Notice that the nodes in the dominator sub-tree are connected through D edges. As it walks down the sub-tree, the procedure **Visit()** also "peeks" at the destination node of J edges, without marking it as *Visited*. Whenever it notices that the level number of a node (that it peeked through a J edge) is less than or

equal to the level number of $CurrentRoot$, it adds the node into the set IDF, if the node is not already in the set (a condition checked at step 116). It also marks the node as $InIDF$ whenever the node is added to the set IDF. This marking is necessary to avoid adding the node again into IDF whenever it may peek at this node through some other J edge in the future. It also inserts the node into the $OrderedBuckets$ if the node is not in the set N_o (a condition checked at step 119).

```
Procedure Visit(x)
{
113:  foreach y ∈ Succ(x)
114:    if(x → y == Jedge)
115:      if(y.level ≤ CurrentRoot.level)
116:        if(y.inidf != InIDF) /* Check if y already in IDF */
117:          y.inidf = InIDF /* y in IDF */
118:          IDF = IDF ∪ {y} /* Compute the set IDF */
                  /* Check if x is already OrderedBuckets */
119:          if(y.alpha != Alpha)
                  /* Put it in OrderedBuckets for future search */
120:            InsertNode(y)
121:          endif
122:        endif
123:      endif
124:    else /* x → y is Dedge */
125:      if(y.visited != Visited) /* Avoid redundant visit */
126:        y.visited = Visited
127:        Visit(y)
128:      endif
129:    endif
130:  endfor
}
```

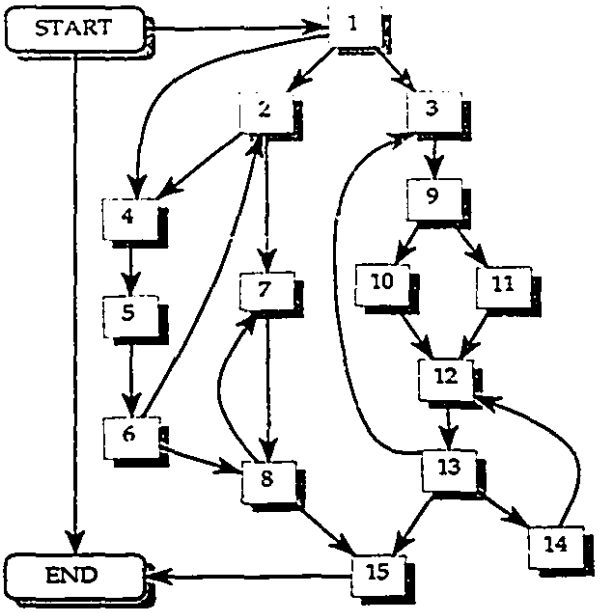
GetNode() returns a node whose level number is the maximum of all the nodes currently in the *OrderedBuckets*. GetNode() also removes this node from the *OrderedBuckets*, and adjusts the *CurrentLevel* accordingly. *CurrentLevel* keeps track of the level number of the node that GetNode() returns. Note that a node will never be inserted in *OrderedBuckets* at a level number greater than

CurrentLevel. As a result, *CurrentLevel* monotonically decreases through the level numbers. That is, the calls to **Visit**(x) at step [111] is performed in a bottom-up fashion, in contrast, with each such call, the traversal of the dominator sub-tree rooted at x is performed in a top-down fashion. The marking of the nodes prevents any nodes from being processed more than once in the algorithm. This is essential to ensure the time linearity of the algorithm.

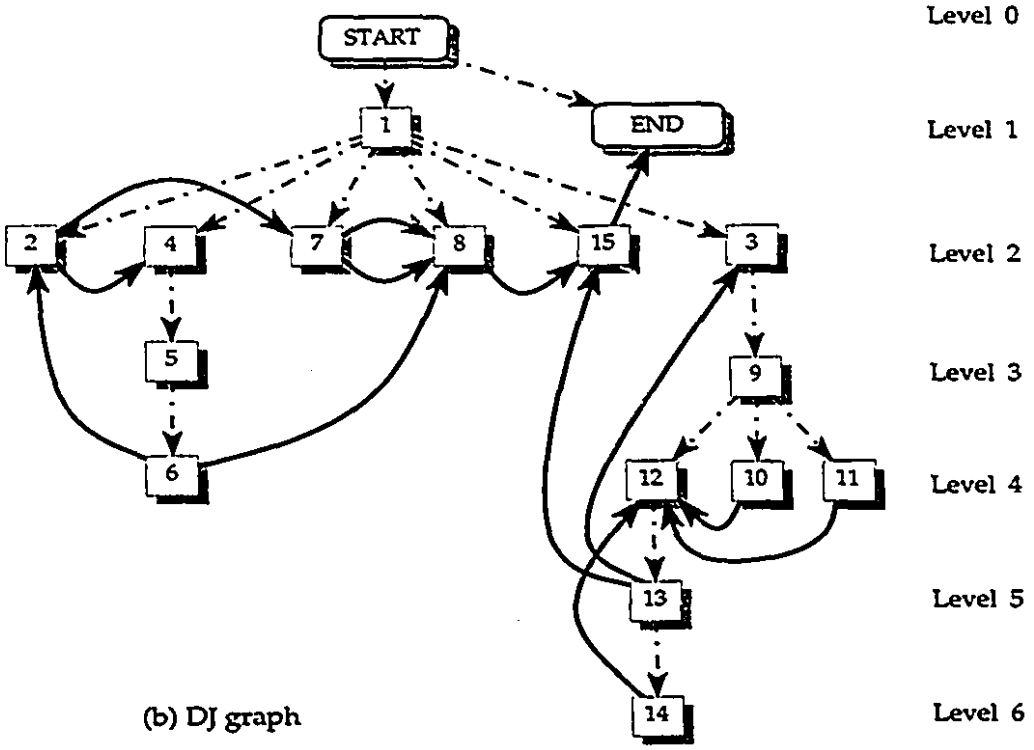
```
Procedure InsertNode(x)
{
131:   x.next = OrderedBuckets[x.level]
132:   OrderedBuckets[x.level] = x
}
Function GetNode()
{
133:   while(CurrentLevel > 0)
134:     if(OrderedBuckets[CurrentLevel] == NULL)
135:       CurrentLevel = CurrentLevel - 1
136:     else
137:       x = OrderedBuckets[CurrentLevel]
                 /* Delete x from OrderedBuckets */
138:       OrderedBuckets[CurrentLevel] = x.next
139:       return x.node
140:     endif
141:   endwhile
142:   return NULL
}
```

7.3 An Example

Next we illustrate Algorithm 7.1 through an example. Consider the flowgraph and its DJ graph shown in Figure 7.1. Let $N_a = \{5, 13\}$. The first step is to deposit the nodes 5 and 13 into the *OrderedBuckets*, and also mark them as *Alpha*. After the for loop at step [104], the *OrderedBuckets* would look like Figure 7.2(a). At step [108], the function **GetNode()** returns node 13. **GetNode()** also removes 13



(a) flowgraph



(b) DJ graph

Figure 7.1: Another example of a flowgraph and its DJ graph.

from the *OrderedBuckets*. At step [109], *CurrentRoot* is set to node 13. To find the dominance frontier of node 13 we call Visit(13) at step [111]. Prior to this, we also mark node 13 as *Visited* at step [110].

In the procedure Visit(), at step [113] we find that the successor nodes of 13 to be nodes 3, 15, and 14. Of these, 13 → 15 and 13 → 3 are J edges, and 13 → 14 is a D edge. Since 15.*level* = 2 and 3.*level* = 2 are less than *CurrentRoot.level* = 13.*level* = 5, nodes 3 and 15 are added to IDF (since they are not already in IDF). Also, neither 3 nor 15 is marked *Alpha* (and hence not in N_α), both the nodes are inserted into the *OrderedBuckets* (step [120]). Figure 7.2(b) shows the new state of the *OrderedBuckets*.

Next, since the edge 13 → 14 is a D edge, and node 14 is not yet visited, we call Visit(14) at step [127]. Again, before calling Visit(14), we mark node 14 as *Visited* (step [126]). The only successor of 14 is node 12, and 12.*level* = 4 is less than *CurrentRoot.level* = 13.*level* = 5. Also, node 12 is neither in IDF nor in N_α, and so is added to IDF and inserted into the *OrderedBuckets* (step [118] and [120], respectively). The call to Visit(13) terminates and returns at step [111].

Now the function GetNode() is executed at step [108] and it returns node 12. Visit(12) is called at step [111], and *CurrentRoot* is set to node 12. The only successor of 12 is node 13, and 12 → 13 is a D edge. Since node 13 is already marked *Visited*, the call to Visit(12) terminates and returns at step [111].

GetNode() is called again, and this time it returns node 5. Visit(5) is called at step [111] and the process continues. Figure 7.2 shows the complete trace of the *OrderedBuckets* for the example.

7.4 Correctness and Complexity

In this section, we first give a proof of correctness (Theorem 7.1), and then analyze the complexity of the algorithm (Theorem 7.2).

7.4.1 Correctness

The main theorem which establishes the correctness of Algorithm 7.1 is Theorem 7.1. The theorem states that the algorithm computes the iterated dominance

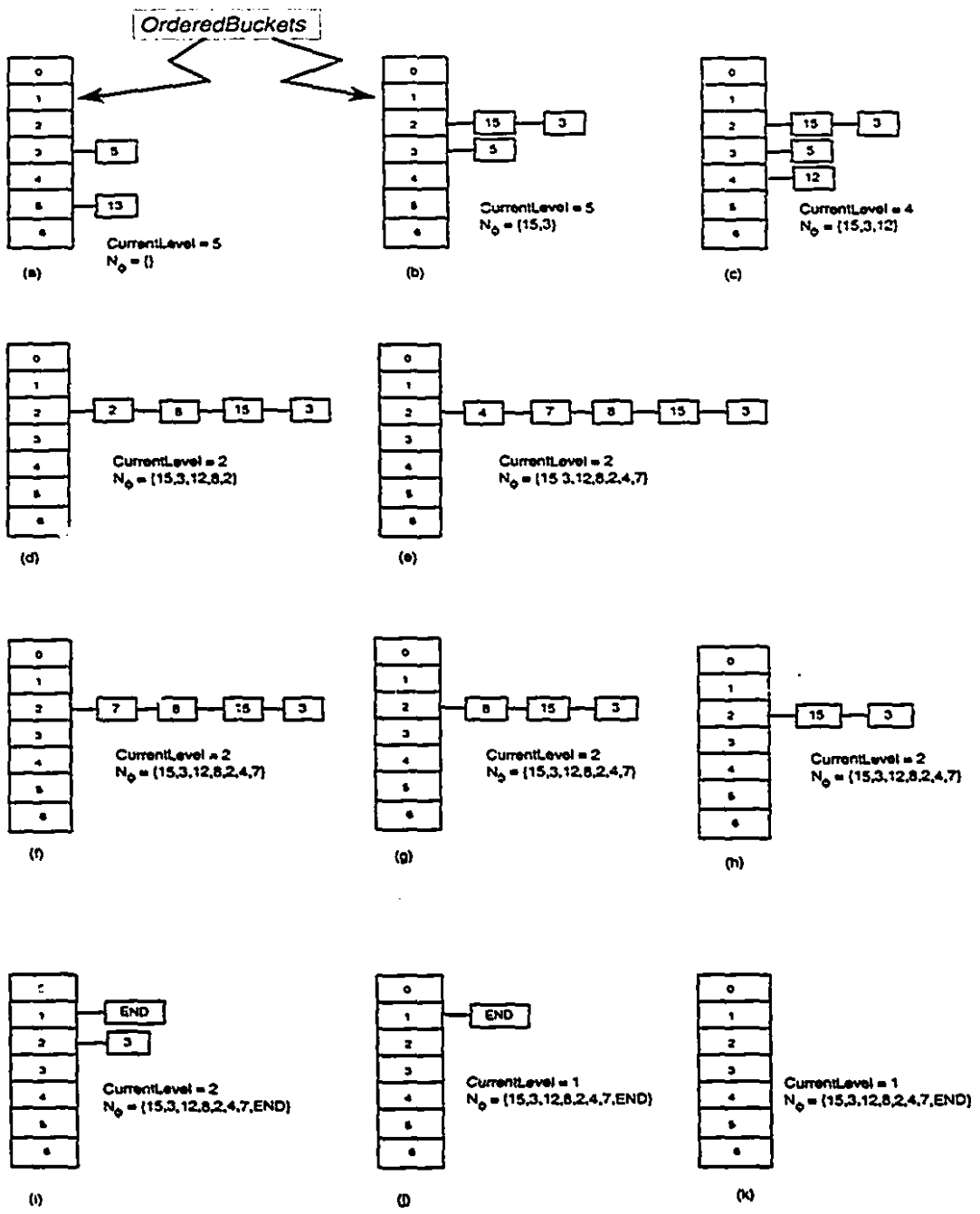


Figure 7.2: A trace of the iterated dominance frontier algorithm.

frontier of the set N_a. The inductive proof of the theorem is based on a major lemma, Lemma 7.4, which establishes the fact that when the algorithm calls Visit(x) at step [111] and the call terminates, all nodes in the dominance frontiers $DF(x)$ are already added into the set IDF (a fact used both in the induction basis and induction steps). Let x be the current root of the dominator sub-tree visited by Visit(x) at step [111]. Let z be in $DF(x)$. Lemma 4.1, introduced earlier in Chapter 4, guarantees that there must exist a node y in $SubTree(x)$ such that $y \rightarrow z$ is a J edge and $level.z \leq level.x$. Another lemma, Lemma 7.3, states that y will already have been marked *Visited* when Visit(x) returns. There are two cases in the algorithm where a node can be marked *Visited*: 1. at step [126], and 2. at step [110]. The validity of Lemma 7.4 for case 1 is straightforward. For case 2, y must be marked *Visited* by an earlier call of Visit(v) for some node v in $SubTree(x)$. This fact is made possible because of the $OrderedBuckets$ structure and we formalize this in Lemma 7.1 and Lemma 7.2. We then make an inductive argument on the decreasing level of the nodes to demonstrate that all nodes in $DF(v)$ should already be inserted into IDF by this time. The node z should also be in IDF according to Lemma 4.1. From this the validity of Theorem 7.1 is established.

In our chain of proofs, we begin with Lemma 7.1.

Lemma 7.1 *A node is never inserted in the OrderedBuckets at an index that is greater than CurrentLevel.*

Proof:

There are only two places (in the algorithm) that a node can be inserted in the *OrderedBuckets*: at step [106] and at step [120]. Since the initial value of *CurrentLevel* is $NumLevel - 1$, the level number of any node that is insert at step [106] can never be greater than $NumLevel - 1$.

At step [120], a node y is inserted in the *OrderedBuckets* only if it is visited through a J edge and if $y.level \leq CurrentLevel$. Therefore y is never inserted in the *OrderedBuckets* at an index that is greater than *CurrentLevel*. ∎

Lemma 7.2 gives an order (based on the level number of nodes) in which calls to Visit(), at step [111], can be performed. The ordering of nodes is controlled

by calls to **GetNode()** at step [108]. Recall that **GetNode()** always returns a node whose level number is the maximum of all nodes currently stored in the *OrderedBuckets* structure.

Lemma 7.2 *Let x and y be any two nodes that are inserted in the OrderedBuckets and later removed (and returned) from the OrderedBuckets by* **GetNode()** *at step* [108]. *If $y.level > x.level$, then* **Visit**(y) *will be called earlier than* **Visit**(x) *at step* [111].

Proof:

First of all observe that **Visit()**, at step [111], is always called on the node that **GetNode()** returns at step [108]. Also, we know that **GetNode()** always returns a node whose level number is the maximum of all the nodes currently in the *OrderedBuckets*. Also, we know from Lemma 7.1 that a node will never be inserted in *OrderedBuckets* at an index greater than *CurrentLevel*. From this we prove the validity of the lemma as follows: There are two cases:

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

Case 2 Before **GetNode()** returns either of the two nodes, only one of the two nodes is in the *OrderedBuckets*. Let x be in the *OrderedBuckets*. This means that either y was already inserted and removed from the *OrderedBuckets*, even before x was inserted into the *OrderedBuckets*, in which case the validity of the lemma is true, or y will be inserted in future. The latter situation is impossible since, from Lemma 7.1, y should be inserted into the *OrderedBuckets* at a level number that is less than or equal to the level number of x. (Recall that we have assumed $y.level > x.level$). We can make a similar argument by assuming that y is in the *OrderedBuckets*.

■

The next lemma establishes an important fact that when a node x is visited by a call of **Visit**(x) from step [111] and returned, that all nodes in $SubTree(x)$ have

been marked *Visited*. Intuitively, this means that when such a visit returns, none of the nodes in the $SubTree(x)$ have been overlooked.

Lemma 7.3 *When* Visit(x) *returns at step* 111, *all nodes in* $SubTree(x)$ *are marked* *Visited*.

Proof:

Using Lemma 7.2 and induction on the levels of nodes, we can easily prove the lemma. The base case is when the node x has the maximum level; the validity of the lemma is straightforward. Assume that the lemma is true for all Visit(x) returned at step 111 with $x.level \geq k$ for some k. Now, assume we examine Visit(x) with $x.level = k - 1$. From our observation above, a top down traversal of nodes in $SubTree(x)$ will be performed during the execution of Visit(x). Assume y is the next node to be probed at step 125. One of the following two cases will be encountered:

Case 1: The next node y is not marked *Visited*. Then the program will continue to mark it *Visited* via a recursive call to Visit() at step 127.

Case 2: y is already marked *Visited*. y could only have been marked *Visited* by some earlier call to Visit(y), and this call must have been invoked at step 111, and not at step 127 (since the nodes are visited in a top-down fashion and there can be only one D edge). But since $y.level \geq x.level$ (i.e. $y.level \geq k$), by induction on k, we know Visit(y) has marked y and all descendants of y. (This is because Visit(y) was called at step 111 prior to Visit(x)).

■

It is easy to see from Lemma 7.2 and Lemma 7.3, that calls to Visit() at step 111 are made in a bottom-up fashion and while each recursive call at step 127, the recursive procedure Visit() visits the nodes in the dominator tree in a top-down fashion.

Lemma 7.4 is the main lemma which shows how the procedure Visit() captures the dominance frontier of a node in the set IDF. We will use this lemma in the main theorem (Theorem 7.1) to inductively argue the correctness of Algorithm 7.1.

Lemma 7.4 *When* **Visit**(x) *is called with* x *as the* $CurrentRoot$ *and returned at step* 111, *all the nodes in* $DF(x)$ *are also in the set* IDF.

Proof:

Let x be the current root of the $SubTree(x)$ visited by a call to **Visit**(x) at step 111, and terminated. Also let z be in $DF(x)$. From Lemma 4.1 there must exist a node y in $SubTree(x)$ such that $y \rightarrow z$ is a J edge and $z.level \leq x.level$. Since y is in $SubTree(x)$, from Lemma 7.3, y is marked *Visited*. As in the proof of Lemma 7.3 , there are two cases:

Case 1: y is marked *Visited* by the current **Visit**(x) invoked at step 111. Then, a recursive call at step 127 will cause its children (in the dominator tree) to be explored subsequently at step 113. Since $y \rightarrow z$ is a J edge, z will be included in IDF at step 118.

Case 2: y is not marked *Visited* by the current **Visit**(x) invoked at step 111. Since nodes in $SubTree(x)$ are visited in a top-down fashion, there must be a node u such that x stdom u and u dom y, and u is not marked *Visited* by the current call to **Visit**(x) (invoked at step 111). That is, u is marked *Visited* by a prior call of **Visit**(u) also invoked at step 111. If $u = y$, z will be added to the set IDF, since $y \rightarrow z$ is a J edge and $z.level < u.level$. If $u \neq y$, then y must be visited at step 125 via a D edge. A subsequent call of **Visit**(y) will add z to the set IDF, since $y \rightarrow z$ is a J edge and $z.level < u.level$.

∎

Notice that the above lemma only says that **Visit**(x), when it returns at step 111, will have added the entire dominance frontier of x to IDF. It does not specify which of the nodes in the set IDF belong to $DF(x)$. Notice that the set IDF can contain nodes that are not in the set $DF(x)$. In the other words, **Visit**() does not *explicitly* compute the dominance frontier of a node.

There is an important subtle point in the proof of Case 2 of Lemma 7.4. Note that we have argued that "u is marked *Visited* by a call of **Visit**(u) at step 111",

the reader may wonder how we can be sure such a u does exist. Is it possible for all the nodes from y up to x in the $SubTree(x)$ to have been marked $Visited$ by some previous call of Visit() via a D edge at step 127? The answer is no! This is because we visit nodes in the $Subtree(x)$ in a top-down fashion and pass through a node by an explicit check at step 125 to see if it is not yet marked *Visited*. This fact is important, as u is now assured to have been called earlier from step 111 afresh from *OrderedBuckets*. With u at the root of such an earlier call **Visit**(u), all the nodes in $DF(u)$ must have been examined and put into the set IDF. And our OrderedBuckets ensures **Visit**(u) happens before **Visit**(x) (Lemma 7.1). Otherwise, the algorithm may fail– we will come back to this issue again in Section 7.4.3.

Finally, we prove the main theorem.

Theorem 7.1 Algorithm 7.1 *correctly computes* $IDF(N_\alpha)$.

Proof:

We will show that, when Algorithm 7.1 terminates, the set IDF is same as $IDF(N_\alpha)$. From now on let $S = N_\alpha$. First of all it is obvious that $DF(S) \subseteq IDF$ (Lemma 7.4). Now we need to show that if $IDF_i(S) \subseteq IDF$ then $IDF_{i+1}(S) \subseteq IDF$, where

$$IDF_{i+1}(S) = DF(S \cup IDF_i(S))$$

Rewrite the above equation as

$$IDF_{i+1}(S) = DF(S) \cup DF(IDF_i(S))$$

We know $DF(S) \subseteq IDF$. So we are left to show $DF(S')$ is in IDF, where $S' = IDF_i(S)$. Let $S' = \{x_1, \ldots x_i, \ldots\}$. Since S' is in IDF (assumption), x_i is in IDF for all i. But the only way the node x_i can be added to IDF is at step 118 of the algorithm. Therefore, x_i must also be inserted into the *OrderedBuckets*, at step 120. Since the algorithm eventually terminates, x_i must be processed as the current root at step 111 by a call to **Visit**(x_i). By Lemma 7.4, $DF(x_i)$ is in IDF when **Visit**(x_i) returns at step 111. And this is true for all nodes $x_i \in S'$. Therefore $DF(S')$ is in IDF. As a result $IDF_{i+1}(S)$ is in IDF, and

$$IDF(N_\alpha) \subseteq IDF \qquad (7.1)$$

We can easily see from Lemma 7.4 that a node is inserted in IDF if it is in the dominance frontier of some node that was previously inserted and retrieved from *OrderedBuckets*. Recall that a node is inserted into *OrderedBuckets* only at two places: step 106 and step 120. A node is inserted in *OrderedBuckets* at step 106 if it is in N_α, and a node is inserted in *OrderedBuckets* at step 120 if it is in the iterated dominance frontier of N_α. Therefore,

$$IDF \subseteq IDF(N_\alpha) \tag{7.2}$$

From Equation 7.1 and 7.2, we get

$$IDF(N_\alpha) = IDF \tag{7.3}$$

■

7.4.2 Complexity

Next we will show that the time complexity of Algorithm 7.1 is $O(|E|)$, where $|E|$ is the number of edges in the DJ graph. Recall that the number of edges in the DJ graph is less than $|N_f| + |E_f|$ (Theorem 3.1). Therefore, the time complexity of Algorithm 7.1 is $O(|N_f| + |E_f|)$. Since $|E_f| \geq |N_f| - 1$, the time complexity of the algorithm is $O(|E_f|)$, which is linear with respect to the number of edges in the flowgraph.

From the proof of the correctness of Algorithm 7.1, readers may have already observed that for any node x in the DJ graph, the node may be processed by a call of **Visit**(x) (which may happen at step 111 or 127) at most once. This observation is a key to the proof of linearity of the algorithm, and is stated as the following lemma.

Lemma 7.5 *When* Algorithm 7.1 *terminates, a node $x \in N$ may be processed by a call to* **Visit**(x) *at most once.*

Proof:

There are only two places a node can be processed by a call to **Visit**():

case 1. at step [127] and case 2. at [111]. It is obvious that step [127] can only be reached by the traversal of an incoming D edge of x. Since there is only one such D edge, x cannot not be processed more than once in case 1. Furthermore, it cannot be processed through case 2 more than once either (since a node can be inserted into and deleted from OrderedBucketsonly once).

Now we prove that it is not possible for x to be processed in both case 1 and case 2. Suppose the contrary is true. Then this is possible only if case 2 happens after case 1. (The condition at step [125] prevents the opposite.) That means node x is already in the *OrderedBuckets* before the current execution of Visit(v) for some v at step [111]. Thus, $x.level \geq v.level$, since $SubTree(v)$ is explored in a top-down fashion. On the other hand, $x.level \leq v.level$, as any node in the *OrderedBuckets* must have a level number no greater than that of the node currently being processed (from Lemma 7.1). This implies $x = v$. But this is impossible since x would been marked *Visited* twice from step [111]. Hence the lemma is true by contradiction. ∎

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

Theorem 7.2 *The time complexity of* Algorithm 7.1 *is* $O(|E|)$.

Proof:

According to Lemma 7.5, a node can be marked *Visited* at most once, and there can be at most $|N|$ calls to **Visit()**. Also, at each node in the procedure **Visit()**, we either visit (through a D edge) or "peek" (through a J edge) all the successor nodes (step [113]) only once. This means that we have effectively visited all the edges in the DJ graph at most once. Hence the complexity of the algorithm is $O(|E|)$. ∎

An astute reader may ask the following question: What about the complexity of inserting/deleting nodes into/from the *OrderedBuckets* structure? It is easy to see that the complexity of inserting a node in the *OrderedBuckets* is $O(1)$. As for the complexity of getting a node from the *OrderedBuckets*, it is again easy

to see that a node will never be inserted in *OrderedBuckets* at the index greater than the *CurrentLevel* (from Lemma 7.1). Each call of **GetNode()** will execute the **while** loop with a monotonically decreasing *CurrentLevel* from $NumLevel - 1$ down to 1 during successive calls for the entire duration of the algorithm (follows from Lemma 7.1). Hence the overall time complexity of deleting all nodes from the *OrderedBuckets* is, in the worst case, $O(|N|)$.

7.4.3 Discussion

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

- First of all notice that the space complexity of our algorithm depends on the size of DJ graphs. In Chapter 3 we established that the size of a DJ graph is linear with respect to the size of its flowgraph. Therefore the space complexity of our algorithm is linear.

- One key point that makes our new algorithm linear is the structure of the *OrderedBuckets*. This structure can be considered as an implementation of a 'Restricted Priority Queue' [CLR90]. One can also consider *OrderedBuckets* to be an ordered set of buckets (although buckets in bucket sort are not ordered as in our case) [CLR90]. The number of buckets that is needed is at most equal to the maximum depth of the dominator tree.

  If one were to use other structures such as a heap, a stack, or a queue, either the proof of correctness would fail (if we still wish to continue to mark the nodes as *Visited* using one color), or the complexity of the algorithm would not be linear (we will need to mark nodes as *Visited* using more than one color). The second situation is similar to finding the iterated dominance frontier by iteratively applying Algorithm 4.1. We can easily show that the complexity of this method will not be linear.

- Let us recall Theorem 3.2. Theorem 3.2 states that if $y \in DF(x)$, then the level number of y will never be greater than the level number of x (i.e., $y.level \leq x.level$). This property is very important in proving the correctness and complexity of our algorithm. Recall that Lemma 4.1 gives a method for computing the dominance frontier of a node from level information and J

edges, and Lemma 7.1 guarantees that a node will never be inserted at an index (in *OrderedBuckets*) greater than the value of *CurrentLevel*. Again, an astute reader will immediately notice the relation between these two lemmas and Theorem 3.2.

- Finally, the framework of our algorithm can easily be adapted to the $\mathcal{APT}$ representation, introduced by Pingali and Bilardi [PB95]. In $\mathcal{APT}$ dominance frontiers are cached at certain nodes, called the *boundary nodes*. The only modification that is necessary in our algorithm is in the procedure **Visit()**. At step 127, before invoking the procedure **Visit**(y) we should check whether y is a boundary node. If so, we avoid visiting the sub-tree rooted at y, since all the candidate nodes to be included in IDF set will be cached at this node. If not we invoke the procedure **Visit**(y).

  Now identifying boundary nodes requires the knowledge of *filtered search techniques*, which is beyond the scope of this dissertation. For details please see [PB95].

7.5 Experiments and Empirical Results

In this section we present our experimental results and give their analysis. We implemented our linear time algorithm using flowgraphs generated from the Parafrase2 compiler and compared it with the original algorithm (due to Cytron et al. [CFR+91]). We will first summarize the major results of our experiments.

- The time complexity of Cytron et al.'s algorithm depends only on the size of the dominance frontier relation. Although, theoretically, the size of the dominance frontier relation can be quadratic, its size appears to be linear in practice. For our test procedures, we found that the size of the dominance frontier relation to be about 0.8 times the size of the DJ graph.[3]

- Cytron et al.'s original algorithm performs better than ours for our test procedures (on average by a factor of 4.46).

[3]The size of a DJ graph is the number of edges in the graph (D edges + J edges); whereas the size of dominance frontiers is the total number of nodes in the dominance frontier set of all the nodes.

- For graphs like ladder graphs and `repeat-until` loops, where the size of dominance frontiers can grow quadratically, our algorithm exhibits linear behavior, whereas Cytron et al's algorithm exhibits quadratic behavior.

We will further elaborate on these results. Table 7.1 gives the summary of our result. The notation used in the table is given below:

Notations used in Table 7.1

Name	Name of the procedure
$\|E_d\|$	Number of edges in DJ graphs (D edges + J edges)
$\|DF_n\|$	Size of the dominance frontier relation represented as a set of nodes
R	The ratio $\frac{\|DF_n\|}{\|E\|}$
$V(o)$	A count of the number of nodes added and removed from the worklist in original algorithm.
$V(n)$	A count of the number of edges visited in our algorithm
V	The ratio $\frac{V(o)}{V(n)}$
$T(o)$	Execution time in milliseconds of the original algorithm
$T(n)$	Execution time in milliseconds of our algorithm
S	The ratio $\frac{T(n)}{T(o)}$

Time measurements shown for $IDF(df)$ and $IDF(new)$ are for computing IDF for a set of randomly chosen nodes (we chose 25 to 30% of the nodes to be N_a). The time complexity of the original algorithm depends on the size of the dominance frontier relation. From the table we can see that, except for 3 procedures, the size of the dominance frontier relation is smaller than the size of the DJ graph. For our test procedures we can see that the average ratio $\frac{|DF_n|}{|E_d|}$ is 0.77. The value of this ratio suggests that Cytron et al.'s algorithm can, at worst, be about 1.29 times faster than our algorithm. In reality Cytron et al.'s algorithm is much faster than ours, as can be seen from speedup ratios given in the table.

We next measured the number edges visited $V(n)$ in our algorithm (for a particular choice of the initial set of sparse nodes), and compared it to the number of nodes added and removed $V(o)$ from the worklist in Cytron et al.'s algorithm. The ratio $\frac{V(o)}{V(n)}$ gives an accurate indication of how good (or how bad) our algorithm will perform when compared to Cytron et al.'s algorithm. The value of this ratio ranges from 2.02 to 8.43, with the average value being 3.84. This suggests that

Name	$\|E_d\|$	$\|DF_n\|$	R	$V(o)$	$V(n)$	V	$T(o)$	$T(n)$	S
aerset	467	323	0.69	113	463	4.10	0.3	1.9	6.33
aqset	263	184	0.70	58	232	4.00	0.2	0.9	4.50
bjt	213	177	0.83	66	212	3.21	0.2	0.7	3.50
card	235	240	1.02	87	234	2.69	0.2	0.7	3.50
chemset	330	282	0.85	86	321	3.73	0.2	1.1	5.50
chgeqz	268	213	0.79	77	253	3.29	0.2	0.8	4.00
clatrs	337	225	0.67	57	319	5.60	0.2	0.9	4.50
coef	154	126	0.82	64	160	2.50	0.2	0.6	3.00
comlr	97	76	0.78	51	103	2.02	0.2	0.5	2.50
dbdsqr	343	269	0.78	83	313	3.77	0.2	1.0	5.00
dcdcmp	205	181	0.88	67	196	2.93	0.2	0.6	3.00
dcop	298	240	0.81	93	288	3.10	0.2	0.9	4.50
dctran	493	407	0.83	87	459	5.28	0.2	1.2	6.00
deseco	259	179	0.69	61	259	4.25	0.2	0.8	4.00
dgegv	246	164	0.67	54	234	4.33	0.2	0.8	4.00
dgesvd	499	314	0.63	58	480	8.28	0.2	1.2	6.00
dhgeqz	433	362	0.84	97	365	3.76	0.2	1.0	5.00
disto	211	158	0.75	65	199	3.06	0.2	0.7	3.50
dlatbs	259	174	0.67	52	243	4.67	0.2	0.8	4.00
dtgevc	485	356	0.73	119	475	3.99	0.3	1.4	4.67
dtrevc	373	263	0.71	70	366	5.23	0.2	0.9	4.50
elprnt	245	176	0.72	87	235	2.70	0.2	1.0	5.00
equilset	467	334	0.72	119	464	3.90	0.3	1.7	5.67
errchk	515	406	0.79	149	498	3.34	0.4	2	5.00
iniset	456	308	0.68	56	472	8.43	0.2	1.4	7.00
init	176	118	0.67	42	167	3.98	0.1	0.5	5.00
initgas	267	203	0.76	111	266	2.40	0.3	1.1	3.67
jsparse	408	307	0.75	138	407	2.95	0.4	1.5	3.75
modchk	455	341	0.75	100	453	4.53	0.3	1.4	4.67
moseq2	246	206	0.84	68	237	3.49	0.2	0.7	3.50
mosfet	333	323	0.97	85	332	3.91	0.2	0.9	4.50
noise	184	136	0.74	53	163	3.08	0.2	0.5	2.50
out	579	490	0.85	157	462	2.94	0.4	1.9	4.75
reader	242	89	0.37	38	249	6.55	0.2	1.0	5.00
readin	637	828	1.30	178	621	3.49	0.3	1.8	6.00
setupgeo	278	193	0.69	81	276	3.41	0.2	1.0	5.00
setuprad	290	217	0.75	104	289	2.78	0.2	1.0	5.00
smvgear	316	337	1.07	161	315	1.96	0.3	1.1	3.67
solveq	298	260	0.87	143	296	2.07	0.3	1.0	3.33
twldrv	258	202	0.78	65	250	3.85	0.2	0.8	4.00
Average	327.95	259.67	0.77	87.5	315.65	3.84	0.23	1.04	4.46

Table 7.1: A comparison of our algorithm with the original algorithm.

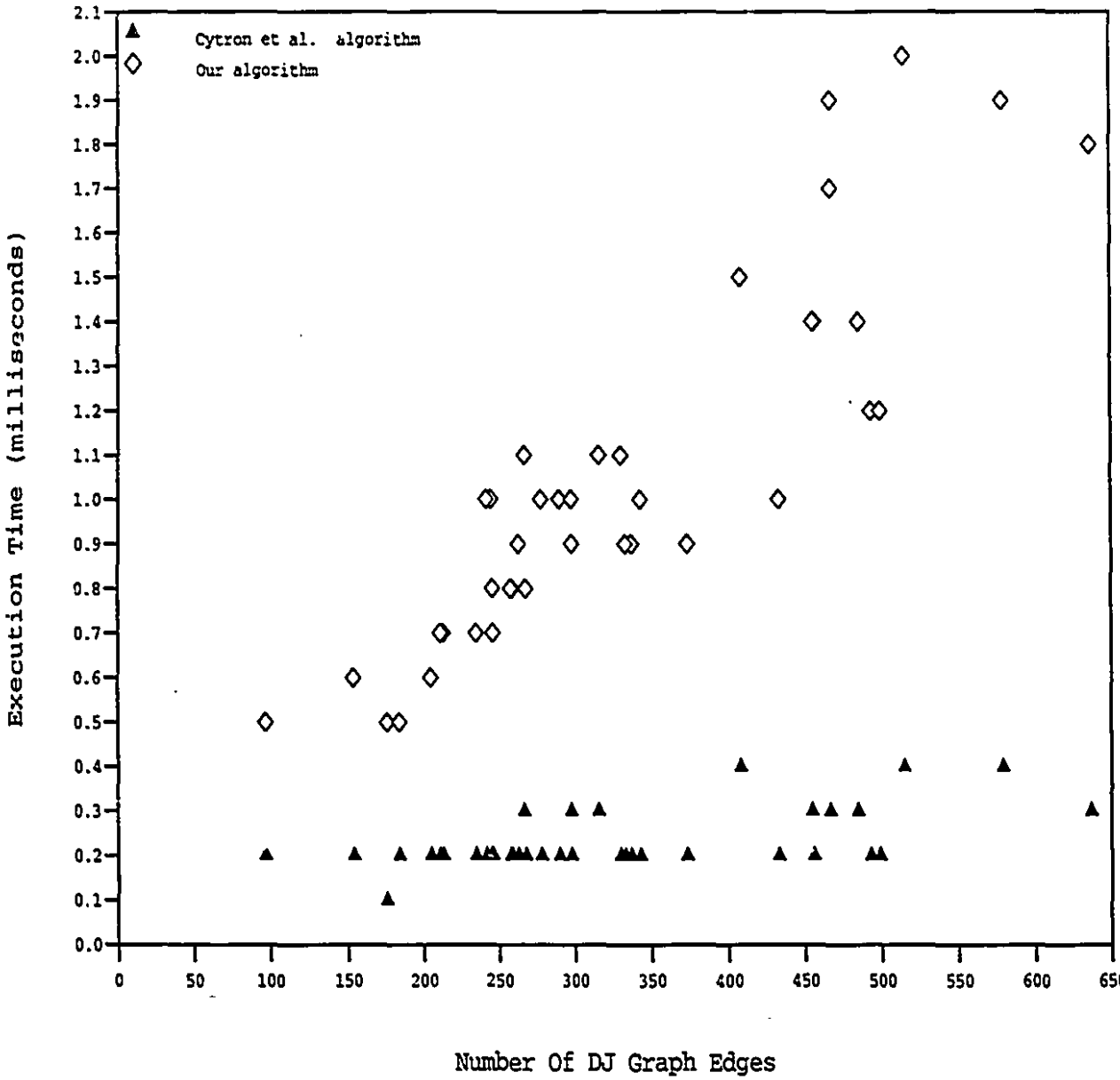


Figure 7.3: Performance of the two algorithms on our test procedures.

Cytron et al.'s algorithm is about 3.84 times faster than ours. This can again be confirmed with the speedup ratio given in Table 7.1. Another point to note is that the value of $V(o)$ is close to the number of edges in a DJ graph. This suggests that in our algorithm we search almost the whole DJ graph for each query.

We next measured the actual execution time of the two algorithms. As can be seen from the table both algorithms are very fast in practice. The actual execution time for our algorithm ranges from 0.5 milliseconds to 2.0 milliseconds, with the average execution time being 1.04 milliseconds. For Cytron et al.'s algorithm the execution time ranges from 0.1 milliseconds to 0.4 milliseconds, with the average execution time being 0.23 milliseconds. The ratio $\frac{T(n)}{T(o)}$ is given in column S. The value of this ratio ranges from 2.5 to 7.0, with the average value being 4.46.

Figure 7.5 gives the performance of the two algorithms plotted against the number of DJ graph edges. As can be seen from the plot our algorithm has a linear time behavior whereas Cytron et al.'s has a constant time behavior. The reason for is because the number of nodes added and deleted from the worklist in Cytron et al.'s algorithm is much smaller than the size of the dominance frontier relation, and does not seem to depend on sizes of flowgraphs. Whereas in our algorithm, we visit all the edges of a DJ graph looking for candidate nodes to be included in the iterated dominance frontier set.

For graphs like deeply nested `repeat-until` loops and ladder graphs, our algorithm performs better than the original algorithm. For these graphs, the size of dominance frontiers grows quadratically with respect to the number of nodes in the graph. Figure 7.5 shows the performance of the two algorithms for increasing taller repeat-until loops of the form shown in Figure 7.4. From the execution profile we can see that our method is indeed linear, whereas the original algorithm is quadratic.

From our experiments we can see that even a quadratic time algorithm can perform better than a linear time algorithm for most practical programs. As we mentioned in the introduction of this chapter, the framework of the algorithm presented here can be adapted to other representations, like the *APT* [PB95]. When our algorithm is implemented on *APT*, the performance of the algorithm is better than Cytron et al.'s original algorithm [Pin95].

Next we will bring out some interesting and debatable issues concerning the choice of DJ representation for dominance frontiers. One reason the size of the

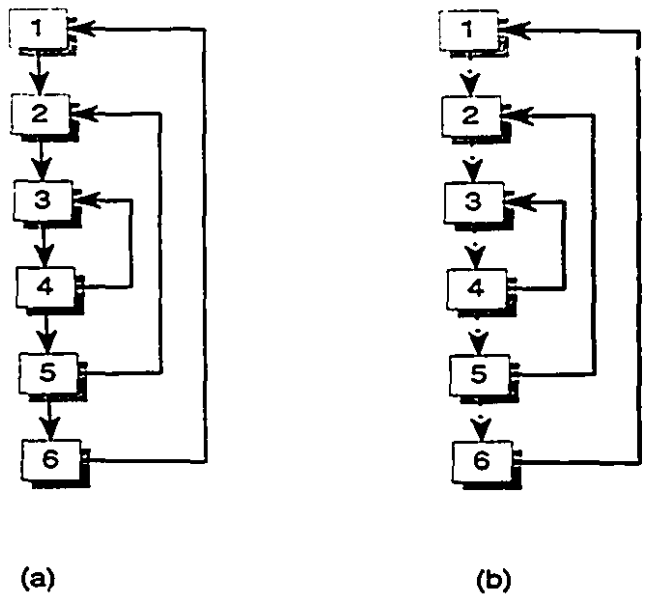


Figure 7.4: A nested repeat-until flowgraph and its DJ graph.

dominance frontier relation is smaller than the size of the corresponding DJ graph is because we represent each basic block as a flowgraph node. Instead, if we represent each statement (or worse, each 3-address instruction) as a flowgraph node, then the size of the dominance frontier relation will dominate the size of the corresponding DJ graph. Now why would any one represent each 3-address instruction to be a flowgraph node? This is an engineering issue. One major advantage of this representation is that it simplifies data flow analysis. Using basic block representation, each data flow analysis has to be performed at two levels: local analysis (that summarizes the effect of a basic block) and global analysis. Also, once the global solution is determined we have to propagate this information to instructions (or statements) within each basic block. This two-level analysis is not needed when each flowgraph node represent a 3-address instruction. Tjiang and Hennessy give several drawbacks of representing each basic block to be a flowgraph node [TH92]. Based on their study they recommend and advocate that each instruction be represented as a flowgraph node [TH92].

We will conclude this section with a final remark. As discussed in Chapter 4, *APT* optimizes both query time and space usage. In *APT* dominance frontiers of certain nodes are cached. Using a "tuner", called α in the paper [PB95],

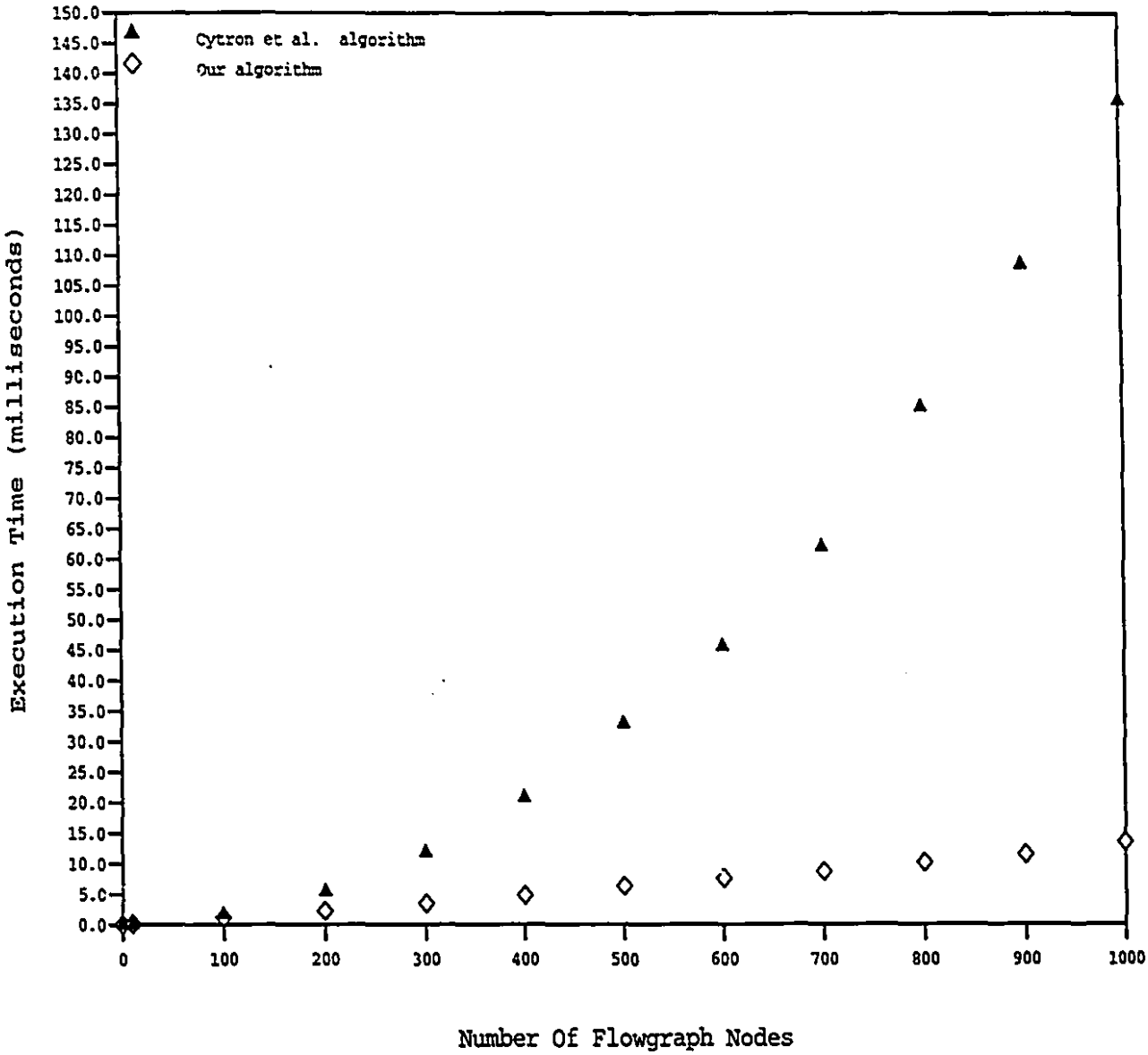


Figure 7.5: Performance of the two algorithms on nested repeat-until graphs.

the authors show how to control the caching of dominance frontiers. The authors also show how their representation can be considered as a spectrum of dominance frontier representations, of which our DJ graph is at one end (with no caching) and the full dominance frontier representation of Cytron et al is at the opposite end (with full caching). In our IDF algorithm we walk down a DJ (sub)graph along D edges looking for candidate nodes via J edges to be included in the set (of IDF). We can avoid walking all the way down the dominator tree if we cached the dominance frontiers at certain nodes. Pingali and Bildari show how to cleverly cache dominance frontiers at certain boundary nodes in a preprocessing step. Once we have cached them, we can limit the top-down traversal upto these boundary nodes. Another point to note is that, in $\mathcal{APT}$, we need not use a worklist model of Cytron et al. for computing the iterated dominance frontier relation even with full caching. This actually allowed them to detect a discrepancy in our experimental results originally published in [SG95b]. In [SG95b] we reported that our algorithm is, on average, faster than Cytron et al.'s algorithm by a factor 5. This is because, in our original implementation of Cytron et al.'s algorithm, we chose bit-vectors to represent the worklist used in the algorithm. This attributed to the poor performance of our implementation of Cytron et al.'s algorithm. For the implementation of the results reported in this chapter we used Sparse Set representation of Briggs and Torczon to implement the worklist [BT93]. This improved the performance of Cytron et al.'s algorithm over our algorithm, and performs better than ours by a factor 5 on average.[4]

7.6 Discussion and Related Work

The sparse evaluation technique is becoming popular, especially for analyzing large programs. To this end, many intermediate representations have been proposed in the literature for performing sparse evaluation [CFR+91, CCF91, JP93, WCES94]. The algorithms for constructing these intermediate representations have one common step—determining program points where data flow information must be merged, the so called ϕ-nodes. The notion of ϕ-nodes dates back to the work of Shapiro and Saint [SS70] (as noted in [CFR+91]). Subsequently,

[4] I thank Prof. Keshav Pingali for pointing out this discrepancy in our original publication, which we corrected in this chapter.

others have proposed sparse evaluation in one form or another that is related to this prior work [RT82, CLZ86, CF87]. Rosen et al. proposed another approach for constructing SSA form using depth-first search algorithm [RWZ88]. Their method is restricted to reducible flowgraphs. Cytron et al. [CFR+89] gave an algorithm for computing ϕ-nodes for arbitrary flowgraphs. This algorithm is original in the sense that it is based on dominance frontiers and can handle arbitrary flowgraph structure. The time complexity of this algorithm depended on the size of the dominance frontier relation, which is $O(|N|^2)$. But Cytron et al. have shown that the size of the dominance relation is linear in practice. Recently, Cytron and Ferrante improved the quadratic behavior of computing ϕ-nodes to be almost linear time [CF93]. The time complexity of the new algorithm is $O(|E| \times \alpha(|E|))$, where $\alpha()$ is the inverse-Ackermann function [CF93]. More experimental studies are needed to evaluate the performance of this algorithm when applied to real programs.

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

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

There is much related work that uses SSA like representation, for example, the Program Dependence Web [BMO90] and the Value Dependence Graph [WCES94],

and our algorithm could improve the complexity of constructing these related intermediate representations. Also there are many optimizations that use SSA form for efficient implementation, for example, constant propagation [WZ85], value numbering [RWZ88], register allocation [Bri92], code motion [CLZ86], etc. Our algorithm could improve the overall running time of these optimizations.

We would like to bring some important concerns regarding the applicability our algorithm for constructing SSA form. In [CFR+91], the authors write: "The method presented here is $O(R^3)$ at worst, but Section 8 gives evidence that it is $O(R)$ in practice. The earlier $O(R^2)$ algorithm have no provision for running faster in typical cases; they appear to be intrinsically quadratic."[5] Our method also is intrinsically quadratic in the above sense. The algorithm of Cytron et. al appears linear for typical cases is because they assume that the size of dominance frontiers to be constant. As can be seen in their paper (Figure 20 in [CFR+91]), the size of dominance frontier is small for smaller program sizes, but is proportional to the number of program statements for larger program sizes. Recall that the framework of our algorithm can easily be adapted to the $\mathcal{APT}$ representation, introduced by Pingali and Bilardi [PB95]. In $\mathcal{APT}$ dominance frontiers are cached at certain nodes, called the *boundary nodes*. As we demonstrated earlier (Section 7.4.3), the only modification that is necessary in our algorithm is in the procedure **Visit()**. Using the framework of our algorithm in conjunction with the $\mathcal{APT}$ representation, one can speedup the overall construction of the SSA form and SEGs.

[5] Where R is the size of the program.

Chapter 8

Incremental Computation of Dominator Trees

A fanatic is one who can't change his mind and won't change the subject.
—Winston Churchill

They must often change, who would be constant in happiness or wisdom.
—Confucius

Dominator trees have many applications in compiler optimization and data flow analysis. It is important that the dominator tree be correctly maintained throughout a multi-pass compiler. In this chapter we present a new framework for incrementally maintaining the dominator tree of a flowgraph, when the flowgraph is subjected to incremental changes, such as insertion and deletion of an edge. Unlike previous approaches our approach can handle arbitrary flowgraph changes, including irreducibility. A novel aspect of our approach is that we use simple properties of dominance frontiers and iterated dominance frontiers to update the dominator tree. Another interesting aspect of our approach is that we update DJ graphs (rather than dominator trees) which subsumes the problem of updating dominator trees.

We begin the chapter by introducing and motivating the problem. In Section 8.2 and Section 8.3 we present our update algorithm for edge insertion and edge deletion, respectively. In Section 8.4 we give some experimental results, comparing the running time of our incremental algorithm with the almost linear Lengauer and Tarjan's exhaustive dominator algorithm [LT79]. Finally, in Section 8.5, we discuss related work, and give our concluding remarks.

8.1 Introduction and Problem Definition

Dominator trees have many applications in compiler optimization and data flow analysis [ASU86]. For example, construction of the SSA form, or hoisting loop invariants requires that the dominator tree be correctly maintained at all times. Now it is possible that the flowgraph structure may change during program optimization (e.g., loop transformation, dead-code elimination, etc.) It is important that the dominator tree is correctly updated during such flowgraph changes.

In this chapter we present a new algorithmic framework for incrementally maintaining the dominator tree of an arbitrary flowgraph. For this problem previous work most relevant to this chapter includes only the Carroll-Ryder algorithm [CR88] and the Ramalingam-Reps algorithm [RR94]. Both methods are restricted to reducible flowgraphs. By contrast, our approach can handle irreducible as well as reducible flowgraphs. For the case where an edge is inserted, our algorithm has an $O(|E|)$ time complexity, where $|E|$ is the number of edges in the DJ graph—better than previous approaches [CR88, RR94]. For the deletion case, our new incremental algorithm is also competitive in terms of running time. It is expected to run faster on the average cases while not compromising its worst-case time complexity.

Updating dominator trees is a non-trivial problem. Carroll and Ryder pointed out in [CR88]:

> "The inherent difficulty in the dominator update problem lies in the 'non-locality' of domination, to wit, given two nodes x and y in the flow graph, whether x dominates y depends on the presence or absence of paths through nodes arbitrarily far from either x or y. Adding or removing a single flow graph edge – an act which can add or remove large numbers of paths – can thus affect domination between nodes arbitrarily far from the altered edge."

Consider the flowgraph and its DJ graph shown in Figure 8.1(a) and (b), respectively. Let us see what happens when a new edge $2 \rightarrow 4$ is inserted. With this new edge, the dominance relation for many nodes is affected. For example, in the newly modified flowgraph, node 3 will no longer dominate 4. Actually node 3 ceases to dominate 6 and 8 as well. The reason for this is obvious: By inserting $2 \rightarrow 4$ we have created alternative paths from START to nodes 4, 6 and 8 that do

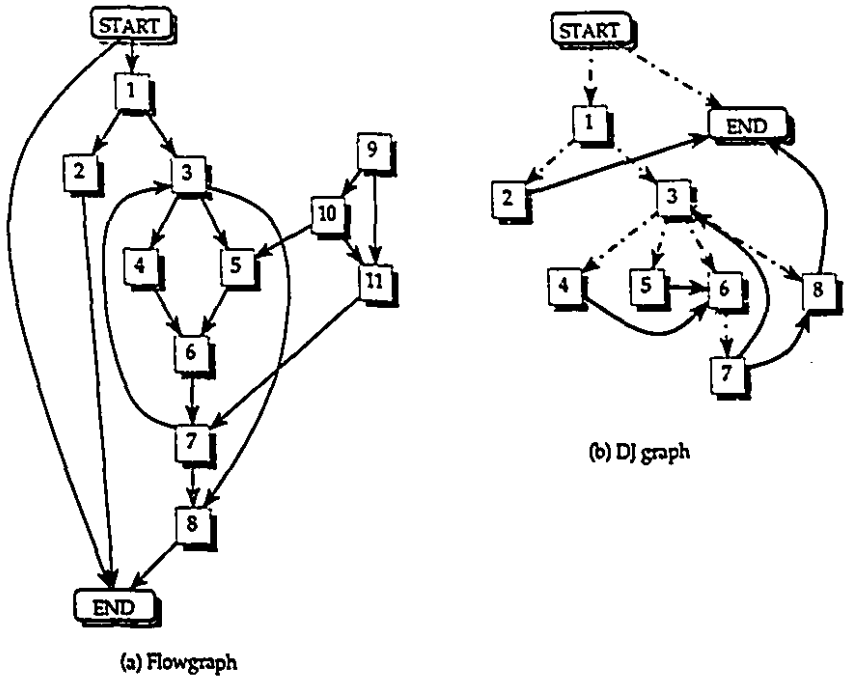


Figure 8.1: Another example of a flowgraph and its DJ graph.

not pass through node 3. Notice that the insertion of $2 \rightarrow 4$ has affected node 8 that is far from both nodes 2 and 4. The situation is also true (but opposite) when an edge is deleted; this deletion can again affect nodes arbitrarily far away.

Also notice in the above example, the flowgraph becomes irreducible after $2 \rightarrow 4$ is inserted. Previous approaches to this problem that we are aware of, the Carroll-Ryder and the Ramalingam-Reps algorithms, cannot proceed further when this happens. One of our important observations is that the dominator tree alone is not sufficient to capture all the path information in a flowgraph; it only gives one type of *path* relationship among nodes in a flowgraph – meaning, "a node x dominates another node y iff all *paths* from START to y must pass through x". By adding J edges to the dominator tree (*á la* DJ graph) we capture all the path information in a flowgraph. Thus in a DJ graph we capture both the path information and the domination relation in a unified representation. This allows us to easily compute the set of affected nodes when an edge is added or deleted. A node is called *DomAffected* iff its immediate dominator changes because of a flowgraph update. Therefore, a key question to be answered in the rest of this

chapter is: how can we efficiently compute the set of affected nodes no matter how far away they may be from the updated edge?

To summarize, the problem we consider in the next two sections is the DJ graph update problem: We want to maintain the DJ graph for the reachable subgraph of a flowgraph in which changes are made one at a time. This problem subsumes the dominator tree update problem. The algorithms given in Section 8.2 and Section 8.3 collectively form our incremental algorithm for maintaining the DJ graph when a new edge is inserted and an existing edge is deleted, respectively. These algorithms handle situations where x and y are reachable from START both before and after an update. We will handle other situations as special cases. Also, to simplify the presentation and without loosing generality we will consider only the following types of incremental changes to the flowgraph structure: (1) insertion of a new edge, (2) deletion of an existing edge. One can implement other more complex changes using a sequence of these two (primitive) changes [Mar89, RR94, CR88].

8.2 Dominator Update: Insertion of an Edge

In this section, we present a simple algorithm for updating the DJ graph (and hence the dominator tree) of a flowgraph in response to a flowgraph edge insertion. Recall that when an edge $x_f \rightarrow y_f$ is inserted, it can affect nodes arbitrarily far away. One of our key observations is that all the *affected* nodes must be in the set $\{y\} \cup IDF(y)$. The reason for this is as follows: By inserting the edge $x_f \rightarrow y_f$, we may have created a path (in the flowgraph) from START_f to a node, say u_f, in $\{y_f\} \cup IDF(y_f)$ such that the path includes $x_f \rightarrow y_f$ but bypasses the immediate dominator of u_f. However, not all the nodes in $\{y\} \cup IDF(y)$ will be truly affected. We will formally prove later that the affected nodes are only those additionally satisfying certain level constraints.

Once we find the *exact* set of affected nodes, we need to answer: What will be their new immediate dominator after the edge insertion? First note that $nca(x_d, y_d)$, the nearest common ancestor of nodes x_d and y_d on the dominator tree, will definitely dominate all the affected nodes. Ramalingam and Reps gave a stronger claim to answer this question: Not only does $nca(x_d, y_d)$ dominate all the affected nodes, but also it is actually their immediate dominator [RR94]. We

will utilize this claim in our algorithm too.

Before presenting our algorithm for handling an edge insertion, we want to clarify the following point: We do not incrementally maintain the iterated dominance frontiers (IDFs) for all the nodes in our approach. There are two reasons. First, we have a very efficient algorithm for computing $IDF(y)$ in linear time when we are told that $x_f \rightarrow y_f$ is *the* newly inserted edge [SG95b]. Second, if we were to update IDFs, we would need to pay extra space and time overhead. The time required for updating IDFs for all the nodes can be much longer than that spent in computing $IDF(y)$ for one node y.

8.2.1 Insertion Algorithm

Let $x_f \rightarrow y_f$ be the newly inserted flowgraph edge. Assume that both x_f and y_f are reachable from START even before the insertion. Algorithm 8.1 below gives a procedure to restructure the DJ graph in response to changes in the dominance relation when $x_f \rightarrow y_f$ is inserted. There are two supporting functions used in the algorithm: (1) $link(x_d, y_d, Jedge\ or\ Dedge)$ inserts into the DJ graph a new edge $x_d \rightarrow y_d$ of the appropriate type; and (2) $cut(x_d, y_d)$ deletes $x_d \rightarrow y_d$.

In the algorithm, we first compute $z_d = nca(x_d, y_d)$ (step 143). Then we compute, at step 144, the set of affected nodes $\text{DomAffected}_\text{I}(y_d) = \{w_d | w_d \in (\{y_d\} \cup IDF(y_d)$ and $w_d.level > z_d.level + 1\}$. That is, there are two conditions for a node to be in $\text{DomAffected}_\text{I}(y_d)$: (1) it must be in $\{y_d\} \cup IDF(y_d)$, and (2) its level number must be greater than $nca(x_d, y_d).level + 1$. Using our linear time algorithm for computing $IDF(y_d)$ on-the-fly, we thus can avoid processing any node whose level number is not greater than $nca(x_d, y_d).level + 1$ (Chapter 7 and [SG95b]).

Once we have computed $\text{DomAffected}_\text{I}(y_d)$, we pull up each affected node w_d and make z_d (i.e., $nca(x_d, y_d)$) its new immediate dominator (step 153). We also delete the D edge to every affected node from its (old) immediate dominator (step 149). If there is a flowgraph edge between these two nodes, we insert a J edge in place of the deleted D edge at step 150. Finally, we update the level number for all the descendant nodes of every affected nodes at step 156. The complete algorithm is given below.

Algorithm 8.1 *The following algorithm updates the DJ graph of a flowgraph when a new flowgraph edge $x_f \rightarrow y_f$ is inserted.*

UpdateInsertEdge(x_d, y_d)
{
143: $z_d = nca(x_d, y_d)$
144: DomAffected$_1(y_d) = \{w_d | w_d \in (\{y_d\} \cup IDF(y_d))$ and
$w_d.level > z_d.level + 1\}$
145: **if**$(z_d! = x_d)$
146: $link(x_d, y_d, Jedge)$ /* if $z_d = x_d$ then we should */
/* insert a D edge. See step 153 */
147: **foreach** $w_d \in$ DomAffected$_1(y_d)$ **do**
148: $u_d = idom(w_d)$; /* u_d is the (old) immediate dominator */
149: $cut(u_d, w_d)$ /* cut the old D edge */
150: **if**($u_f \rightarrow w_f$ *is a flowgraph edge*)
151: $link(u_d, w_d, Jedge)$ /* insert a J edge from u_d to w_d */
152: **endif**
153: $link(z_d, w_d, Dedge)$ /* new D edge */
154: **endfor**
155: **foreach** $w_d \in$ DomAffected$_1(y_d)$ **do**
156: **UpdateLevelNumber**(w_d) /* Update the level for nodes */
/* in the $SubTree(w_d)$ */
157: **endfor**
}

Example 8.1

Consider our example flowgraph in Figure 8.1(a) and its DJ graph in Figure 8.1(b). Let us insert a new edge $2 \rightarrow 4$ in the flowgraph. The resulting flowgraph is shown in Figure 8.2(a). From the DJ graph (in Figure 8.1(b)) we can find $nca(2,4) = 1$ and compute DomAffected$_1(4) = \{4, 6, 8\}$. After this we pull up all the affected nodes and make node 1 their new immediate dominator. At the same time, we remove D edges $3 \rightarrow 4$, $3 \rightarrow 6$ and $3 \rightarrow 8$. Since node 2 does not dominate 4, we insert a new J edge $2 \rightarrow 4$. We also insert J edges $3 \rightarrow 4$ and $3 \rightarrow 8$ in

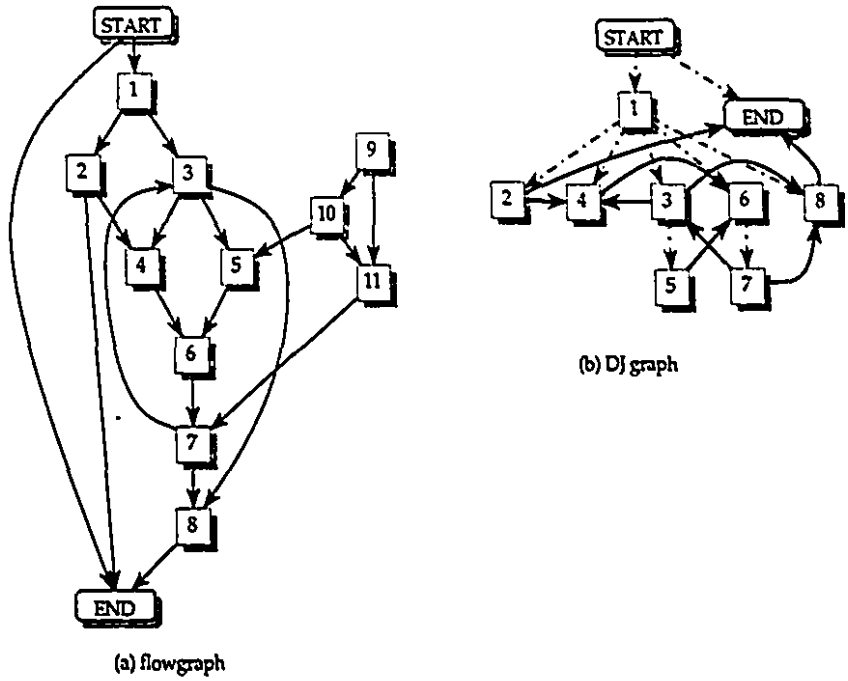


Figure 8.2: The flowgraph and its DJ graph after $2 \rightarrow 4$ is inserted.

the DJ graph, because their counterparts exist in the flowgraph. The updated DJ graph is given in Figure 8.2(b).

8.2.2 Correctness and Complexity

In this subsection, we prove the correctness Algorithm 8.1 (Theorem 8.1) and analyze its complexity (Theorem 8.2). Without loss of generality, we assume that x_f and y_f are both reachable before and after the insertion of $x_f \rightarrow y_f$.

Theorem 8.1 is the main theorem that establishes the correctness of the algorithm. Its proof is based on Lemmas 8.1 and 8.4. Lemma 8.1 claims that a unique node $z_d = nca(x_d, y_d)$ will be the new immediate dominator of all the affected nodes. Lemma 8.4 gives a necessary and sufficient condition to determine the *exact* set of affected nodes. Its validity is further based on two other lemmas, Lemma 8.2 and Lemma 8.3. Lemma 8.2 claims that if $u \in IDF(y)$, then $idom(u)$ strictly dominates y. Lemma 8.3 establishes a relation between any two nodes u and v when $v \notin IDF(u)$ and v is reachable from u.

We will begin the proof chain by first defining the notion of *DomAffected* when an edge is updated in the corresponding flowgraph.

Definition 8.1 *A node is said to* **DomAffected** *iff its immediate dominator node changes because of an update in its flowgraph.*

Given the notion of $DomAffected$, Lemma 8.1 claims that $nca(x_d, y_d)$, before the DJ graph is updated, will be the new immediate dominator of all the $DomAffected$ nodes (after the update). This was originally given by Ramalingam and Reps in [RR94]. The lemma is one of the key result to support the correctness of our algorithm. It also forms the basis for our second key result (Lemma 8.4).

Lemma 8.1 *Let $x_f \rightarrow y_f$ be a newly inserted edge and let $z_d = nca(x_d, y_d)$. Then z_d must immediately dominate every $DomAffected$ node after the insertion.*

Proof:

Let w be $DomAffected$ after $x_f \rightarrow y_f$ is inserted. We will first show that z must dominate every affected node before and after the insertion of the edge $x_f \rightarrow y_f$. Insertion of an edge can only reduce the domination relation [PM72, ASU86], and therefore can only shrink the height of the dominator tree. From this we can conclude that the new immediate dominator of w must have dominated w even in the original flowgraph. But we know that after inserting the edge $x_f \rightarrow y_f$, u will no longer immediately dominate w (since w is affected). From this we can conclude that, after inserting the edge, there exists a path P from START to w that does not pass through u. But this path should contain the edge $x_f \rightarrow y_f$ (since this was the only new edge inserted). All nodes on this path must also contain z (since $z_d = nca(x_d, y_d)$). Therefore, by definition of dominance relation, z must dominate every node on the path P.

Next we will show that z will be the new immediate dominator of all the $DomAffected$ nodes. Let v be the new immediate dominator of w after the insertion. Suppose that v were not z. Then z stdom v stdom w. Consider any path P: $z_f \xrightarrow{*} x_f \rightarrow y_f \xrightarrow{*} w_f$ in the new flowgraph. P must include v_f since $v = idom(w)$. There are two possibilities:

1. Node v_f is in $z_f \xrightarrow{*} x_f$. This implies that v_f is also on any path from z_f to x_f before $x_f \rightarrow y_f$ is inserted. Therefore, v should have been the nearest common ancestor of x and y. This contradicts $z_d = nca(x_d, y_d)$.
2. Node v_f is not in $y_f \xrightarrow{*} w_f$. This implies that the insertion of $x_f \rightarrow y_f$ should not have $DomAffected$ v. Therefore, v is already w's immediate dominator before the edge insertion, so w should not be $DomAffected$. This contradicts our assumption that w is an $DomAffected$ node.

Since neither possibility can be true, v must be the same as z; that is, z must be the immediate dominator of any $DomAffected$ node. ∎

To prove our second key result (Lemma 8.4), we need the following two supporting lemmas. Lemma 8.2 establishes a relation between a node w and the immediate dominator of a node $u \in IDF(w)$. Recall that we proved a similar result in Chapter 3 (Theorem 3.2). The proof of Lemma 8.2 follows from Theorem 3.2.

Lemma 8.2 *If $u \in IDF(w)$, then $idom(u)$* **stdom** *w.*

Proof:

Follows from Theorem 3.2. ∎

The next supporting lemma establishes a relation between any two nodes u and w such that u is reachable from w but is not in $IDF(w)$. As illustration, consider Figure 8.1(a), node 7 is reachable from node 4, but is not in $IDF(4)$. An astute reader can notice that node 6, which strictly dominates node 7, is in $IDF(4)$. The following lemma generalizes the above illustration. In other words, let u be reachable from another node $w \neq u$, and let $u \notin IDF(w)$. Then there exists a node $s \in w \cup IDF(w)$ such that s stdom u. This result is similar to Lemma 4 in [CFR+91].

Lemma 8.3 *Let u be reachable from another node $w \neq u$, and let $u \notin IDF(w)$. Then there exists a node $s \in w \cup IDF(w)$ such that s* **stdom** *u.*

Proof:

Assume w !stdom v (otherwise, the proof is trivial).

Since u is reachable from w, let P be a path from w to u. Let s be the node on P such that $s \in IDF(w)$ and is closest to u among all nodes which are on P and are in $IDF(w)$. Such a s must exists (e.g. in the extreme w can be such a s).

Now we claim that all nodes on P between s and u must be dominated by s. We show this claim by contradiction. Assume that some of these nodes are not dominated by s. Let t be one such node that is closest to s on P. Then by definition of dominance frontiers, t must be in $DF(s)$. This is because, t is on the path P, but is not dominated by s. Therefore, there must be an edge $r \rightarrow t$ such that s dom r. From Lemma 4.1 we know that t must be in $DF(s)$. Now since $t \in DF(s)$, t must also be in $IDF(w)$. But we assumed that s is the last node in $IDF(w)$ which lies on P—a contradiction. Therefore, u must be strictly dominated by s.

■

Next we present Lemma 8.4. Using this lemma we can determine the exact set of nodes that are indeed $DomAffected$ when a flowgraph edge $x_f \rightarrow y_f$ is inserted.

Lemma 8.4 *Let $x_f \rightarrow y_f$ be a newly inserted edge in the flowgraph, and let $z = nca(x, y)$. Let* $\text{DomAffected}_1(y) = \{v | v \in (\{y\} \cup IDF(y))$ **and** $v_d.level > z_d.level + 1\}$. *Then a node u is $DomAffected$ iff $u \in \text{DomAffected}_1(y)$.*

Proof:

The "if" part: We want to show that if u is in $\text{DomAffected}_1(y)$, then it is $DomAffected$. Let $w = idom(u)$ before $x_f \rightarrow y_f$ is inserted. Since u is in $\text{DomAffected}_1(y)$, $u_d.level > z_d.level + 1$. This implies $w_d.level = u_d.level - 1 > z_d.level$. Thus we first conclude that w_d cannot be the same as z_d.

The fact of $u \in \text{DomAffected}_1(y)$ also implies $u = y$ or $u \in IDF(y)$. If $u = y$, then w stdom y (by our assumption above). If $u \in$

$IDF(y)$, then $w = idom(u)$ stdom y by Lemma 8.2. In summary, w must strictly dominate y. But w !stdom x because otherwise w, in place of z, would have been the nearest common ancestor of x and y (since $w.level > z.level$).

Now consider the insertion of $x_f \rightarrow y_f$ in the flowgraph. Since w does not strictly dominate x, the edge insertion creates at least one path $\text{START}_f \stackrel{*}{\rightarrow} x_f \rightarrow y_f \stackrel{*}{\rightarrow} u_f$ that bypasses w_f both when $u = y$ and when $u \in IDF(y)$. Consequently, u is truly $DomAffected$.

The "only if" part: Here we will show that if u_d is $DomAffected$ then u_d is in $\text{DomAffected}_\text{I}(y_d)$.

First we will show that if u_d is $DomAffected$ then $u_d.level > z_d.level+1$. The proof of this is based on the following observation: From Lemma 8.1 we know that z_d will immediately dominate every $DomAffected$ node after the update. Therefore, z_d must strictly dominate every $DomAffected$ nodes. Hence $u_d.level > z_d.level + 1$.

Next we will show that u_d is either y_d or is in $IDF(y_d)$. It is obvious to see that that if u_d is same as y_d, u_d is in $\text{DomAffected}_\text{I}(y_d)$.

Now assume that $u \neq y$. We will show that $u \in IDF(y)$. Assume to the contrary that u is not in $IDF(y)$. Since u is reachable from y, either y strictly dominates u or there must be a node s that strictly dominates u and $s \in IDF(y)$ (follows from Lemma 8.3). If y strictly dominates u, then u is not affected. Since there is a node s as above, we have z dom $idom(s)$ (by Lemma 8.2) which strictly dominates s which strictly dominates u, which contradicts the fact that z is the immediate dominator of u. Therefore u must be in $IDF(y)$.

■

Finally, we state our first main theorem that establishes the correctness of our algorithm for handling an edge insertion.

Theorem 8.1 Algorithm 8.1 *correctly updates the DJ graph of a flowgraph when a new edge $x_f \rightarrow y_f$ is inserted in the flowgraph.*

Proof:

From Lemma 8.1 we know $nca(x_d, y_d)$ must immediately dominate all the $DomAffected$ nodes. Now at step 144 we use the result of Lemma 8.4 to determine which nodes are indeed $DomAffected$. From these we can easily see the validity of the theorem. ∎

Our second main theorem gives the worst-case time complexity of Algorithm 8.1

Theorem 8.2 *Assume that both x_f and y_f are reachable before the insertion of $x_f \rightarrow y_f$. Then the worst-case time complexity of* Algorithm 8.1 *is $O(|E|)$, where $|E|$ is the number of edges in the DJ subgraph induced by the nodes in $SubTree(nca(x_d, y_d))$.*

Proof:

The dominating step in Algorithm 8.1 is computing the set $IDF(y)$. Using the result of Chapter 7 we can easily see that the time complexity of Algorithm 8.1 is $O(|E|)$. ∎

Using the result of Chapter 7 we can compute the set $IDF(y)$ in linear time. But as we showed in that chapter, our linear time algorithm actually performs worse than the quadratic-time algorithm given by Cytron et al. [CFR+91]. Also, our linear time algorithm potentially searches the whole DJ graph while computing the iterated dominance frontier relation. In incremental analysis, it is important that we want to limit the search only to small portion of the graph. In Chapter 9 we will show how to improve the efficiency of the dominator tree update algorithm by pruning search during the computation of $IDF(y)$. This algorithm requires that the dominance frontier relation be correctly maintained. In Chapter 9 we will show how to incrementally maintain the dominance frontier relation.

8.2.3 Other Cases

Let $x_f \rightarrow y_f$ be the newly inserted edge in the flowgraph. Here we will describe how to handle other cases where the reachability of x_f and y_f from $START_f$ is different from what we have assumed in the previous subsection. The first case is where x_f is not reachable, for which we do nothing because we maintain the DJ graph only for the reachable subgraph REACH(START).

In the second case, y_f becomes reachable only after the edge insertion. For this we first build (using exhaustive algorithm) the DJ subgraph for the sub-flowgraph induced by nodes reachable from y_f but not reachable from START_f. In constructing this DJ subgraph, we treat y_d as its root. Since x must dominate y, we then insert a D edge from x_d to y_d (to connect the newly built DJ subgraph with the existing DJ graph). Finally, from the viewpoint of updating DJ graph, we pretend each edge $u_f \rightarrow v_f$ to be a newly inserted flowgraph edge, where u_f becomes reachable only after $x_f \rightarrow y_f$ is inserted, and v_f is reachable even before the edge insertion. This surprisingly corresponds to the case discussed in the previous subsection. To complete the DJ graph update, we invoke Algorithm 8.1 once for each $u_f \rightarrow v_f$.

For example, node 9 is unreachable in Figure 8.1(a), but becomes reachable after we insert $1 \rightarrow 9$. Therefore, we first construct the DJ subgraph for the sub-flowgraph induced by nodes 9, 10, and 11. Then we insert a D edge from 1 to 9. Finally, we use Algorithm 8.1 to update the DJ graph once for $10 \rightarrow 5$ and once for $11 \rightarrow 7$ as if they were newly inserted flowgraph edges.

8.3 Dominator Update: Deletion of an Edge

In this section, we show how to update a DJ graph when a flowgraph edge $x_f \rightarrow y_f$ is deleted. The effect of deleting an edge is opposite and complementary to that of inserting the same edge. When inserting $x_f \rightarrow y_f$, $z_d = nca(x_d, y_d)$ will be the new immediate dominator node for all the $DomAffected$ nodes. Therefore, we pull up all the $DomAffected$ nodes in the DJ graph to the level of $nca(x_d, y_d).level + 1$. Also notice that the "old" immediate dominators of all the affected nodes will be different. On the contrary when an edge $x_f \rightarrow y_f$ is deleted, all the affected nodes should be at the same level as node y. Also, the new immediate dominators of all the affected nodes will be same as the "old" immediate dominators, as mentioned above for the insertion case. For the deletion case, computing both the *exact* set of affected and the corresponding new immediate dominators of the nodes in the affected set is difficult. To overcome the first difficulty, we use a conservative approximation for the set of $DomAffected$ nodes. This set of *possibly* $DomAffected$ nodes is $\text{DomAffected}_D(y_d) = \{w_d | w_d \in IDF(y_d)$ and $w_d.level =$

$y_d.level\}$.[1] Since this set is a safe approximation, not every node in the set will be pulled down in response to a flowgraph edge deletion. To overcome the second difficulty while avoiding an exhaustive algorithm, we observe that the new immediate dominator of any $DomAffected$ node must be a descendant of $idom(y_d)$. Therefore, if any update on the DJ graph needs to happen, it will only affect the nodes in $SubTree(idom(y_d))$. Based on these two observations, we present an efficient incremental algorithm for only computing the possibly $DomAffected$ nodes' *new dominators*, which are then used to compute their new *immediate* dominators.

8.3.1 Deletion Algorithm

As pointed out previously, our approach to updating a DJ graph in response to a flowgraph edge deletion will be centered at an incremental algorithm for computing the new dominators for all the possibly $DomAffected$ nodes. In the following, we will explain how we transform the *exhaustive* Purdom-Moore algorithm into an *incremental* algorithm that will be much more efficient in practice. For reference we have given the the original Purdom and Moore's exhaustive algorithm for computing the dominator set. The version of the algorithm is adapted from [ASU86], where the subscript *pm* indicates that the set $Dom()$ is computed using Purdom and Moore's algorithm.

Algorithm 8.2 (Finding Dominators.) *The following is an iterative algorithm for computing the dominator set.*

158: $Dom_{pm}(\text{START}) = \{\text{START}\}$
159: **foreach** $n \in N - \{\text{START}\}$ **do**
160: $\quad Dom_{pm}(n) = N$
`/* end initialization */`
161: **while** *Changes* to any $Dom_{pm}(n)$ occur **do**
162: $\quad$ **foreach** $n \in N - \{\text{START}\}$ **do**
163: $\quad\quad Dom_{pm}(n) = \{n\} \cup \cap_{p \in Pred(n)} Dom_{pm}(p)$

[1] The set $\text{DomAffected}_D(y_d)$ can possibly be made more precise by not including any node v_d $idom(y_f) \rightarrow v_f$ is a flowgraph edge, since v_d can never be pulled down. For simplicity, we use only the two given conditions to determine if a node is in $\text{DomAffected}_D(y_d)$.

Let $Dom(v)$ denote the set of dominators for node v. Our modifications to the exhaustive Purdom-Moore algorithm are the following: First, $z_d = idom(y_d)$ will continue to dominate every possibly $DomAffected$ node in $\text{DomAffected}_D(y_d)$ after the update, although it may no longer be its immediate dominator. We, therefore, can focus on the nodes in $SubTree(z_d)$ while ignoring others. Consequently, we will compute the $Dom()$ sets for all the possibly $DomAffected$ nodes as if no other nodes existed outside of $SubTree(z_d)$. Second, we will monitor the $Changes$ condition (step 161 in Algorithm 8.2) only for the possibly $DomAffected$ nodes. Third, whenever $Dom(w_d)$ changes for a node $w_d \in \text{DomAffected}_D(y_d)$, the $Dom()$ set also changes for every descendant v_d of w_d (this is true in Algorithm 8.2). Therefore, we must also observe the $Changes$ condition for any descendant of every possibly $DomAffected$ node. But, fortunately, $Dom(v_d)$ can be "partially deduced" from $Dom(w_d)$. But before explaining this we need to introduce a few concepts.

For our algorithm design, we will first partition all the nodes in $SubTree(z_d)$ into three classes:

1. **PossiblyAffected:** The set of nodes in this class is the same as $\text{DomAffected}_D(y_d)$.

2. **PseudoAffected:** This set consists of any *proper* descendant of a **PossiblyAffected** node. Note that a **PseudoAffected** node's immediate dominator does not change, but its dominators may change.

3. **NotAffected:** This set is defined to capture all the nodes not in the first two classes. Neither does a **NotAffected** node's immediate dominator change, nor do its dominators.

Example 8.2

Consider deleting the edge $2 \rightarrow 4$ from the flowgraph in Figure 8.2(a). $IDF(4) = \{3, 6, 8\}$. By examining the DJ graph in Figure 8.2(b), we can see that (1) nodes 3, 4, 6 and 8 are **PossiblyAffected** nodes; (2) nodes 5 and 7 are **PseudoAffected** nodes; and (3) nodes 1 and 2 are **NotAffected** nodes.

Next we partition $Dom(v_d)$ for each node $v_d \in SubTree(idom(y_d))$ into two parts: the static part $Dom_{st}(v_d)$ and the dynamic part $Dom_{dy}(v_d)$. The static part

captures nodes that will continue to dominate v_d after the update. For example, consider the DJ graph shown in Figure 8.2(b). Nodes 1 and 2 will dominate node 2 even after the edge deletion, so $Dom_{st}(2) = \{1, 2\}$. By contrast, the dynamic part of a $Dom()$ set normally will change during the fixed point iteration. We are interested in $Dom_{dy}()$ for only the **PossiblyAffected** nodes because we want to compute their immediate dominators. Therefore, we carefully initialize and manipulate the $Dom()$ set in such a way that we update the $Dom()$ set only for the **PossiblyAffected** nodes.

Let $z_d = idom(y_d)$. The first step of our approach is to initialize the $Dom()$ set for each node according to its class as follows.

$v_d \in$ **PossiblyAffected.** For each **PossiblyAffected** node v_d, $Dom_{st}(v_d) = \emptyset$ and $Dom_{dy}(v_d)$ = all the nodes in $SubTree(z_d)$. We basically need to recompute the dominators for each **PossiblyAffected** node. For instance, consider the DJ graph in Figure 8.2(b). Node 4 is in **PossiblyAffected** and $Dom_{dy}(4) = \{1, 2, 3, 4, 5, 6, 7, 8\}$.

$v_d \in$ **PseudoAffected.** Recall that the immediate dominator of a **PseudoAffected** node will not change, but its $Dom()$ set may. Let v_d be a **PseudoAffected** node. By definition, there will be a unique node $w_d \in$ **PossiblyAffected** that strictly dominates v_d. Here we will discuss how to initialize the static part and compute the dynamic part of $Dom(v_d)$.

Let S be the set of nodes on the dominator tree path from w_d to v_d (excluding w_d). All the nodes in S will still dominate v_d even after the update, so they are used to initialize $Dom_{st}(v_d)$. For example, consider the DJ graph in Figure 8.2(b) again. The set S for 5 is $\{5\}$, so $Dom_{st}(5) = \{5\}$. (Intuitively, even node 3 can be included in $Dom_{st}(5)$. We did not do that since it will be included in the dynamic part). Now for the dynamic part, we can see that the nodes in $Dom()$ that is not accounted for by the static part are all the nodes dominating w_d. This implies that $Dom_{dy}(v_d)$ is the same as $Dom(w_d)$. Returning to our example; $Dom_{dy}(5) = Dom(3) = Dom_{dy}(3) = \{1, 2, 3, 4, 5, 6, 7, 8\}$. This is because, node 3 immediately dominates node 5, and since node 3 is **PossiblyAffected**, but not 5, we include all the elements of $Dom(3)$ in $D_{dy}(5)$.

$v_d \in$ **NotAffected.** For a NotAffected node v_d, neither its $Dom()$ set nor its immediate dominator will change. Therefore, we initialize $Dom_{st}(v_d)$ to be all the nodes dominating v_d, and make its dynamic part empty. Again consider the DJ graph in Figure 8.2(b). Node 2 is a **NotAffected** node, and $Dom_{st}(2) = \{1, 2\}$.

Given the above initialization we next give the complete algorithm for updating the DJ graph when $x_f \rightarrow y_f$ is deleted from the flowgraph (Algorithm 8.3). When Algorithm 8.3 terminates, the dominators for each **PossiblyAffected** node w_d will be in $Dom(w_d)$. Using this information, we can easily determine the immediate dominators for all the possibly $DomAffected$ nodes. After this we can update the DJ graph accordingly. The complete algorithm is given below.

Algorithm 8.3 *The following algorithm updates the DJ graph of a flowgraph when an existing edge is deleted.*

UpdateDomDel(x, y)
{
164: DomAffected$_D(y_d) = \{w_d | w_d \in (\{y_d\} \cup IDF(y_d))$ **and**
$w_d.level = y_d.level\}$
165: **if**$(x_d! = idom(y_d))$ `/* do not cut if` $x_d \rightarrow y_d$ `is a D edge! */`
166: $cut(x_d, y_d)$
167: $z_d = idom(y_d)$
168: Partition all the nodes in $SubTree(z_d)$ into **PossiblyAffected, PseudoAffected,** and **NotAffected.**
169: Initialize each node $w_d \in SubTree(z_d)$ as described in the main text.
170: $Change = True$
171: **while**$(Change == True)$ **do**
172: $Change = False$
173: **foreach** $w_d \in$ **PossiblyAffected do**
174: $DomTemp = SubTree(z_d)$ `/* temporary variable */`
175: **foreach** $p_f \in Pred(w_f)$ **do**
`/*` p_f `is an immediate predecessor in the flowgraph */`
176: **if**$(p_d \in$ **PseudoAffected**$)$
177: $Dom_{dy}(p_d) = Dom(u_d)$, where $u_d \in$ **PossiblyAffected**
and u_d **stdom** p_d

178: endif
179: $DomTemp = DomTemp \cap (Dom_{st}(p_d) \cup Dom_{dy}(p_d))$
180: endfor
181: $NewDom(w_d) = \{w_d\} \cup DomTemp$
182: endfor
183: if $Dom(w_d) \neq NewDom(w_d)$ then
184: $Change = True$
185: endif
186: $Dom(w_d) = NewDom(w_d)$
187: endwhile
188: Compute the immediate dominators for all the possibly $DomAffected$ nodes.
189: Update the DJ graph accordingly.

}

In the above algorithm, after the completion of step 175 foreach loop, the variable $DomTemp$ contains the intersection of the $Dom()$ sets from a node's immediate predecessors. There are two key points to note in this algorithm: (1) We check the $Changes$ condition only for the nodes in **PossiblyAffected**; and (2) If an immediate predecessor p_d at step 176 belongs to **PseudoAffected**, we make $Dom_{dy}(p_d)$ be $Dom(u_d)$, where $u_d \in$ **PossiblyAffected** and u_d stdom p_d. We, therefore, do not explicitly recompute the dynamic part of $Dom(p_d)$.

Example 8.3

Consider the flowgraph and its DJ graph shown in Figure 8.2. Assume that $2 \rightarrow 4$ is to be deleted from the flowgraph. To update the DJ graph, we first incrementally compute the $Dom()$ set for every possibly $DomAffected$ node. Since $\text{DomAffected}_D(4) = \{4,3,6,8\}$, we have **PossiblyAffected** $= \{3,4,6,8\}$, **PseudoAffected** $= \{5,7\}$, and **NotAffected** $= \{1,2\}$. Also $idom(4) = 1$. The initial values of the $Dom()$ sets shown in Table 8.1.

Once $Dom()$ sets have been initialized we next perform fixed-point computation as in the exhaustive Prudom-Moore algorithm. At step 169 we initialize, for each node, the static and the dynamic

$Node$	$Node\ Type$	$Dom_{st}()$	$Dom_{dy}()$
3	**PossiblyAffected**	$\emptyset$	F
4	**PossiblyAffected**	$\emptyset$	F
6	**PossiblyAffected**	$\emptyset$	F
8	**PossiblyAffected**	$\emptyset$	F
5	**PseudoAffected**	$\{5\}$	$Dom(3)$
7	**PseudoAffected**	$\{7\}$	$Dom(6)$
1	**NotAffected**	$\{1\}$	$\emptyset$
2	**NotAffected**	$\{1,2\}$	$\emptyset$

(a)

$Node$	$Pred_f$	$DomTemp = \cap_{p \in Pred_f} Dom(p)$	$Dom_{dy}() = DomTemp \cup Node$
3	$\{1,7\}$	$\{1\}$	$\{3,1\}$
4	$\{3\}$	$\{1,3\}$	$\{1,3,4\}$
6	$\{4,5\}$	$\{1,3\}$	$\{1,3,6\}$
8	$\{3,7\}$	$\{1,3\}$	$\{1,3,8\}$

(b)

Table 8.1: (a) Initial and (b) Final values of $Dom()$ for the example. (In the above tables $F = \{1,2,3,4,5,6,7,8\}$)

part of $Dom()$. These initializations are shown in Table 8.1(a). Next we iterate and compute the fixed-point of the $Dom()$ set for each node in **PossiblyAffected**step [171]. For our example, the set **PossiblyAffected** = $\{3,4,6,8\}$. For each node w_f in **PossiblyAffected** we first compute the intersection of all the $Dom()$ set of its predecessor node (steps [175] to [180]). At step [176] we check if a predecessor node p_d of a node in PossiblyAffected is in **PseudoAffected**. If so, we make the dynamic part of its $Dom()$ set to be same as $Dom(u_d)$, where $u_d \in$ **PossiblyAffected** and u_d stdom p_d. For example, consider node 3 in PossiblyAffected. The predecessors of node 3 are 1 and 7. Node 1 is in **NotAffected**and so $Dom(1) = \{1\}$. But 7 is in **PseudoAffected**. At step [179] we make the dynamic part of node 7 to be same as $Dom(6)$, which is $\{1,2,3,4,5,6,7,8\}$. Then we compute the intersection of $Dom(1)$ and $Dom(7)$. Once we compute the intersection we then union the set with $\{3\}$ (step [181]). We continue this process until a fixed-point is reached. At step [184] we check to see if a fixed point is reached otherwise we set $Change$ to $True$ and continue the iteration. The final fixed point values for the possibly $DomAffected$ nodes is shown in Table 8.1(b).

Observe that when computing $DomTemp$ at step [179] we always select the latest value of $Dom(p_d)$. In addition, whenever such a p_d is a **PseudoAffected** node, we update its $Dom_{dy}()$ set at step [176]. In the actual implementation, we do not need to update $Dom_{dy}(p_d)$ for $p_d \in$ **PseudoAffected**. Using pointer data structures we can easily point to the set $Dom(u_d)$, where u_d is an ancestor of p_d and $u_d \in$ **PossiblyAffected**.

Once we have computed the $Dom()$ sets, we can easily determine the new immediate dominators for all the $DomAffected$ nodes. In our example, node 1 is the new immediate dominator of 3, which is the new immediate dominator of 4, 6 and 8. With this information we can proceed to update the DJ graph. The updated DJ graph is shown in Figure 8.2.

One key point to observe in the above algorithm is how we initialize the starting solutions for different types of nodes (i.e., the $Dom()$ sets). If we were to start the iteration from the old fixed point (i.e., $Dom()$ initialized to the old the dominators) we would get wrong result (this is because, during fixed-point iteration we

can never increase the size of a $Dom()$). For a comprehensive treatment on this and other related problems of incremental iteration, please see [RMP88, Mar89].

8.3.2 Correctness and Complexity

In order to find the new *immediate dominators* for all the possibly $DomAffected$ nodes, we use Algorithm 8.3 to find their new *dominators* instead. Consequently, the correctness of our approach to handling an edge deletion relies on the correctness of finding the dominators for every PossiblyAffected node. The following lemma claims that when Algorithm 8.3 terminates, the dominators for every PossiblyAffected node are correctly found.

Lemma 8.5 *Let $x_f \rightarrow y_f$ be the deleted flowgraph edge. Then when the* Algorithm 8.3 *terminates, $Dom(w_d)$ contains exactly all the dominators that are in $SubTree(idom(y_d))$ for any* **PossiblyAffected** *node w_d.*

Proof:

First of all notice that the set $NewDom(w_d)$ computed at step 181 is always a subset of the current $Dom(w_d)$ (this because of the intersection operation at step 179). Since $Dom(w_d)$ cannot get smaller indefinitely, we must eventually terminate the **while** loop. Next we will show that, after convergence, the set $Dom(w_d)$ contains all the dominators of w_d that are in $SubTree(idom(y_d))$. For this we will have to show that if a node $u_d \in SubTree(idom(y_d))$ is in $Dom_{pm}(w_d)$ (i.e. dominates w_d) then u_d will be included in $Dom(w_d)$ and vice versa.

Let $p \in Pred_f(w)$. We know that p is either in **PossiblyAffected**, **PseudoAffected**, or **NotAffected**. We will show that for each category that p may belong to, its $Dom(p)$ set will be either a correct estimate or an overestimate of the actual (final) $Dom(p)$

1. $p \in$ **NotAffected**. It is obvious that $Dom(p)$ is a correct one.
2. $p \in$ **PseudoAffected**. Here $Dom(p) = Dom_{st}(p) \cup Dom_{dy}(p)$. The nodes in the static part will always dominate p. At step 177 we assign the dynamic part to $Dom(u)$, where is an ancestor node of

p and $u \in Aff$. The union of the two will be an overestimate. In other words, if a node s dominates p it will be in $Dom(p)$.

3. $p \in$ PossiblyAffected. The $Dom()$ will be an overestimate.

From above we see that if a node u_d dominates p it will be in the set $Dom(p)$. Now if s dominates all the predecessors p of w it will be in the $Dom(p)$ set of all the predecessors. Therefore u_d will also be in the intersection of the $Dom(p)$ set of all the predecessors, and so will be included in $Dom(w_d)$.

Now let u_d be some node in $Dom(w_d)$ when the algorithm terminates. We will next show that u_d dominates w_d. If $u_d = w_d$, we are done. Otherwise, the only way u_d was included in $Dom(w_d)$ is because it was included in the dominator set of all the predecessor nodes p in $Pred_f(w)$ in some previous iteration. This is possible only if u_d dominates all the predecessor nodes p. If u_d dominates all the predecessors, it must dominate w_d. Hence the result.

∎

In the worst case, the time complexity of Algorithm 8.3 can be the same as the Purdom-Moore algorithm. In practice, however, we expect our algorithm to be much faster in the average case.

8.3.3 Other Cases

Let $x_f \rightarrow y_f$ be the edge to be deleted from the flowgraph. Here we will describe how to handle other cases where the reachability of x_f and y_f from START$_f$ is different from what we have assumed. If x is not reachable, then we do nothing because we only intend to maintain the DJ graph for the reachable subgraph REACH(START).

Now assume that x is reachable and y becomes unreachable after deletion. In this case, we remove from the DJ graph all the nodes in $SubTree(y_d)$ and their incident edges. This is because if removing $x_f \rightarrow y_f$ makes y_f become unreachable, then all the nodes strictly dominated by y will not be reachable either. Next, we remove the D edge $x_d \rightarrow y_d$. Finally, we update the reachability status for all the nodes that become unreachable due to the edge deletion.

8.4 Experiments and Empirical Results

In this section we present empirical results for our incremental dominator algorithm. We will first describe our experimental strategy, and then present the results and their analysis.

8.4.1 Experimental Strategy

In [RLP90], Ryder et al. discuss some of the issues involved in experimentally evaluating incremental algorithms. One major problem in accurately evaluating incremental algorithms is selecting suitable test cases [Ram93, RLP90]. For instance, Ryder et al. chose a set of randomly generated flowgraphs, as their test suite, to evaluate their incremental dominator algorithm [RLP90]. They use the following strategy in their evaluation: They first induce random incremental changes (such as insertion and/or deletion of an edge) and measure the time taken to update the dominator tree. To calculate the speedup gained by their algorithm, they then measure the time taken by the exhaustive Purdum-Moore algorithm for computing the dominator information of the *changed* flowgraph. They repeat this process for each incremental change. Using this evaluation strategy, they show that their incremental algorithm performs better than the exhaustive algorithm.

In this chapter we take a different approach for evaluating our incremental algorithm.

- Instead of using randomly generated flowgraphs, we use control flow graphs generated from real FORTRAN programs for our experiments.
- Instead of comparing our incremental algorithm with Purdom and Moore's exhaustive algorithm (whose worst-case time complexity is quadratic), we compare our results with the almost linear time Lengauer-Tarjan (LT) algorithm [LT79].
- Instead of inducing random incremental changes to a flowgraph, we incrementally construct the DJ graph of a flowgraph for real programs. In our evaluation we then compare the time taken to construct (incrementally) the complete DJ graph of the flowgraph with the time taken to compute (exhaustively) the immediate dominator relation of the *final* flowgraph using

the LT algorithm. In other words, if $T(inc)$ is the time take to incrementally construct the DJ graph and $T(LT)$ is the time taken to compute the immediate dominators of the final flowgraph, then the speedup in our case is $\frac{T(LT)}{T(inc)}$.

In our experiments we handle only edge insertions.[2] We incrementally construct the DJ graph in the depth-first order of the nodes in the flowgraph.[3] To evaluate our algorithm, we performed the following measurements:

- For each algorithm we maintain a count of the number of times we visit nodes and edges in a flowgraph. In other words, for each algorithm, we increment a counter whenever we visit a node or an edge. We will denote the final value of the counter for the LT algorithm as $P(LT)$, and for our algorithm as $P(inc)$.[4]

- We measured the execution time of both algorithms on our test procedures. We will denote the execution time of the LT algorithm as $T(LT)$, and the execution time for incrementally constructing DJ graphs as $T(inc)$.

Given these measurement, we will next present the empirical results and their analysis.

8.4.2 Empirical Results and Their Analysis

Table 8.2 gives a summary of our results. We will first give a summary of the major results of our experiments.

- The value of $P(LT)$ ranges from 322 to 2180, with the average value being 1071. The ratio $\frac{P_{LT}}{|E_f|}$ ranges from 3.26 to 3.65 with the average being 3.43. This ratio suggests that the LT algorithm should, on average, make 3.43 passes over a flowgraph during the computation of the immediate dominance relation.[5]

[2]We did not implement our deletion algorithm to test it.

[3]The depth-first order is only incidental, since the nodes in a flowgraph, in our implementation, are numbered in the depth-first order.

[4]Since in our algorithm edges are inserted in the depth-first order of their source nodes, we did not count the edges visited during the depth-first numbering phase of the LT algorithm.

[5]The values of $P(LT)$ and $P(inc)$ indicated in the table are only approximate values. Actual analysis of the LT algorithm indicate that the algorithm makes approximately four passes over

Name	$\|E_f\|$	$P(LT)$	$P(inc)$	$\frac{P(LT)}{P(inc)}$	$\frac{P(LT)}{\|E_f\|}$	$\frac{P(inc)}{\|E_f\|}$	$T(LT)$	$T(inc)$	$\frac{T(LT)}{T(inc)}$
aerset	460	1590	710	2.24	3.46	1.54	12.2	10.1	1.21
aqset	258	911	468	1.95	3.53	1.81	7.7	6.2	1.24
bjt	187	641	2122	0.30	3.43	11.35	5.7	15.2	0.38
card	216	770	1109	0.69	3.56	5.13	6.2	9.8	0.63
chemset	320	1091	800	1.36	3.41	2.50	9.1	9.0	1.01
chgeqz	248	858	1899	0.45	3.46	7.66	7.1	16.5	0.43
clatrs	308	1022	2290	0.45	3.32	7.44	8.2	17.7	0.46
coef	137	451	504	0.89	3.29	3.68	4.6	5.5	0.84
comlr	91	322	337	0.96	3.54	3.70	4.1	4.2	0.98
dbdsqr	327	1100	1853	0.59	3.36	5.67	9.2	15.2	0.61
dcdcmp	187	682	865	0.79	3.65	4.63	5.7	9.0	0.63
dcop	261	901	1997	0.45	3.45	7.65	7.7	14.7	0.52
dctran	458	1588	4958	0.32	3.47	10.83	11.8	40.8	0.29
deseco	236	850	1168	0.73	3.60	4.95	7.4	10.4	0.71
dgegv	232	770	1776	0.43	3.32	7.66	7.1	14.1	0.50
dgesvd	470	1586	1827	0.87	3.37	3.89	12.1	17.8	0.68
dhgeqz	408	1395	3468	0.40	3.42	8.50	11.4	29.3	0.39
disto	191	622	1403	0.44	3.26	7.35	5.3	10.4	0.51
dlatbs	238	803	1490	0.54	3.37	6.26	7.2	11.7	0.62
dtgevc	459	1555	3148	0.49	3.39	6.86	12.0	26.6	0.45
dtrevc	353	1212	1305	0.93	3.43	3.70	9.3	13.3	0.70
elprnt	227	788	1982	0.40	3.47	8.73	6.7	16.4	0.41
equilset	451	1584	1023	1.55	3.51	2.27	12.9	13.0	0.99
errchk	482	1722	3457	0.50	3.57	7.17	12.0	31.0	0.39
iniset	486	1657	486	3.41	3.41	1.00	12.7	9.5	1.34
init	175	572	245	2.33	3.27	1.40	5.5	3.6	1.53
initgas	263	896	678	1.32	3.41	2.58	7.6	7.8	0.97
jsparse	403	1355	742	1.83	3.36	1.84	11.0	9.6	1.15
modchk	419	1498	1167	1.28	3.58	2.79	11.0	14.1	0.78
moseq2	217	771	2738	0.28	3.55	12.62	7.0	21.0	0.33
mosfet	295	1025	4903	0.21	3.47	16.62	8.5	33.8	0.25
noise	160	547	1143	0.48	3.42	7.14	5.1	8.4	0.61
out	590	1944	1357	1.43	3.29	2.30	15.1	16.5	0.92
reader	235	824	723	1.14	3.51	3.08	7.8	9.7	0.80
readin	611	2180	3513	0.62	3.57	5.75	15.1	28.1	0.54
setupgeo	275	918	718	1.28	3.34	2.61	7.8	8.4	0.93
setuprad	286	938	396	2.37	3.28	1.38	7.8	6.1	1.28
smvgear	310	1056	1415	0.75	3.41	4.56	8.6	13.1	0.66
solveq	289	960	518	1.85	3.32	1.79	7.8	7.4	1.05
twldrv	243	838	1025	0.82	3.45	4.22	7.3	10.1	0.72
Average	312	1071	1593	1.00	3.43	5.31	8.71	14.37	0.74

Table 8.2: Timings and speedups

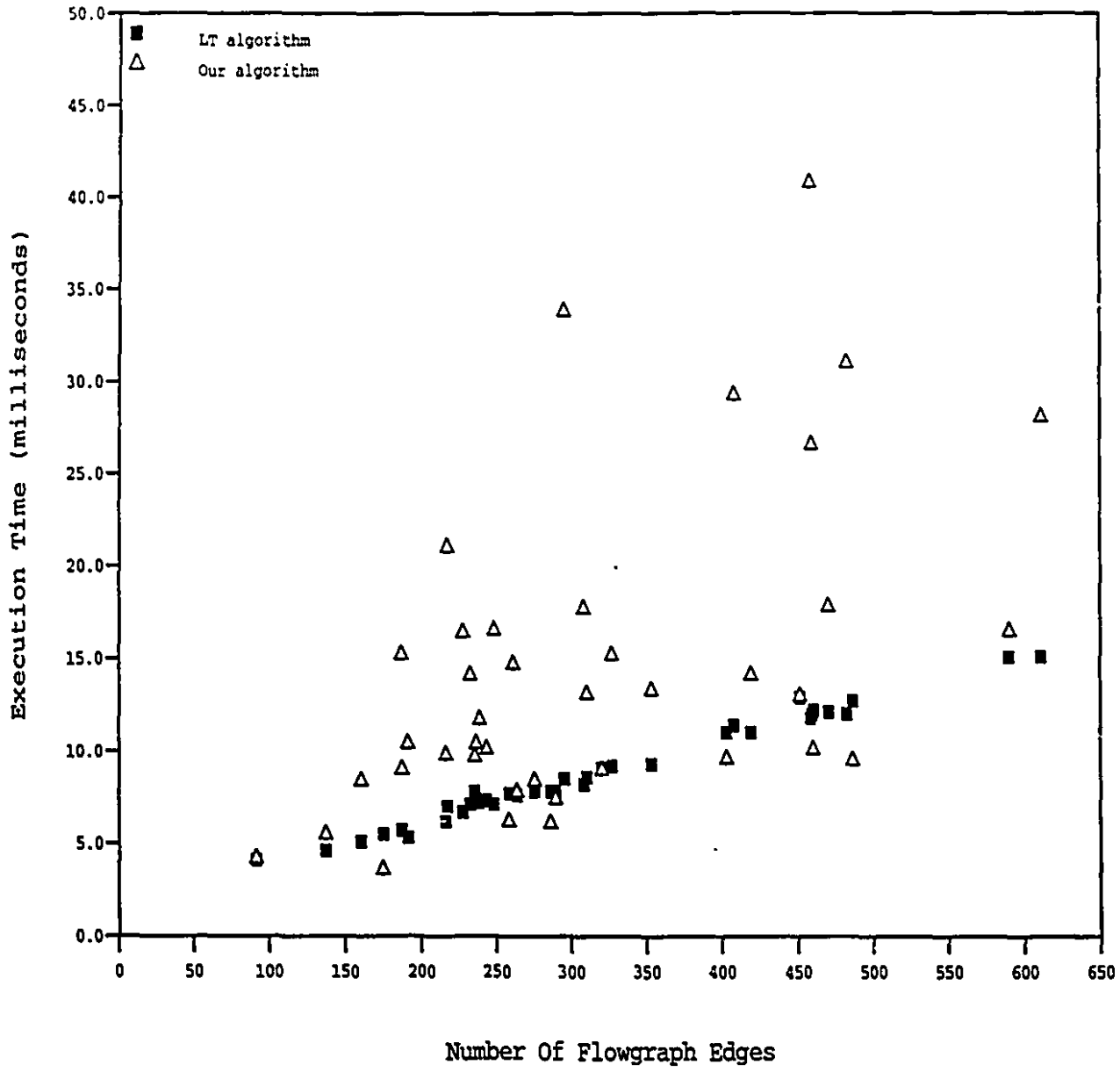


Figure 8.3: Performance of the two algorithms on the test procedures.

The value of $P(inc)$ ranges from 245 to 4903, with the average value being 1593. The ratio $\frac{P(inc)}{|E_f|}$ ranges from 1.00 to 16.62, with the average value being 5.1. This ratio indicates that our algorithm, on average, make 5.31 passes over the entire flowgraph during the incremental construction its DJ graph.

- The value of the $\frac{P(LT)}{P(inc)}$ ranges from 0.21 to 3.41, with the average value being 1.00. The value of this ratio suggests that constructing DJ graphs incrementally, on average, takes about the same time as computing the immediate dominance relation using the LT algorithm. But as can be seen by the data reported in the table constructing DJ graphs incrementally is about 1.35 times slower than computing the immediate dominance relation.

- From the table we can see that the ratio $\frac{T(inc)}{T(LT)}$, ranges from 0.25 to 1.53 with the average value being 0.74. This suggests that constructing DJ graphs incrementally is, on average, about 1.35 times slower than computing the immediate dominator relation using the LT algorithm. It is important to note that the LT algorithm only computes the immediate dominance relation, whereas using our incremental approach we construct complete the DJ graph of flowgraph. In Chapter 3 we gave time measurements for constructing DJ graphs given the immediate dominance relation. If this is included in the timing measurements of the LT algorithm, then the average performance of our algorithm would improve. Figure 8.3 shows the performance graph of the two algorithms on our test procedures. From the plot we can see that the LT behaves linearly, whereas our algorithm exhibits a more complex (or random) pattern.

In our experiments we inserted edges in the depth-first order of the source nodes of the edges. We believe this ordering gives a better result for certain procedures. For instance, consider the procedure `iniset`. This procedure consists of the 154 simple DO loops that initializes arrays. From the table we can see that the ratio $\frac{P(inc)}{|E_f|}$ is 1.00 for this procedure, indicating that we never invoked the procedure for computing $\text{DomAffected}_I()$ during insertion, and so we never had to calculate the IDF set. This is the reason why our algorithm performs better

a flowgraph during the computation of the immediate dominance relation [LT79]. Since $P(LT)$ does not include passes made during depth-first numbering, our measurements confirm to the actual analysis.

than the LT algorithm for this procedure. Remember that the LT algorithm makes four passes over a flowgraph no matter what is the structure of the graph. Since the value of the ratio $\frac{P(inc)}{|E_f|}$ is 1.00, our algorithm constructs the DJ graph for this procedure is (asymptotically) optimal time. In general, we suspect that if the structure of the dominator tree is close to the structure of the depth-first tree, then constructing DJ graphs incrementally would be faster than the LT algorithm.

It is important to emphasize that in our experimental strategy we compute the immediate dominance relation (using the LT algorithm) only once, and compared its performance with the performance of the algorithm for computing DJ graphs incrementally. To be fair we should actually compute the immediate dominance relation after each incremental change, as was done by Ryder et al. [RLP90]. Even in our experimental strategy we can see that our incremental algorithm performs very well, and so we can expect that ours will do a lot better if we use the evaluation strategy of Ryder et al.

As a final remark, the dominating factor in our algorithm is computing the set $\mathsf{DomAffected}_f()$. For our experiments we use the algorithm given in Chapter 7 for computing IDF of a node. It would interesting to see how much improvement we can obtain if we use Cytron et al.'s original algorithm. To use this algorithm we should also incrementally maintain the dominance frontier relation. This is the topic of the next chapter.

8.5 Discussion and Related Work

In this chapter we have presented an approach to the DJ graph update problem, which subsumes the problem of updating dominator trees. Previous work, most relevant to our approach, for the dominator tree update problem includes the Carroll-Ryder algorithm [CR88] and the Ramalingam-Reps algorithm [RR94]. Both these algorithms are restricted to the class of *reducible* flowgraphs. By contrast, our algorithm can handle *irreducible* as well as reducible flowgraphs.

We will first compare ours with Ramalingam and Reps' approach, and then compare with Carroll and Ryder's approach.

8.5.1 Ramalingam and Reps's Approach

To simplify the presentation, we will refer to Ramalingam and Rep's algorithm as the RR algorithm. Unlike the RR algorithm, ours can handle both reducible and irreducible flowgraphs. The RR algorithm is based on the properties of pseudo-circuit value problem [AHR+90], while we use properties of DJ graphs and IDFs to update DJ graphs. They use a modified algorithm of Alpern et al. to update priorities of nodes [AHR+90]. Priorities are equivalent to reverse topological sorting of the nodes in the forward flowgraph.[6] Priorities can be assigned to nodes only if the *forward flowgraph* is a directed acyclic graph. This property is only true for reducible flowgraphs. During the insertion of a new edge, if a cycle is detected in the forward flowgraph, then their algorithm immediately signals that this insertion has introduced irreducibility into the flowgraph. After that they do not allow further insertions and deletions of edges, because priorities cannot be assigned to nodes in the forward flowgraph containing cycles.

In the insertion case, our algorithm can begin with the exact set of affected nodes (e.g. $\mathbf{DomAffected}_I()$), while the RR algorithm needs to begin with a conservative set of possibly affected nodes. When an edge $x_f \rightarrow y_f$ is deleted, Ramalingam and Reps make all the sibling nodes of y_d as affected. Our set of possibly affected nodes, although not exact, is always a subset of y_d's sibling nodes; therefore, it is a better approximation than theirs.

In the worst case, the time complexity of our algorithm for the insertion case is $O(|E|)$, while that of Ramalingam and Reps' algorithm is $O(|E| \times \log|N|)$. Recall that, since we are handling updates of DJ graphs, it is crucial that we update the levels of nodes too. This means that we definitely have to visit all the descendants of the affected nodes to update their level numbers. This will also be true for Ramalingam and Reps' algorithm, if they too update the level information in the dominator tree. Since the two algorithms are fundamentally different, it is difficult to make precise statements on their timing comparisons without substantial tests on real programs. In particular, it is generally very difficult to analytically compare incremental algorithms [Mar89, Ram93]. Therefore, we only make some qualitative observations. Our algorithm, for the insertion case, begins by computing precisely the set of affected nodes, at the expense of visiting

[6] Recall the definition of reducible flowgraphs in Chapter 6 (Definition 6.1). A forward flowgraph is nothing but a reducible flowgraph with all 'back edges' removed.

arguably more nodes compared to the RR algorithm. (This is true even for the deletion case). On the other hand, the RR algorithm has a $log(|N|)$ overhead factor (in the worst case). Also the RR algorithm has to do priority-updating. One way to compare the performance of the two algorithms is to implement and test them on real benchmark programs.

8.5.2 Carroll and Ryder's Approach

The Carroll-Ryder algorithm uses two local properties: *niceness* and *deepness*. Niceness is a property for edges; deepness is a property for nodes. Non-nice edges and non-deep nodes cannot exist in the dominator tree of a reducible graph. Using these two local properties and the notion of *representative edges*, they maintain the dominator tree of a reducible flowgraph. For every edge $x \rightarrow y$ in the flowgraph, and for every node z that dominates x without strictly dominating y, a representative edge $z \rightarrow y$ needs to be maintained. An astute reader can immediately observe that if $x \rightarrow y$ is a representative edge, the $y \in DF(x)$. Rather than using properties of properties of dominance frontiers and iterated dominance frontiers for updating the dominator tree they use "local rotation" operations that moves a subtree up or down one level at a time in the dominator tree. This adds to the complexity of their algorithm, which in the worst case could be quadratic in terms of the number of flowgraph nodes. In contrast we use properties of iterated dominance frontiers for determining the exact set of $DomAffected$ and move the affected nodes in one-shot, once we compute the new immediate dominators.

8.5.3 Other related work

In a recent paper [JPP94], Johnson et al. introduced the Program Tree Structure (PST) for performing fast program analysis. The PST represents a program with a hierarchy of single-entry, single-exit regions. Using the PST, Johnson in his thesis proposed an algorithm for updating dominator trees [Joh94]. His approach is to identify regions, corresponding to sub-flowgraphs, where the dominance relation may no longer be correct because of an update. Once a region is identified, he applies an exhaustive algorithm for all the nodes in this region. In the worst-case there can be only one node in a PST, so that they must exhaustively recompute the dominance relation for all the nodes in the flowgraph. In contrast, we update

the dominance relation only for the (possibly) affected nodes. But again, one should compare the two approaches on real benchmark programs and see their performance.

Chapter 9

Incremental Computation of Dominance Frontiers

> *Take up one idea. Make that one idea your life-think of it, dream of it, live on that idea. Let the brain,muscles,nerves,every part of your body, be full of that idea, and just leave every other idea alone. This is the way to success. If we really want to be blessed, and make others blessed, we must go deeper.*
>
> —Swami Vivekananda

In this chapter we present a simple incremental algorithm for updating the dominance frontier relation of a flowgraph. Dominance frontiers have many applications, including the construction of the SSA form [CSS94] and incremental data flow analysis (Chapter 11). We begin the chapter by introducing and motivating the problem of updating dominance frontiers (Section 9.1). In Section 9.2 and Section 9.3 we present our incremental dominance frontier algorithm for handling edge insertions and edge deletions, respectively. In Section 9.4, we prove the correctness and analyze the complexity of the algorithms. In Section 9.5, we will use the result of this chapter to (potentially) speedup the incremental dominator tree algorithm. Finally, in Section 9.6, we give our conclusion.

9.1 Introduction and Problem Statement

In this chapter we present a simple algorithm for updating the dominance frontier relation of a flowgraph. In Chapter 4 we discussed three ways of representing

dominance frontiers: (1) the full dominance frontier representation, where for each node x, we explicitly represent the set $DF(x)$ as a list; (2) the DJ graph representation, where, for each query, we compute the set $DF(x)$ on-the-fly by walking down the dominator subtree rooted at x; and (3) the $\mathcal{APT}$ representation, where dominance frontiers are cached at certain nodes, called the boundary nodes. The space complexity of the first representation can potentially be quadratic, but querying the dominance frontier of a node is time optimal [CFS90b, PB95]. Using DJ graphs, we can store the dominance frontier relation in linear space, but querying the dominance frontier of a node is not time optimal (Chapter 4). The $\mathcal{APT}$ representation occupies linear space and takes time proportional to size of the set for each query [PB95]. In a pre-processing step they identify boundary nodes where the dominance frontiers are *cached* [PB95]. Comparing $\mathcal{APT}$ with the full representation and the DJ graphs, we can see that every node in the full representation is a boundary node, whereas the source nodes of J edges are the boundary nodes in the DJ graph representation. In $\mathcal{APT}$ boundary nodes are identified in a preprocessing step. Since identifying boundary nodes depends on the input flowgraph (and is sensitive to flowgraph changes), we suspect it may be harder to update $\mathcal{APT}$ than DJ graphs or the full dominance frontier representation. In Chapter 8 we gave a simple algorithm for updating DJ graphs. Once a DJ graph is updated, we can use Algorithm 4.1 given in Chapter 4 to (re-)compute the dominance frontier of any node (Algorithm 4.1). In this chapter we will present an algorithm for maintaining the the full dominance frontier relation. We will use this result in Chapter 11 for updating arbitrary data flow properties.

In the rest of this section we will set the stage for our incremental dominance frontier algorithm. As in Chapter 8, we will allow only two types of updates: (1) insertion of an edge, and (2) deletion of an edge. Recall that the solution procedure for updating the dominator tree of a flowgraph consists of first identifying the set of nodes that are "affected" because of an update. A node is said to be *DomAffected* if its immediate dominator changes because of an update. For both the insertion case and the deletion case, we can determine the exact set of affected nodes when $x \rightarrow y$ is updated, even prior to restructuring the DJ graph. In this chapter we will use the notation DomAffected(y) to represent the (exact) set of *DomAffected* nodes when $x \rightarrow y$ is updated (either inserted or deleted). In

Chapter 8 we showed, when a new edge $x \rightarrow y$ is inserted, the new immediate dominator of all the *DomAffected* nodes is $nca(x, y)$, the nearest common ancestor of x and y on the dominator tree. For the deletion case, a single node does not immediately dominate all the affected nodes. But we can still determine the new immediate dominators of the affected nodes prior to restructuring the DJ graph.

Given the set DomAffected(y) and $nca(x, y)$ we will next show how to incrementally update the dominance frontier of all the "affected" nodes. A node is *PossiblyDFAffected*, if its dominance frontier set possibly changes because of an update in the flowgraph. In the next two sections we give our algorithm for updating the dominance frontier relation. As in Chapter 8, we will initially assume that both x and y are reachable from START; and then we will handle other cases separately.

9.2 Updating Dominance Frontiers: Insertion of an Edge

In this section we give a simple algorithm for updating dominance frontiers of all nodes in DFAffected$_l(y)$ when an edge $x \rightarrow y$ is inserted in the flowgraph. The key question to ask is: at which nodes the dominance frontier may change when $x \rightarrow y$ is inserted into the flowgraph. Recall that a node $w \in$ DomAffected(y) will move up in the dominator tree after the DJ graph is updated. When this happens, the dominance frontier of all the nodes that dominate node w or x, prior to the incremental change, will possibly be affected. Now let $z = nca(x, y)$, and let DFAffected$_l(y) = \{u | z$ **stdom** u **and** u dominates a node $w \in \{x\} \cup$ DomAffected(y) *prior* to updating the DJ graph$\}$. We will claim that if a node is *not* in DFAffected$_l(y)$ then its dominance frontier will *not* change. But if a node is in DFAffected$_l(y)$ we cannot say for sure whether its dominance frontier will change or not. We will formally prove our claim later in Section 9.4. Notice that we can easily compute the set DFAffected$_l(y)$ by a simple bottom-up traversal of the (old) dominator tree starting from nodes in $w \in \{x\} \cup$ DomAffected(y).

Once we determine the set of possibly affected nodes, next we recompute the dominance frontier of these affected nodes. For this we will first update the DJ graph using the algorithm given in Chapter 8. Next, for each node $w \in$

DFAffected$_I(y)$, we will recompute $DF(w)$ in a bottom-up fashion on the new DJ graph as follows: Let w be a node in DFAffected$_I(y)$ and assume that the dominance frontiers of all the children nodes on w are correct. Let $p \in Children(w)$, then the new $DF(w)$ is given by the following formula (see Chapter 4).

$$\begin{aligned} DF_{local}(w) &= \{r | w \rightarrow r \text{ is a J edge}\} \\ DF_{up}(p) &= \{q | q \in DF(p) \text{ and } q.level \leq idom(p).level\} \\ DF(w) &= DF_{local}(w) \cup \bigcup_{p \in Children(w)} DF_{up}(p) \end{aligned} \tag{9.1}$$

Notice that the above formula is exactly the same as given in Chapter 4 for (exhaustively) computing the dominance frontier of all nodes. But, unlike in the exhaustive case, we apply the formula only for the nodes in DFAffected$_I(y)$, the set of affected nodes. The complete algorithm is given below.

UpdateDFIns(x, y)
{
190: Insert $x \rightarrow y$ in the flowgraph
191: $z = nca(x, y)$
192: Compute DomAffected(y) ;
193: Compute DFAffected$_I(y)$;
194: Update the DJ graph ;
195: For each node $w \in$ DFAffected$_I(y)$ in a bottom-up fashion
(ordered by their levels) **do**
196: Compute $DF(w)$ using the formula given in the main text.
197: **endfor**
}

Example 9.1

Consider the flowgraph and its DJ graph shown in Figure 9.1. Let us insert an edge from node 2 to node 5. The resulting flowgraph and the updated DJ graph is shown in Figure 9.2. Using the algorithm given in Chapter 8 the set DomAffected(5) is {5,7}, and $nca(2,5) = 1$. Next we compute the DFAffected$_I$(5). This set consists of all the nodes that dominate the set of nodes in DomAffected(5) (i.e., 5 and 7), and are strictly dominated by $nca(2,5)$ (i.e., 1). We compute the

set $\text{DFAffected}_I(5)$ before updating the dominator tree. To compute the set $\text{DFAffected}_I(5)$ we simply perform a bottom-up traversal of the dominator starting from nodes in DomAffected(5) until we reach $nca(2,5)$. We include all the nodes, except $nca(2,5)$, that are visited during this bottom-up traversal into the set $\text{DFAffected}_I(5)$. By doing this we get $\text{DFAffected}_I(5) = \{2,4,5,7\}$.

The dominance frontiers of the nodes in $\text{DFAffected}_I(5)$, prior to edge insertion are: $DF(2) = \{4, \text{END}\}$, $DF(4) = \{3, \text{END}\}$, $DF(5) = \{7\}$, and $DF(7) = \{3,4, \text{END}\}$. After inserting the edge the new dominance frontiers for these nodes are: $DF(2) = \{4,5, \text{END}\}$, $DF(4) = \{5,7\}$, $DF(5) = \{7\}$, and $DF(7) = \{3,4, \text{END}\}$. Notice that the dominance frontiers for nodes 5 and 7 did not change because of the update, but we still have to recompute its dominance frontiers. In general, it is much more difficult to determine the exact set of nodes whose dominance frontiers will definitely change. Also, since computing the dominance frontier relation using above formula is linear practice it may not be worth the effort to determine the exact set of affected nodes.

Other Cases

Next we will extend the insertion algorithm for cases where both x and y are not reachable. The first case is where x is not reachable, for which we do nothing because we maintain the DJ graph and dominance frontiers only for the reachable sub-graph of the flowgraph. In the second case, y becomes reachable only after the edge insertion. For this we first build, using the exhaustive algorithm, the DJ sub-graph and the corresponding dominance frontier set for the sub-flowgraph induced by nodes reachable from y but not reachable from START. In constructing this DJ sub-graph and dominance frontiers, we treat y as its root. Since x must dominate y, we then insert a D edge from x to y (to connect the newly built DJ sub-graph with the existing DJ graph). Finally, from the viewpoint of updating DJ graph and dominance frontiers, we pretend each edge $u \rightarrow v$ to be a newly inserted flowgraph edge, where u becomes reachable only after $x \rightarrow y$ is inserted, and v is reachable even before the edge insertion. This surprisingly corresponds to the case that we discussed earlier.

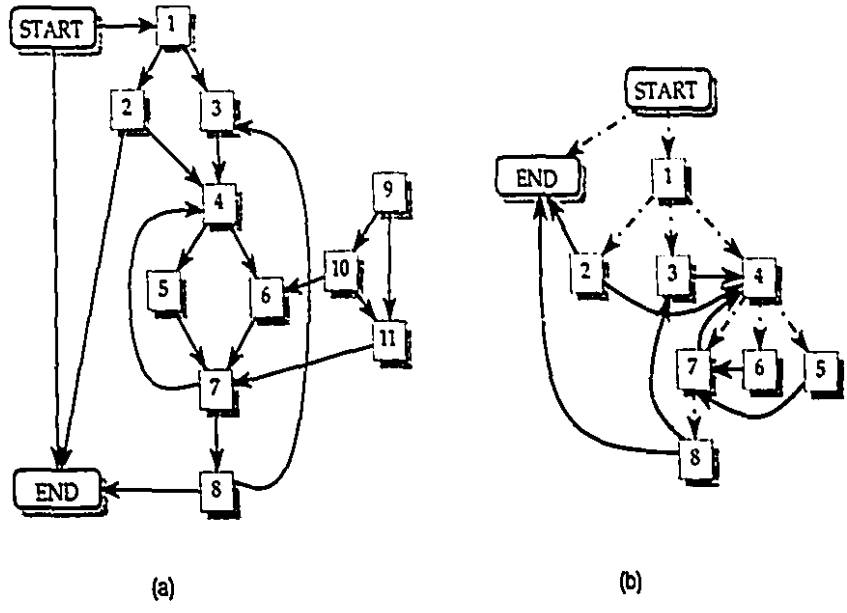


Figure 9.1: An example flowgraph and its DJ graph

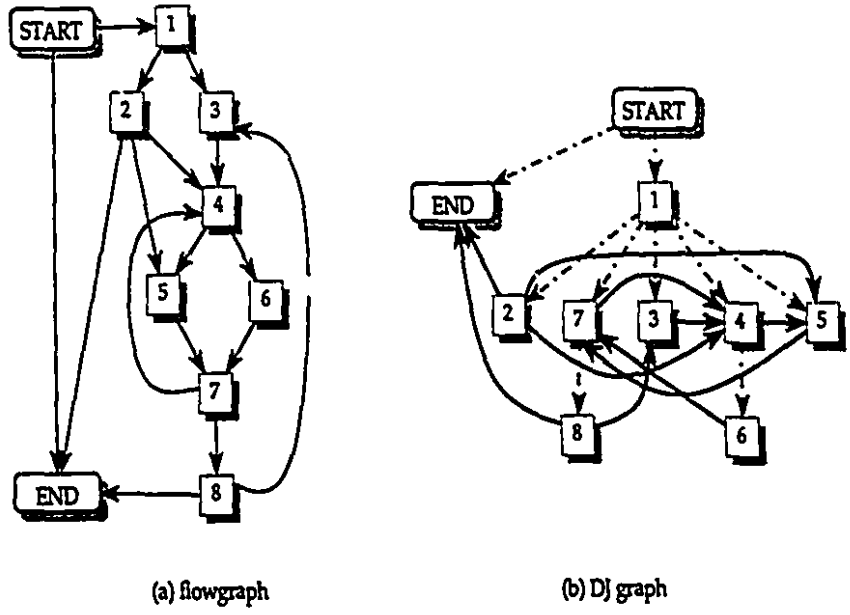


Figure 9.2: The flowgraph and its DJ graph after inserting $2 \rightarrow 5$

9.3 Updating Dominance Frontiers: Deletion of an Edge

In this section, we show how to update dominance frontiers when a flowgraph edge $x \rightarrow y$ is deleted. The effect of deleting an edge is opposite and complementary to that of inserting the same edge. Recall that when inserting an edge $x \rightarrow y$, we pull up every node whose immediate dominator node changes. By contrast, when deleting the edge $x \rightarrow y$, we pull down these affected nodes. So we have recompute the dominance frontiers of all possibly affected nodes. Again let $z = nca(x, y)$, and let $\mathsf{DFAffected}_D(y) = \{u | z \text{ stdom } u \text{ and } u \text{ dominates a node } w \in \{x\} \cup \mathsf{DomAffected}(y) \textit{ after}$ updating the DJ graph$\}$. Now we claim that if a node is not in $\mathsf{DFAffected}_D(y)$ then its dominance frontiers does not change.

In the deletion case we compute the set $\mathsf{DFAffected}_D(y)$ *after* updating the DJ graph. Once the $\mathsf{DFAffected}_D(y)$ set is determined then we update the dominance frontiers of the nodes in $\mathsf{DFAffected}_D(y)$ as in the insertion case. The complete algorithm is given below.

UpdateDFDel(x, y)
{
198: Delete $x \rightarrow y$ in the flowgraph
199: $z = nca(x, y)$
200: Compute DomAffected(y) ;
201: Update the DJ graph ;
202: Compute $\mathsf{DFAffected}_D(y)$;
203: For each node $s \in \mathsf{DFAffected}_D(y)$ in a bottom-up fashion
(ordered by their levels) **do**
204: Compute $DF(s)$ using the formula given in the main text.
205: **endfor**
}

Other Cases

Here we extend the deletion algorithm for cases where both x and y are not reachable. Let $x \rightarrow y$ be the deleted edge in the flowgraph. If x is not reachable, then we do nothing because we only maintain the DJ graph and the dominance

frontier relation for the reachable subgraph. Now assume that x is reachable and y becomes unreachable after deletion. In this case, we remove from the DJ graph all the nodes in $SubTree(y)$ and their incident edges. This is because if removing $x \rightarrow y$ makes y become unreachable, then all the nodes strictly dominated by y will not be reachable either. Next, we remove the D edge $x \rightarrow y$. We update the reachability status for all the nodes that become unreachable due to the edge deletion. Finally, notice that we never need to update the dominance frontiers of nodes that are reachable from the START node. This is because none of the nodes in $SubTree(y)$ will be in the dominance frontiers of any node that is reachable from the START node.

9.4 Correctness and Complexity

In this section we prove the correctness (Theorem 9.1) and analyze the complexity (Theorem 9.2) of both the insertion and the deletion algorithm. In the last two sections we claimed that if a node is not in $\text{DFAffected}_\text{I}(y)$ ($\text{DFAffected}_\text{D}(y)$) then it is not $PossiblyDFAffected$. By possibly affected we mean its dominance frontier may or may not change due to the insertion (deletion) of an edge $x \rightarrow y$. In the following lemma we will show the result for the insertion case, and in Lemma 9.2 we will show the result for the deletion case.

We will first formally define the concept of $PossiblyDFAffected$.

Definition 9.1 *A node is $PossiblyDFAffected$ if its dominance frontier relation possibly changes because of an update in the corresponding flowgraph.*

Given the above definition, the next lemma shows that if a node w is not in $\text{DFAffected}_\text{I}(y)$ then it is not $PossiblyDFAffected$. We will prove the lemma by case analysis.

Lemma 9.1 *If a node w is not in $\text{DFAffected}_\text{I}(y)$ then its dominance frontier does not change due to the insertion of an edge $x \rightarrow y$.*

Proof:

Let $z = nca(x, y)$. It is obvious to see that if w is not in $SubTree(z)$ then none of the nodes in $SubTree(z) - \{z\}$ will be in $DF(w)$, and so

its dominance frontier will not be affected. So we will consider only nodes within $SubTree(z)$.

We can partition the set of nodes in $SubTree(z)$ into three mutually exclusive regions and totally exhaustive regions. The three regions are as follows:

(1) **Region1** consisting of the only node z.

(2) **Region2** consisting of all nodes that dominate some node in $\{x\} \cup$ DomAffected(y), but is strictly dominated by z.

(3) **Region3** consisting of the remaining nodes in the set $SubTree(z)$.

Note that **Region2** is same as the set DFAffected$_I(y)$. In the rest of the lemma we will show that if a node is not either in **Region1** or in **Region3**, then it is not affected.

Case 1: $w \in$ **Region1**. In this case again w will not be affected. This is because w will strictly dominate all the nodes whose immediate dominator change.

Case 2: $w \in$ **Region3** Let $s \in DF(w)$. So there must be a J edge $t \rightarrow s$ such that w **dom** t. Now if $s \notin$ DomAffected(y) then s will not be $DomAffected$ and so s will still be in $DF(w)$. Now if $s \in$ DomAffected(y), then it will move up, and so its level number never increases (follows from Lemma 8.1). Therefore s will still be in $DF(w)$ and so dominance frontier of w is not affected due to s's movement. Therefore w is not in DFAffected$_I(y)$ (follows from Lemma 4.1).

∎

Now consider the deletion case. In the next lemma we show that if a node is affected then it is in DFAffected$_D(y)$.

Lemma 9.2 *If a node w is not in* DFAffected$_D(y)$ *then its dominance frontiers does not change due to the deletion of an edge $x \rightarrow y$.*

Proof:

We can use a similar argument as in Lemma 9.1 to prove this lemma.

∎

Now we are ready to prove the main theorem.

Theorem 9.1 *The procedures* **UpdateDFIns**(x, y) (**UpdateDFDel**(x, y)) *correctly updates the dominance frontier relation when an edge* $x \rightarrow y$ *is inserted (deleted).*

Proof:

Cytron et al. have shown that the Equation (9.1) correctly computes the dominance frontiers for all nodes when the nodes are processed in a bottom-up fashion. Using this result in conjunction with Lemma 9.1 (Lemma 9.2) we can easily see the procedures correctly update the dominance frontiers of all the affected nodes when performed in a bottom-up fashion. ∎

Theorem 9.2 *The worst case time complexity of both the insertion algorithm and the deletion algorithm is* $O(|N|^2)$.

Proof:

We can easily see that the time complexity of computing the sets $\mathsf{DFAffected_I}(y)$ and $\mathsf{DFAffected_D}(y)$ is linear. The time complexity in both algorithms is dominated by the computation of the dominance frontier relation using Equation (9.1), which in the worst case could be quadratic. ∎

9.5 Improving the Efficiency of the Dominator Update Algorithm

In this section we will show how to potentially speed-up the incremental dominator algorithm using the results of this chapter. Recall that one of the key step in updating the DJ graphs is computing the set $\mathsf{DomAffected}(y)$ (assuming that an edge $x \rightarrow y$ is updated). But $\mathsf{DomAffected}(y)$ is a subset of $\{y\} \cup IDF(y)$. Therefore computing $IDF(y)$ is a dominating step in the incremental dominator tree algorithm. If we use the algorithm given in Chapter 7 to compute $IDF(y)$

then we may potentially visit all nodes dominated by the set $IDF(y)$. The time complexity of this method, although linear in terms of the size of DJ graph, may visit more nodes than is needed. In this section we will show how to potentially improve the efficiency of computing $IDF(y)$.

The key idea is to construct a graph called the **DF graph.** A DF graph is nothing but the dominator tree of a flowgraph augmented with edges $u \rightarrow v$, called the **DF edges**, such that $v \in DF(u)$. Notice the relation between DJ graphs and DF graphs. In DF graphs we capture the full dominance frontier relation via DF edges. Figure 9.3 shows the DF graph for Figure 9.1. Now the problem of incrementally updating DF graphs is isomorphic to incrementally updating the dominance frontier relation, so we can use the results of this chapter to update DF graphs.

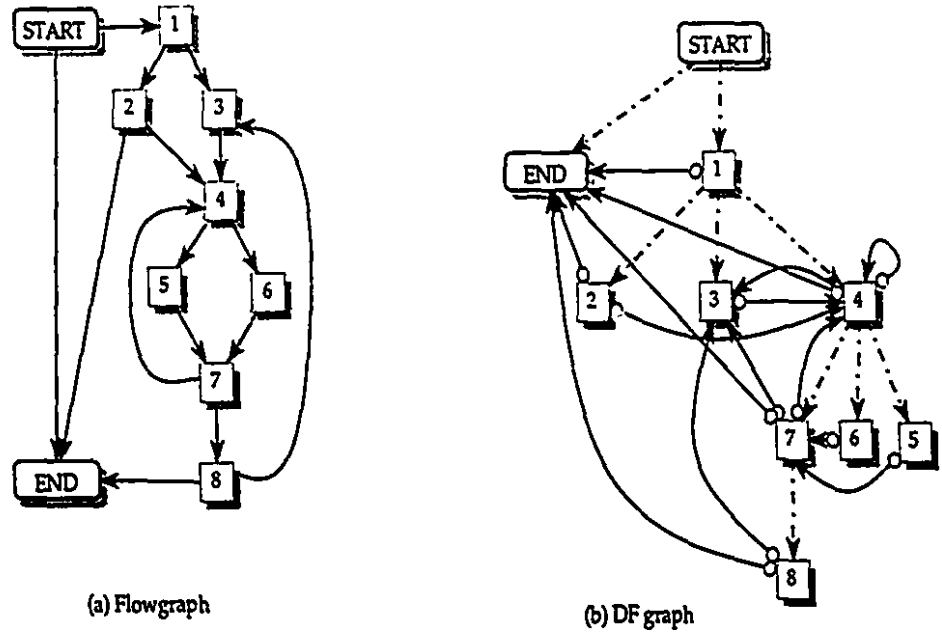


Figure 9.3: The DF graph of the flowgraph shown in Figure 9.2.

Next we will show how to compute $IDF(y)$ very fast. We can compute the set $IDF(y)$ by visiting all the nodes that are reachable from y without visiting any D edges. All such nodes will be in $IDF(y)$. We can use a simple depth first search on DF graph to compute $IDF(y)$, by restricting the search to only DF edges

reachable from y.

Now let us analyze the time complexity of performing both the dominator tree updates and the dominance frontier updates using the above method. The dominating step, for the insertion case, is updating the dominance frontier relation. Since the size of the dominance frontier relation is linear for all practical programs [CFR+91], we expect that updating both the dominance frontier and the domination relation to be faster in practice.

9.6 Discussion

We are not aware of any previous work that gives an incremental dominance frontier algorithm. We expect the result to be useful for work that depends on the dominance frontier relation, like the SSA form and Program Dependence Graphs (PDGs). Compilers that use these representation require that the dominance frontier relation be correctly maintained during the optimization phases. Recently Choi et al. proposed an incremental algorithm for updating the SSA form. Their algorithm cannot handle arbitrary program changes. It would be interesting to extend their algorithm for arbitrary program changes using the results of this section. We will come back to this problem in Chapter 12. Finally, in Chapter 11, we will show how to use the results of this chapter to incrementally update arbitrary monotone data flow information.

Chapter 10

A New Framework for Elimination-Based Data Flow Analysis: Exhaustive Analysis

> *... when you have eliminated the impossible, that which remains, however improbable, must be the truth.*
>
> —Sir Arthur Conan Doyle

In this and the next chapter we introduce a new framework for elimination-based data flow analysis. In this chapter we will focus on exhaustive data flow analysis, and in Chapter 11 we will focus on incremental data flow analysis. For our exhaustive elimination-based data flow analysis we present two variations: (1) **eager elimination method**, and (2) **delayed elimination method**. We begin the chapter by introducing and motivating the problem. Then, in Section 10.2 (and Appendix A), we provide the necessary background material on data flow analysis and also introduce some notation that will be useful for this and the next chapter. Next, in Section 10.3, we outline the foundation of our approach. In Section 10.4, we present our eager elimination method; and in Section 10.5, we prove its correctness and analyze its complexity. In Section 10.6, we present our delay elimination method; and in Section 10.7, we prove its correctness and analyze its complexity. In Section 10.8, we will show how to handle irreducible flowgraphs using our approach. In Section 10.9, we present our empirical results and discuss our observation. Finally, in Section 10.10, we compare our work with other related work, and also give our conclusion.

10.1 Introduction and Motivation

Despite much ground research work that has been done in elimination methods, many researchers and practitioners prefer to use iterative methods for the following two reasons: (1) they are simple and easy to implement, and (2) they can handle arbitrary flowgraphs, including irreducible flowgraphs. Elimination methods, on the other hand, are more efficient than iterative methods, but are more complex to implement. Also, some elimination methods cannot handle irreducible flowgraphs, and even if they do, they are not very efficient.

In this and the next chapter we propose a new framework for data flow analysis based on elimination methods. We will demonstrate that our approach is simple, easy to implement, practically efficient, able to handle irreducible flowgraphs, and amenable to incremental analysis. At the heart of our approach is the DJ graph representation. Within our framework we propose two methods for exhaustive data flow analysis, and one method for incremental data flow analysis. In this chapter we present our approach for exhaustive data flow analysis, and in Chapter 11 we present our approach for incremental data flow analysis.

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 chapter 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,

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

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 trees (for remembering the common substitution sequences), and an $O(|E| \times \alpha(|E|, |N|))$ 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.

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

- We neither use any auxiliary data structure 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 chapter, 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 chapter, 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, which we will discuss in Chapter 11.

10.2 Background and Notation on Data Flow Analysis

In this section we will introduce background material and notation for data flow analysis that are useful for this and the next chapter.

Data flow analysis is a process of estimating facts about a program statically. These facts, or data flow information, can be modeled by elements of a lattice $\mathcal{L}$. Associated with each node x is a flow function f_x that maps input information to output information [Kil73, Mar89].[2]

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

$$O_x = f_x(I_x)$$

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

We can rewrite this equation as follows:

$$O_x = f_x(I_x) = I_x \cap P_x \cup G_x \tag{10.1}$$

where $P_x, G_x \in \mathcal{L}$.[3] We can interpret the above equation as follows: The *O*utput flow information at a node's exit is either (1) what is *G*enerated within the node or (2) what arrives at its *I*nput and is *P*reserved through the node. For convenience, we will use the following notation instead of the notation in Equation (10.1).

$$O_x = f_x(I_x) = P_x I_x + G_x \tag{10.2}$$

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 $Pred_f(x)$; i.e.,

$$I_x = \bigwedge_{y \in Pred_f(x)} O_y \tag{10.3}$$

$\wedge$ is usually a union or intersection operation, depending upon the data flow problem being solved. Combining Equations (10.2) and (10.3), we obtain the following equation, denoted H_x, for each $x \in N$.

$$H_x : O_x = f_x(\bigwedge_{y \in Pred_f(x)} O_y) \tag{10.4}$$

$$H_x : O_x = P_x(\bigwedge_{y \in Pred_f(x)} O_y) + G_x \tag{10.5}$$

Since there is one equation for each node, we have a total of $|N|$ equations. Notice that the above equation has two variations. The first variation (Equation (10.4)) is more general than the second (Equation (10.5)), since for some data flow problems the information generated within a node x is not independent of I_x [Mar89]. We use the term **output variable** to name the variable O_x that appears on the left-hand side (LHS) of equation H_x. Any variable appearing on the right-hand side (RHS) of the equation is called an **input variable**. Furthermore, P_x and G_x are called the **parameters** of the equation.

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

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

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

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

In this example, equation H_x depends only on H_y. This also means that node x has only one incoming edge $y \rightarrow x$, in the corresponding flowgraph. 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:

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

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 $z \in Pred_f(y)$.

For the more general variation in Equation (10.4), variable elimination corresponds to function composition. To illustrate this, let us consider the following equations:

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

After eliminating O_y in H_x we get

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

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

$$H_y : O_y = mO_y + k = f(O_y), \tag{10.6}$$

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

$$H_y : O_y = f^*(O_y) \tag{10.7}$$

where f^* is a closure operator of the recursive equation. Such a closure exists if the flow function associated with each node is monotone, and the lattice does not contain infinite descending chains [Mar89, KU76, Kil73].

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

Example 10.1

Consider the flowgraph shown in Figure 10.1(a) (the corresponding dominator tree and DJ graph is given in Figure 10.1(b) and Figure 10.1(c), respectively. The data flow equation for each node is summarized below.

x	f_x	O_x	$I_x = \wedge_{y \in Pred_f(x)} O_y$	$f_x(I_x) = O_x = P_x I_x + G_x$
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	$(O_4 \wedge O_7 \wedge O_8)$	$P_9(O_4 \wedge O_7 \wedge O_8) + G_9$

10.3 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. Furthermore, we reduce a DJ graph to its dominator tree instead of a single node.

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.
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 in 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 demonstrate in this chaper, 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

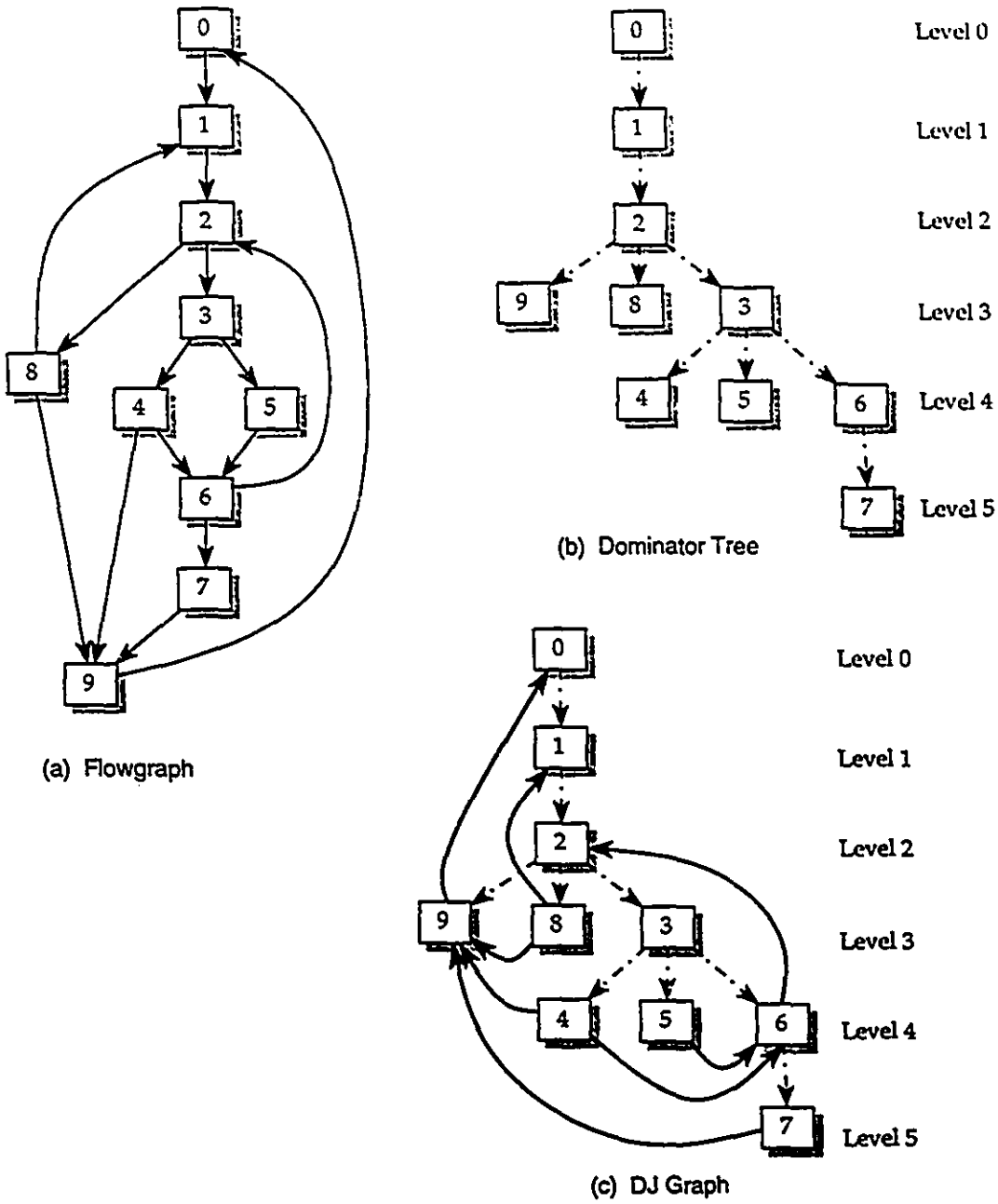


Figure 10.1: Another example of a flowgraph, its dominator, and its DJ graph.

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.

10.4 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 chapter we will adopt the following relaxed definition:

Definition 10.1 (non-join node) *A node y is a* **non-join** *node iff $Pred_J(y)$ 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 10.4.3.

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

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

Definition 10.2 (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 an E2a rule; otherwise we apply an E2b rule.

Definition 10.3 (E2 rules) *Let $\mathcal{G}^i = (N, E)$ be an 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*

$$E2b(< \mathcal{G}^i, N, E, y \xrightarrow{J} z >) = < \mathcal{G}^{i+1}, N, (E - \{y \xrightarrow{J} z\}) \cup \{x \xrightarrow{J} z\} > .^4$$

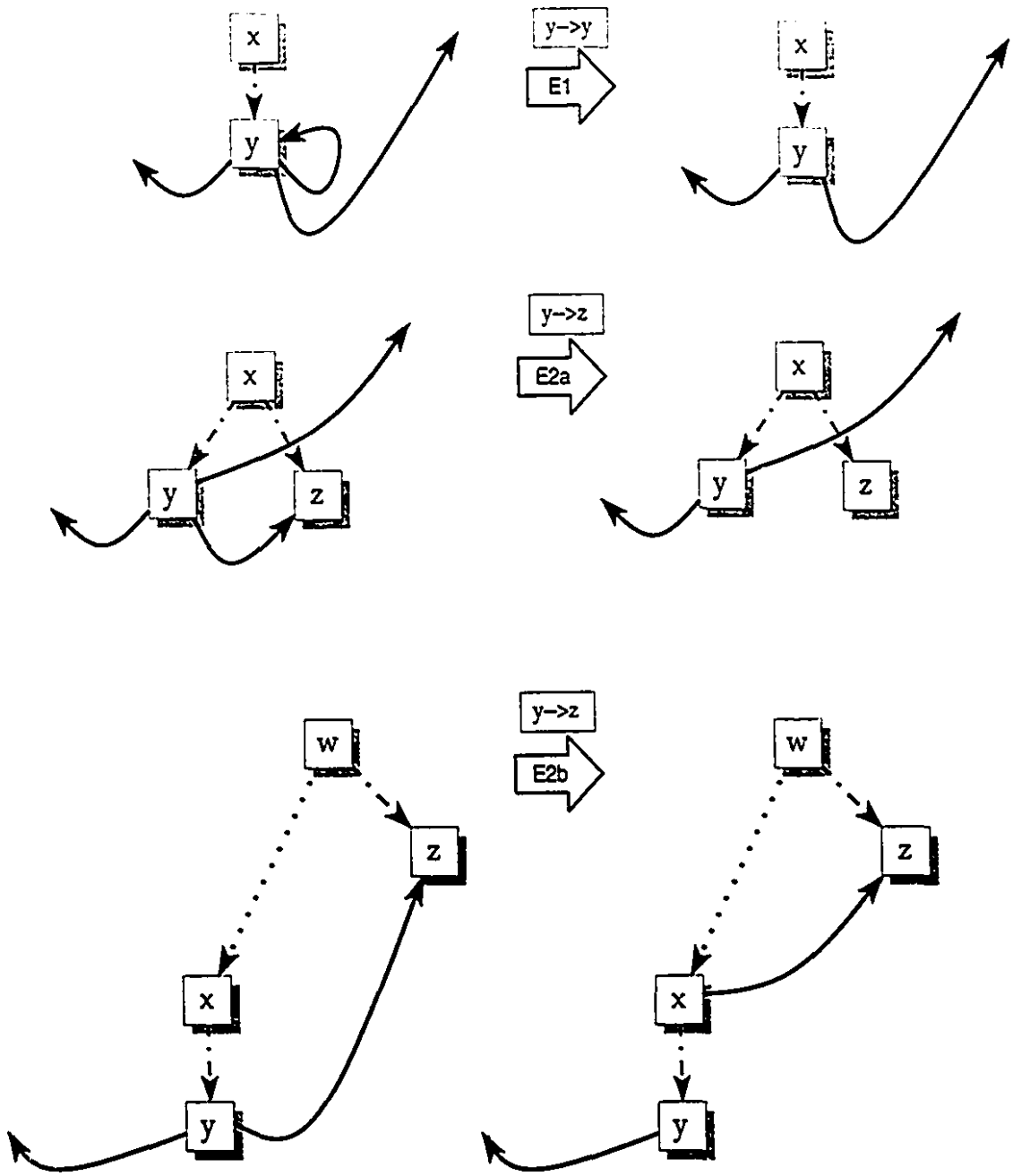


Figure 10.2: A graphical illustration of *E-rules*.

Figure 10.2 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.

10.4.2 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.

- *OrderedBuckets* 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  */
}
```

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

The structure of the *OrderedBuckets* 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 $OrderedBuckets[i]$, at each index i. At all times we ensure that non-join nodes, if any, appear at the head end of the list.

Algorithm 10.1 (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 *OrderedBuckets* at their respective levels. Then we call **ReduceLevel()** in a bottom-up fashion. For reducible flowgraphs, the condition at step 208 will fail. In other words, after the call to **ReduceLevel()** terminates, for reducible flowgraphs, the list $OrderedBuckets[i]$, for the current level i, will be empty (step 208); 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 10.8. The procedure **DomTDPropagate()** is for propagating solutions down the dominator tree, and is explained in Section 10.4.5.

♣ **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 *OrderedBuckets*, such that at each index non-join nodes appear at the head of the list (i.e., before join nodes).

```
MainElim()
{
206:   for i = NumLevel − 1 downto 1 do /* bottom-up reduction */
207:     ReduceLevel(i);
208:     if(OrderedBuckets[i].head ≠ NULL) then
209:       CollapseIrreducible(i);
210:     endif
211:   endfor
212:   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 215). At step 216 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 220 we apply the E2a rule if $idom(y) = idom(z)$; otherwise we apply the E2b rule (step 222). 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)
{
213:   while((y = GetNJNode(i)) ≠ NULL) do
214:     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 */
215:       if(y → z == Jedge) then
216:         if(z == y) then /* self-loop */
217:           Eager1(y → y); /* apply E1 rule */
218:         else
219:           if(idom(y) == idom(z)) then
220:             Eager2a(y → z);
221:           else
222:             Eager2b(y → z);
223:           endif
224:         endif
```

```
225:        endif
226:      endfor
227:    endwhile
}
```

The function GetNJNode(i) returns either a non-join node (if one exists at the current level i) (step 232) or $NULL$ (step 229 or step 234). When GetNJNode(i) returns $NULL$ from step 234, we encounter irreducibility. This is because there are still nodes in $OrderedBuckets[i]$ at level i, but none of them are non-join nodes. In Section 10.5 we will prove why this condition is sufficient to detect irreducibility.

```
Function GetNJNode(i)
{
228:  if(OrderedBuckets[i].head == NULL) then
229:    return NULL ;
230:  endif
231:  if the first node at OrderedBuckets[i].head is non-join node then
232:    Remove that node from the list and return it.
233:  endif
234:    return NULL ;
235:  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 → y)
{
236:  [[ Compute the closure H_y : O_y = f*(O_y) ]]
237:  z.indegree = z.indegree − 1 ;
238:  Delete the edge y --J--> 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 $OrderedBuckets[z.level]$ list (step 243). We do this to ensure constant time operation for the procedure GetNJNode().

Procedure Eager2a($y \rightarrow z$)

```
{
239:  [[Eliminate O_y in H_z by replacing it with the RHS of H_y.]]
240:  Delete the edge y --J--> z ;
241:  z.indegree = z.indegree - 1 ;
242:  if(z is a non-join node ) then /* z becomes a non-join node */
243:    Put z at the head of OrderedBuckets[z.level] list ;
244:  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($y \rightarrow z$)

```
{
245:  [[Eliminate O_y in H_z by replacing it with the RHS of H_y.]]
246:  x = idom(y) ;
247:  Delete the edge y --J--> z ;
248:  if(x -> z exists) then
249:    z.indegree = z.indegree - 1 ;
250:  else
251:    Insert a new J edge x --J--> z ;
        /* If x -> z is a self-loop edge we ensure that this */
        /* is the first edge in the list of edges at node x. */
252:  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 10.5, 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 10.5

To illustrate our reduction method, consider the example flowgraph in Figure 10.1. Figure 10.3 gives a trace of the reduction process. In Section 10.4.4 we will give a detailed explanation of the trace, after we discuss variable elimination.

10.4.3 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

$$H_y : O_y \;=\; kO_y + m, \tag{10.8}$$

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:

$$H_y : O_y \;=\; f^*(O_y). \tag{10.9}$$

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 236.

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:

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

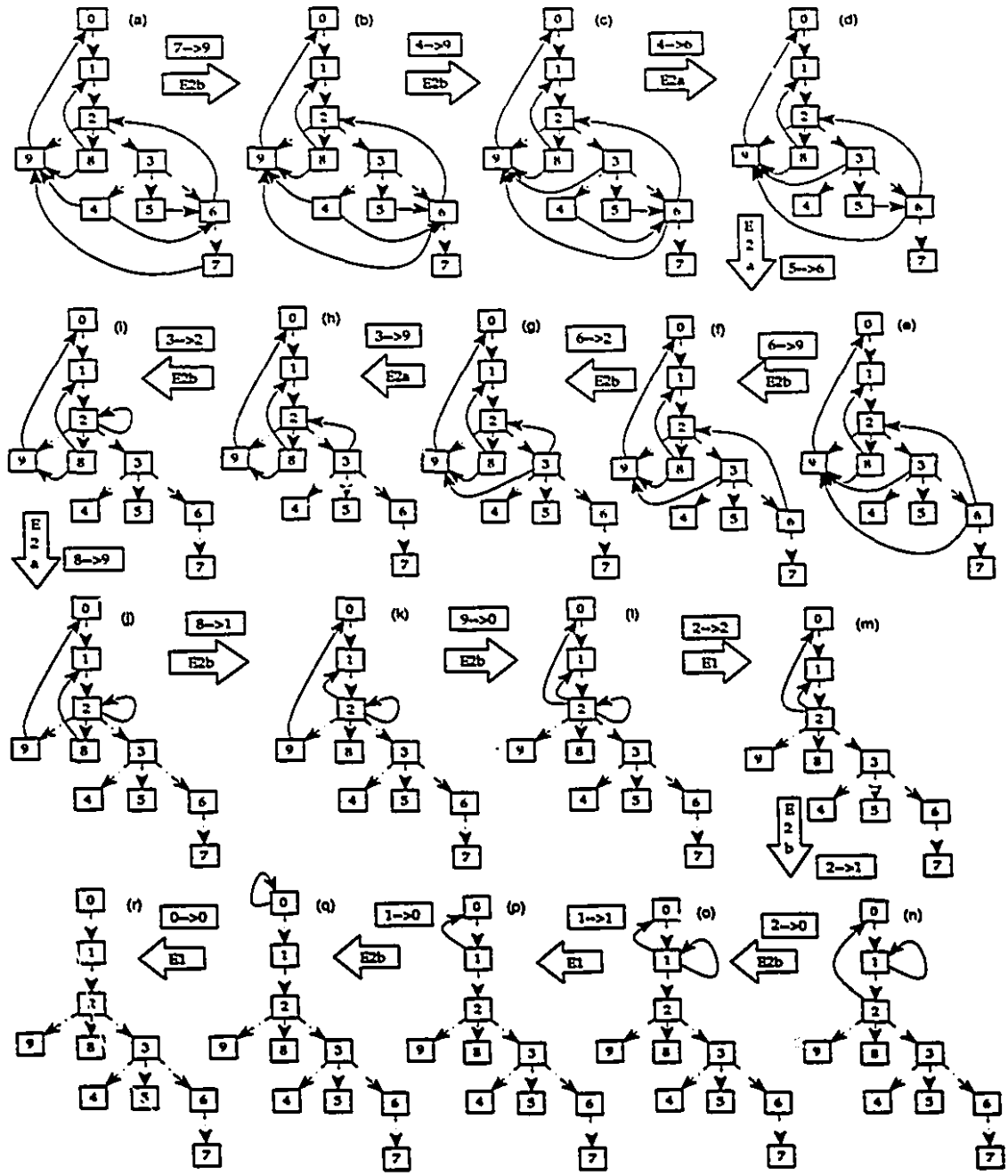


Figure 10.3: A trace of the DJ graph reduction using $\mathcal{E}$-*rules*.

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

$$H_y : O_y = aO_x + b$$

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

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

In the procedure Eager2a($y \rightarrow z$) (Eager2b($y \rightarrow z$)), the variable elimination is done at step 239 (step 245). 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 10.6 we will show how to delay certain variable elimination so that we can speed up the overall algorithm.

10.4.4 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 10.3.

	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 10.3

1. Recall that the DJ graph reduction and variable elimination are performed in a bottom-up manner (the **for** loop at step 206).

2. At step 213 GetNJNode(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 222). This transforms the DJ graph shown in Figure 10.3(a) to the DJ graph shown in Figure 10.3(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 GetNJNode(4) is invoked at step 213, there are two non-join nodes, node 4 and node 5, at level 4. Assume that GetNJNode(4) returns node 4, and we apply E2b rule to the outgoing J edge $4 \rightarrow 9$ by invoking Eager2b($7 \rightarrow 9$) (step 222). This transforms the DJ graph shown in Figure 10.3(b) to the DJ graph shown in Figure 10.3(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 GetNJNode(4) would return node 5, and we invoke $rule a5 \rightarrow 6$. This transforms the DJ graph shown in Figure 10.3(c) to Figure 10.3(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.

10.4.5 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 212 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()

{

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

254: 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.

}

10.5 Correctness and Complexity of Eager Elimination Method

In this section we prove the correctness of Algorithm 10.1 and analyze it time complexity.

10.5.1 Correctness

The main theorem which establishes the correctness of Algorithm 10.1 is Theorem 10.1. 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 10.4); and
- we have to show that variable elimination and top-down propagation are correct (Lemma 10.5).

We first define the reducibility of DJ graphs as well as flowgraphs [HU74]. Recall that we introduced the notion reducibility in Chapter 6 (Definition 6.1). We have reproduced the definition below for convenience.

Definition 10.4 *A DJ graph 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 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 (Chapter 6). 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 10.1. 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 10.1 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 10.2, we will show that the reduction rules preserve the reducibility of a DJ graph. Given this and Lemma 10.1 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 10.3, that when ReduceLevel(i) is called at step 207 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 10.5, that when the procedure **DomTDPropagate()** is called at step 212, the DJ graph has been reduced to its dominator tree. Given this we will also show, in Lemma 10.5, that when the **DomTDPropagate()** is invoked at step 212, 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 10.1 *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 10.4, 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 10.2) shows that applying *$\mathcal{E}$-rules* preserves reducibility of the DJ graph.

Lemma 10.2 *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 10.4 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 10.4, 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 10.4 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 10.4 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 10.1 and 10.2, 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 10.3 *In eager elimination, when* **ReduceLevel**(i) *is called at step* 207 *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

GetNJNode(i) (steps 213 and 214). 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 206

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 214). 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 10.2 we know that $\mathcal{E}$-*rules* preserve reducibility, and from Lemma 10.1 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 10.1 again). Therefore, **GetNJNode**(k) when called at step 213 with the current level k, will return a non-join node. The procedure **GetNJNode**(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 10.4 *When the* Algorithm 10.1 *begins at step* 212, *all the J edges have been eliminated.*

Proof:

From Lemma 10.3 we know that when the call to ReduceLevel(i) at step 207 terminates, all J edges whose source nodes are at level i are eliminated. Also, the **foreach** loop at step 206 calls ReduceLevel(i) in the decreasing order of i. Therefore when the loop terminates, all the J edges have been eliminated. ∎

In Lemma 10.5, we will argue that when the **DomTDPropagate()** is invoked at step 212, 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 10.5 *When* **DomTDPropagate()** *is invoked at step* 212, *the flow equation at each node depends only on the flow variable of its immediate dominator.*

Proof:

The procedure **DomTDPropagate()** is invoked at step 212 only when the call to **ReduceLevel(1)** is terminated. From Lemma 10.3 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 10.1 Algorithm 10.1 *correctly computes the solutions to a set of data flow equations for a reducible flowgraph.*

Proof:

From Lemma 10.4 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 10.5 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. ∎

10.5.2 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 10.6 *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 $O(|E|)$ times.

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 operations.

Theorem 10.2 *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 10.6 and Lemma 10.4. ∎

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 10.6). 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 10.9). Therefore, our eager elimination method can be expected to be linear for most practical programs.

10.5.3 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 the T1 rule, whereas our E2 rule is equivalent to the T2 rule with one difference: Our E2 rule eliminates outgoing edges of a non-join node one at a time, whereas the T2 rule eliminates all the outgoing edges in one shot.

- In this chapter 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 GetNJNode(). During the initialization portion of the algorithm we ensure that non-joins appear before join nodes in $OrdercdBuckets[i]$ at each level. Also, during the bottom-up reduction (at step 243) 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 GetNJNode().

10.6 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. For this we require some knowledge of dominance frontier intervals introduced in Chapter 4. Before presenting our method, we will review the concept of dominance frontier intervals in Section 10.6.1; and also introduce the notion of "derived edges" of a J edge (Section 10.6.2), that will simplify our presentation.

10.6.1 Dominance Frontier Interval Revisited

In Chapter 4 we introduced the concept of dominance frontier intervals. Let $y \rightarrow z$ be a J edge in $\mathcal{G}^0$, and let $w = idom(z)$. By the definition of dominance frontiers (Definition 2.4), 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 (see also Chapter 4). In Chapter 4 we called the path $y \xrightarrow{+} w$ the Dominance Frontier Interval path of the J edge $y \rightarrow z$. In that chapter, we also gave a simple algorithm for computing cTop for all the J edges in linear time. We will use these two concepts in this chapter. (For example, in Figure 10.4, we have shown the cTop for each J edge as $< x >$, for the example DJ graph shown in Figure 10.1.)

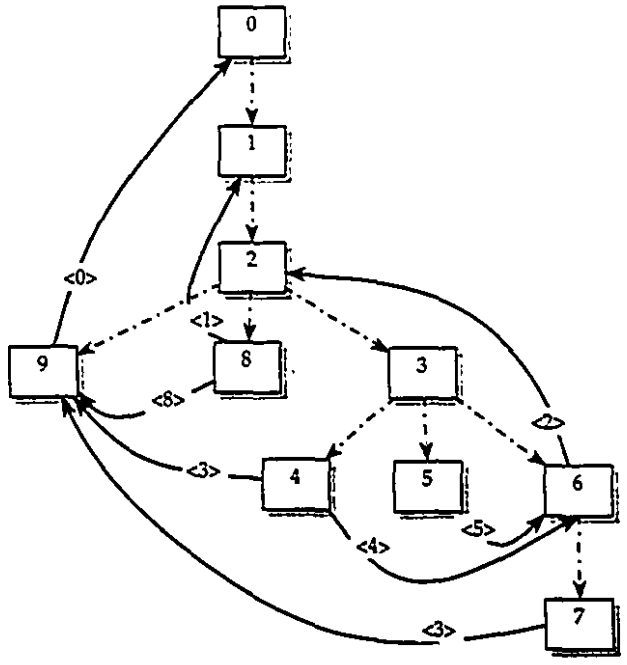


Figure 10.4: The DJ graph of Figure 10.1 annotated with cTop nodes.

10.6.2 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 10.5 (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 10.3. We can see that the edge $6 \rightarrow 9$ in Figure 10.3(b) is a **derived edge** of $7 \rightarrow 9$ in Figure 10.3(a). The edge $2 \rightarrow 1$ in Figure 10.3(k) is again derived from the edge $8 \rightarrow 1$ in Figure 10.3(j).

10.6.3 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.

10.6.4 $\mathcal{E}$-*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 10.6 (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 $x = \mathbf{cTop}_{y-z}$. If $idom(y) \neq idom(z)$ then*

$$D'2b(< \mathcal{G}^i, N, E, y \xrightarrow{J} z, x >) \Longrightarrow < \mathcal{G}^{i+1}, N, (E - \{y \xrightarrow{J} z\}) \cup x \xrightarrow{J} z > .^5$$

We will carry over the definition of E1 and E2a rules to our delayed 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 chapter 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.

10.6.5 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 $x \xrightarrow{+} y$, 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 $x \xrightarrow{+} y$, the children of x.

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

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 $x \xrightarrow{+} y$ 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 $x \xrightarrow{+} y$) will change its parent from w to x.

Procedure CompPath(x, y)

{

255: **foreach** node $w \neq x$ in the path from $x \xrightarrow{+} y$ and in the path order **do**

256: Express O_w as a linear function of O_x.

257: Make w a child of x.

258: **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 10.1 as follows:

```
struct NodeStructure {
    int indegree; /* as defined in Algorithm 10.1 */
    int level; /* as defined in Algorithm 10.1 */
    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 10.1, 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 10.2 *The algorithm* **MainElim()** *with D-rules and delayed elimination can be used for solving a system of data flow equations.*

In Algorithm 10.2 we will use the main procedure of Algorithm 10.1 for reducing the DJ graph in a bottom-up fashion. The only difference between Algorithm 10.1 and Algorithm 10.2 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 $x = \text{cTop}_{y \rightarrow z}$ in the list x->*jedges* (step 276). 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 260 or step 266, respectively). As explained earlier, for each node w on the (compressed) dominator tree path $x \stackrel{+}{\rightarrow} y$, 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 10.6.7.

In the procedure Delayed1($y \rightarrow y$), the function GetJedge(y) returns a J edge $u \rightarrow w$ from the list y->*jedges* (step 259). For each such edge we invoke CompPath(y, u) to perform path compression and delayed variable substitution (step 260). Finally, we compute the closure of the recursive equation H_y to break the loop (step 263). At step 264 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($y \rightarrow y$)

{

259: **while**(($(u \rightarrow w = \text{GetJedge}(y)) \neq NULL$)) **do**
  /* Get the next J edge for which this node is a top. */

260: CompPath(y,u); /* Replace node equations on the path */
  /* $y \stackrel{+}{\rightarrow} u$ as a function of O_y */

261: Eliminate O_u on the RHS of H_y by replacing it
  with a linear function of O_y computed in the previous step.

262: **endwhile**

263: Compute $f^*(O_y)$. /* Compute the closure. */

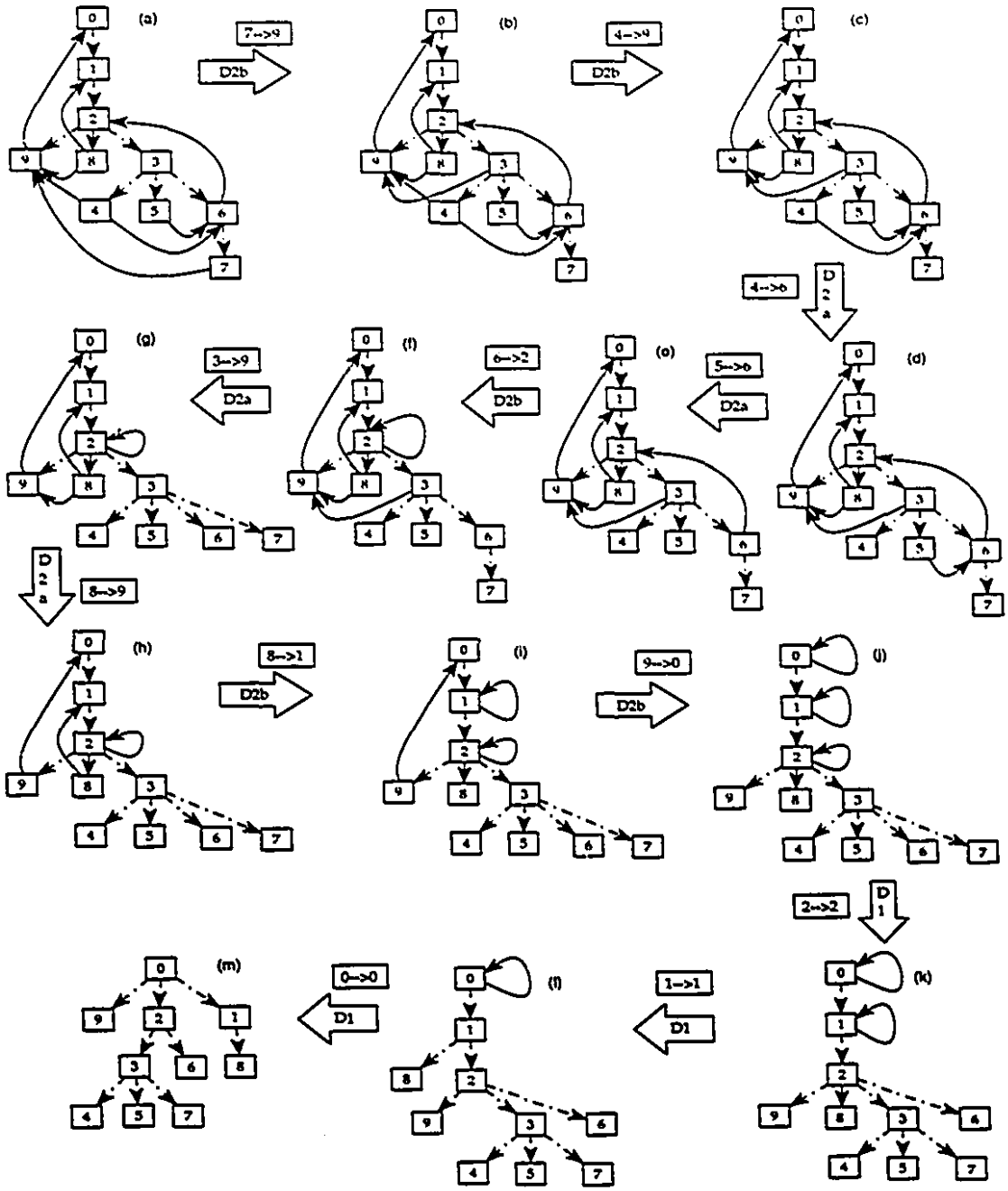


Figure 10.5: A trace of the DJ graph reduction using *D-rules*.

264: 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 267 we replace O_u in equation H_w with the linear function of O_y computed in step 266. After the **while** loop, we replace O_y in H_w by the RHS of H_y (step 269). 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 270 to step 274 are the same as in Eager2a(), and essentially perform a DJ graph reduction.

Procedure Delayed2a($y \rightarrow z$)

{

265: **while**(($(u \rightarrow w = \text{GetJedge}(y)) \neq NULL$)) **do**

/* Get the next J edge for which this node is a top. */

266: CompPath(y,u) ; /* Express the node equations on the path */

/* $y \stackrel{+}{\rightarrow} u$ as a function of O_y */

267: Eliminate O_u on the RHS of H_w by replacing it with the linear function of O_y computed in the previous step.

268: **endwhile**

269: 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.

270: Delete the edge $y \rightarrow z$;

271: $z.indegree = z.indegree - 1$;

272: **if**($z.indegree \leq 1$) **then** /* z becomes a non-join node */

273: Put z at the head of $OrderedBuckets[z.level]$ list ;

274: **endif**

}

In the procedure **Delayed2b**($y \rightarrow z$) we do not eliminate variables. At step 276 we save the J edge $y \rightarrow z$ in x->$jedges$, where $x = \text{cTop}_{y \rightarrow z}$ (step 276). These J edges (returned by the function GetJedge()) will be processed later in

the procedure for D1 and D2a rules at step 259 and step 265, respectively. The operations from step 277 to step 282 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($y \rightarrow z$)

{

275: $x = \mathbf{cTop}_{y-z}$;

276: Insert node $y \rightarrow z$ in the list x ->$jedges$.

277: Delete the edge $y \rightarrow z$;

278: **if**($x \rightarrow z$ exists) **then**

279: $z.indegree = z.indegree - 1$;

280: **else**

281: Insert a new J edge $x \rightarrow z$;

282: **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 10.7.3), 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)

{

283: **if** the list w->$jedges$ is not empty **then**

284: Remove the first edge from the tail of list and return it.

285: **else**

286: **return** $NULL$

287: **endif**

}

10.6.6 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 10.2 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 10.7. To propagate the data flow information on the compressed dominator tree, we invoke **cDomTDPropagate()** at step 212 instead of **DomTDPropagate()**. The procedure **cDomTDPropagate()** is defined below:

cDomTDPropagate()

{

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

289: 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.

}

10.6.7 An Example

Here we illustrate the delayed elimination method for our example DJ graph. The complete DJ graph reduction process is shown in Figure 10.5.

1. For the DJ graph in Figure 10.5(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 10.5(a) to Figure 10.5(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 10.5(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 10.5(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 265). At step 269 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 10.5(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 265). At step 269 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 10.5(e) to Figure 10.5(f). We invoke Delayed2b($6 \rightarrow 2$), in which we store $6 \rightarrow 2$ in 2->$jedges$ (step 276).

5. Next we apply D2a rule to the edge $3 \rightarrow 9$, transforming the DJ graph of Figure 10.5(f) to Figure 10.5(g). We invoke Delayed2a($3 \rightarrow 9$). At this point 3->$jedges$ contains edges $4 \rightarrow 9$ and $7 \rightarrow 9$. So at step 265 the procedure GetJedge(y) returns these two edges, one after another. Let $4 \rightarrow 9$ be the first edge returned, and so at step 266 we invoke CompPath(3,4). In the procedure CompPath() we express the flow equation at every node on the path $3 \stackrel{+}{\rightarrow} 4$, 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 255 we replace O_4 on the RHS of H_9 as a linear function of O_3. The resulting equation of node 9 is

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

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 $3 \stackrel{+}{\rightarrow} 7$, 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 257 we also make node 7 a child of node 3, resulting in the compressed dominator tree shown in Figure 10.5(g).

Next, at step 267 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 267 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.

10.7 Correctness and Complexity of Delayed Elimination Method

In this section we prove the correctness of Algorithm 10.2 and analyze its complexity. We use some of the results of the eager elimination method.

10.7.1 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 10.10), and (2) that variable elimination is properly performed in the bottom-up phase (Lemma 10.11). First note that Lemma 10.1 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 10.7 *Let G be a reducible DJ graph. Let one of $\mathcal{D}$-rules be applied to G, resulting in G^1. Then G^1 is also reducible.*

Proof:

Recall that, in terms of graph reduction, the 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 $x = \text{cTop}_{y \rightarrow z}$. We use the results from Lemma 10.2 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 *D-rules* eliminate all the J edges in a bottom-up fashion, we will first prove a result similar to Lemma 10.3.

Lemma 10.8 *In delayed elimination,*

1. *at the time when* **ReduceLevel**(i) *is invoked at step* 207, *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 206.

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 10.3 and Lemma 10.8. In Lemma 10.8, 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 10.3). 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 10.9). 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 $y = \text{cTop}_{u \rightarrow w}$. 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 206).

Lemma 10.9 *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 206.

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 $y\text{->}jedges$. 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 $y \xrightarrow{+} u$ (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 10.4 and 10.5, respectively.

Lemma 10.10 *When* Algorithm 10.2 *begins at step* 212 *all the J edges have been eliminated.*

Proof:

This lemma follows from (1) Lemma 10.8 and (2) the fact that the call to ReduceLevel(1) has completed before step 212. ∎

Lemma 10.11 *When* **cDomTDPropagate()** *is invoked at step* 212, *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 10.9 since **cDomTDPropagate()** is invoked only after **ReduceLevel(0)** has completed. ∎

Theorem 10.3 Algorithm 10.2 *correctly computes the solutions to a set of data flow equations for a reducible flowgraph.*

Proof:

Follows from Lemmas 10.7, 10.10, and 10.11. ∎

10.7.2 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 10.12 *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 $\mathbf{cTop}_{y\_z}$ 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 10.7 *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 $O(e \times \log(|N|))$. Therefore, if we can show that the sequence of path compressions does satisfy the RCC, it will immediately follow that the total cost of our e path compressions is also $O(e \times \log(|N|))$.

Lemma 10.13 *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 10.4 *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. For most flowgraphs, $|E| = O(|N|)$, so the above bound is tight.

10.7.3 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 10.9 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.

10.8 Handling Irreducibility

In this section we will show how to handle irreducibility in flowgraphs with delayed elimination.[6] 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 10.6(a) gives an example of an irreducible flowgraph, and its DJ graph is given in Figure 10.6(b). In our elimination method we detect irreducibility if during the bottom-up reduction GetNJNode(i) returns $NULL$ but there are more J edges to be processed at this level. At step [208] if $OrderedBuckets[i].head$ 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.

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 [293]). 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 [297]).

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 [301]). 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 [302]). We compute closure of equations of all nodes in S, if needed (step [303]). We also express the flow equations at all nodes at level i in terms of flow variable of their immediate dominator node (step [304]). 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 [305]).

Procedure CollapseIrreducible(i)

```
{ /* See also the description in the main text */
290:   foreach node y at level i do
291:     foreach z ∈ Succ(y) do
292:       if((y → z == Jedge) and (z.level < i))
293:         Delayed2b(y → z).
```

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

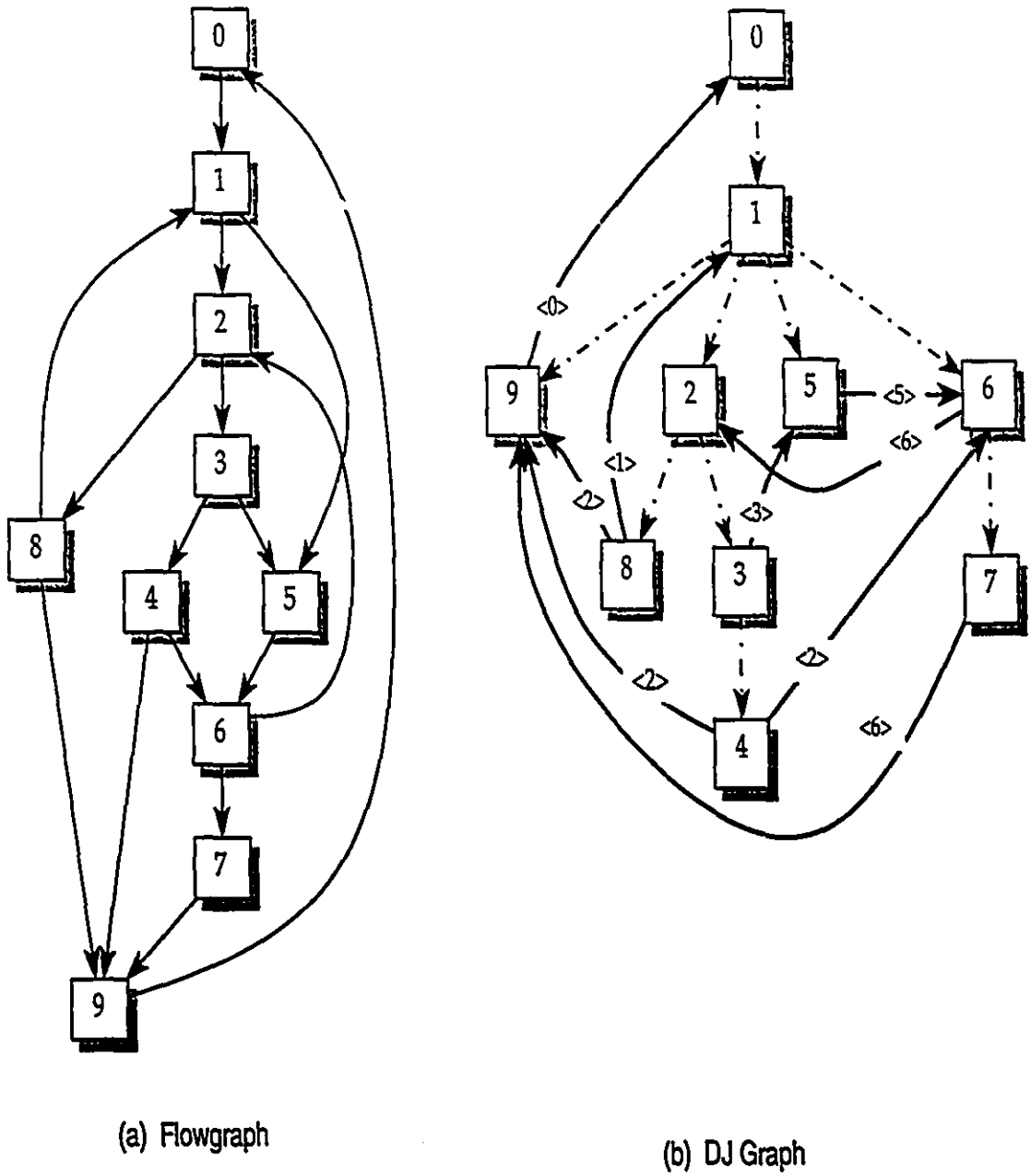


Figure 10.6: An irreducible flowgraph and its DJ graph.

294: endif
295: endfor
296: while(((u → w = GetJedge(y)) ≠ NULL)) do
297: CompPath(y,u) ;
298: 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.
299: endwhile
300: endfor
301: Determine the SCCs of the nodes at level i, and construct dag(s) of SCCs;
302: Process each SCC S in each dag in the topological order as follows:
303: 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.
304: 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.  */
```

305: 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 10.7. During the reduction when we reach to the DJ graph shown in Figure 10.7(g), we can notice that all nodes at level 2 are join nodes, and so we cannot apply any of the *D-rules*. At this point we invoke CollapseIrreducible(2). In the procedure CollapseIrreducible() we first apply D2b rule by invoking Delayed2b(9 → 0). Next we compress the path 2 → 3 → 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 10.7(h).

10.9 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 discussed in Chapter 7 the size of the dominance frontier relation is linear in practice. Using Definition 2.4 for dominance frontiers, we found the average ratio $\frac{|DF_*|}{|E|} = 1.09$. This ratio suggests that the average size of dominance frontiers (represented as a set of edges) is about $\frac{1.09-1}{1.09} = 8.23\%$ more than the size of the flowgraph. This confirms to the claim made by Cytron et al. that size of dominance frontiers is proportional to the size of the flowgraph.

- As expected, the number of $\mathcal{E}$-*rules* rules applied is bounded by the size of the dominance frontier relation. We found the avarage ratio $\frac{nE}{|DF_*|}$ of the

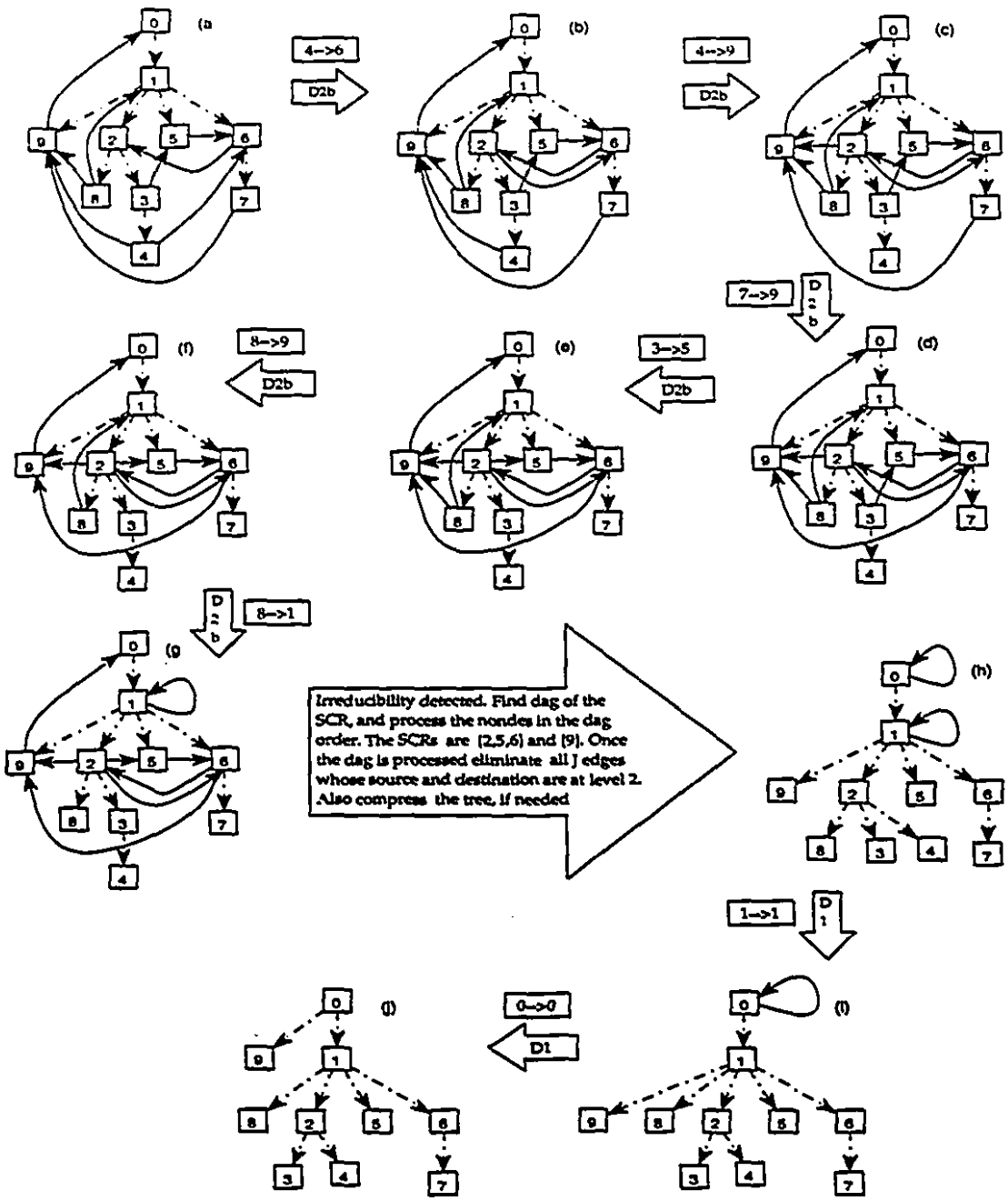


Figure 10.7: A trace of DJ graph reduction for the irreducible flowgraph.

number of $\mathcal{E}$-*rules* applied nE to the size of dominance frontiers $|DF_r|$ 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 1.28 (i.e, $\frac{C'}{C} = 1.28$). This ratio indicates that all the *dominance frontier interval paths* have about $\frac{1.28-1.0}{1.28} = 21.8\%$ of their edges overlapped.

- The value of $\frac{C'}{C} = 1.28$ also suggests that, in an ideal situation, delayed elimination method can be about 1.28 times faster than eager elimination method. In practice, delayed method incurs overhead from bookkeeping and non-profitable path compressions. This is evidenced by the data reported in Table 10.2, where we can see that, on average, delayed elimination method is about 1.15 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 1.45 times faster than iterative method, and eager elimination method is on average 1.27 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 1.15 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 10.1 and Table 10.2 give a summary of our empirical results. The notation used in these tables are given below:

Notation used in Table 10.1 and Table 10.2

Name	Names of procedures.
$\|N\|$	Number of flowgraph nodes.
$\|E\|$	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
$S_{i/e}$	$\frac{T_i}{T_e}$
$S_{i/d}$	$\frac{T_i}{T_d}$
$S_{d/e}$	$\frac{T_e}{T_d}$

10.9.1 Structural Characteristics

Table 10.1 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.[7]

The column Ir shows the number of times the procedure CollapseIrreducible() is invoked. Recall that CollapseIrreducible() is invoked only if irreducibility is

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

Name	$\|N\|$	$\|E\|$	$\|G\|$	Ir	S	$ED1$	$ED2a$	$E2b$	nE	$D2b$	nD	C'	C	$\frac{C'}{C}$	$\|DF_c\|$
aerset	329	460	166	0	0	44	79	200	323	81	204	232	213	1.09	371
aqset	189	258	74	0	0	24	42	118	184	48	114	127	116	1.09	202
bjt	135	187	213	0	0	1	62	114	177	50	113	132	95	1.39	211
card	150	216	63	2	3	11	43	170	224	58	112	230	165	1.39	316
chemset	229	320	94	0	0	23	68	191	282	57	148	198	166	1.19	300
chgeqz	174	248	115	0	0	13	58	142	213	61	132	203	141	1.44	298
clatrs	214	308	160	0	0	17	79	129	225	76	172	152	142	1.07	276
coef	95	137	117	1	4	11	32	69	112	32	75	82	59	1.39	142
comlr	69	91	63	1	3	11	13	47	71	20	44	49	49	1.00	78
dbdsqr	228	327	144	0	0	19	74	176	269	75	168	275	188	1.46	391
dcdcmp	137	187	75	1	2	14	35	127	176	50	99	176	127	1.39	245
dcop	186	261	127	0	0	10	87	143	240	69	166	182	154	1.18	295
dctran	326	458	151	0	0	10	119	278	407	93	222	577	273	2.11	745
deseco	175	236	240	0	0	18	54	107	179	47	119	135	114	1.18	220
dgegv	160	232	108	0	0	11	51	102	164	55	117	200	120	1.67	287
dgesvd	321	470	405	0	0	6	102	206	314	111	219	406	257	1.58	585
dhgeqz	285	408	261	0	0	32	92	238	362	94	218	335	221	1.52	484
disto	133	191	183	0	0	7	53	98	158	58	118	107	102	1.05	186
dlatbs	167	238	134	0	0	12	60	102	174	57	129	127	113	1.12	220
dtgevc	321	459	257	0	0	50	93	213	356	114	257	274	230	1.19	439
dtrevc	248	353	148	0	0	21	85	157	263	75	181	192	173	1.11	318
elpmt	162	227	138	0	0	18	59	99	176	44	121	99	96	1.03	183
equilset	327	451	179	0	0	55	74	205	334	93	222	212	202	1.05	353
errchk	346	482	154	0	0	40	99	267	406	102	241	358	278	1.29	528
iniset	333	486	280	0	0	154	0	154	308	154	308	154	154	1.00	308
init	122	175	129	0	0	17	35	66	118	31	83	70	68	1.03	125
initgas	189	263	114	0	0	40	37	126	203	52	129	126	126	1.00	205
jsparse	281	403	214	0	0	52	70	185	307	76	198	185	183	1.01	313
modchk	306	419	137	0	0	25	111	205	341	74	210	240	213	1.13	390
moseq2	161	217	248	0	0	2	67	137	206	50	119	181	122	1.48	267
mosfet	214	295	263	0	0	1	91	231	323	76	168	307	184	1.67	427
noise	115	159	135	0	0	7	57	72	136	43	107	77	76	1.01	147
out	403	589	404	0	0	92	92	306	490	136	320	330	313	1.05	525
reader	182	235	159	0	0	8	53	28	89	15	76	28	28	1.00	89
readin	406	611	240	2	2	29	84	685	798	174	285	1457	574	2.54	1675
setupgeo	188	275	195	0	0	44	37	112	193	61	142	127	118	1.08	218
setuprad	195	286	223	0	0	55	38	124	217	68	161	124	124	1.00	220
smvgear	212	310	224	0	0	66	23	248	337	91	180	363	203	1.79	468
solveq	196	289	186	0	0	72	28	160	260	93	193	162	162	1.00	265
twldrv	168	243	594	0	0	16	59	127	202	49	124	189	133	1.42	280
Average	219	312	188	0.2	0.3	29	62	167	258	72	163	229	164	1.28	341

Table 10.1: Structural characteristics.

detected at a particular level. The column S shows the maximum size of non-trivial 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.[8] Previous approaches perform iteration over a normally much larger region when an irreducible region is encountered [Bur90, SS79]. One classical approach for handling irreducible regions consists of identifying the *smallest single entry strongly connected region* that encloses the irreducible region [SS79]. Using this method we found the sizes of the single entry regions enclosing the irreducible regions to be 25 for coef, 33 for colmr 78 for card, and 28 for dcdcmp.[9] In [Bur90] proposes a method that is similar to Schwartz and Sharir's method, except that the single entry region need not be strongly connected. Burke's method, although improves upon Schwartz and Sharir's method, still identifies a much larger region than our method. We counted manually using Burke's method for the procedure comlr and found the size of the single entry region that encloses the irreducbile region to be 31.

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 10.8 and Figure 10.9, 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.

[8]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.

[9]These numbers were generated using *Sparse* compiler being developed at the Department of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology. I sincerely thank Priyadarshan Kolte for providing me with these results.

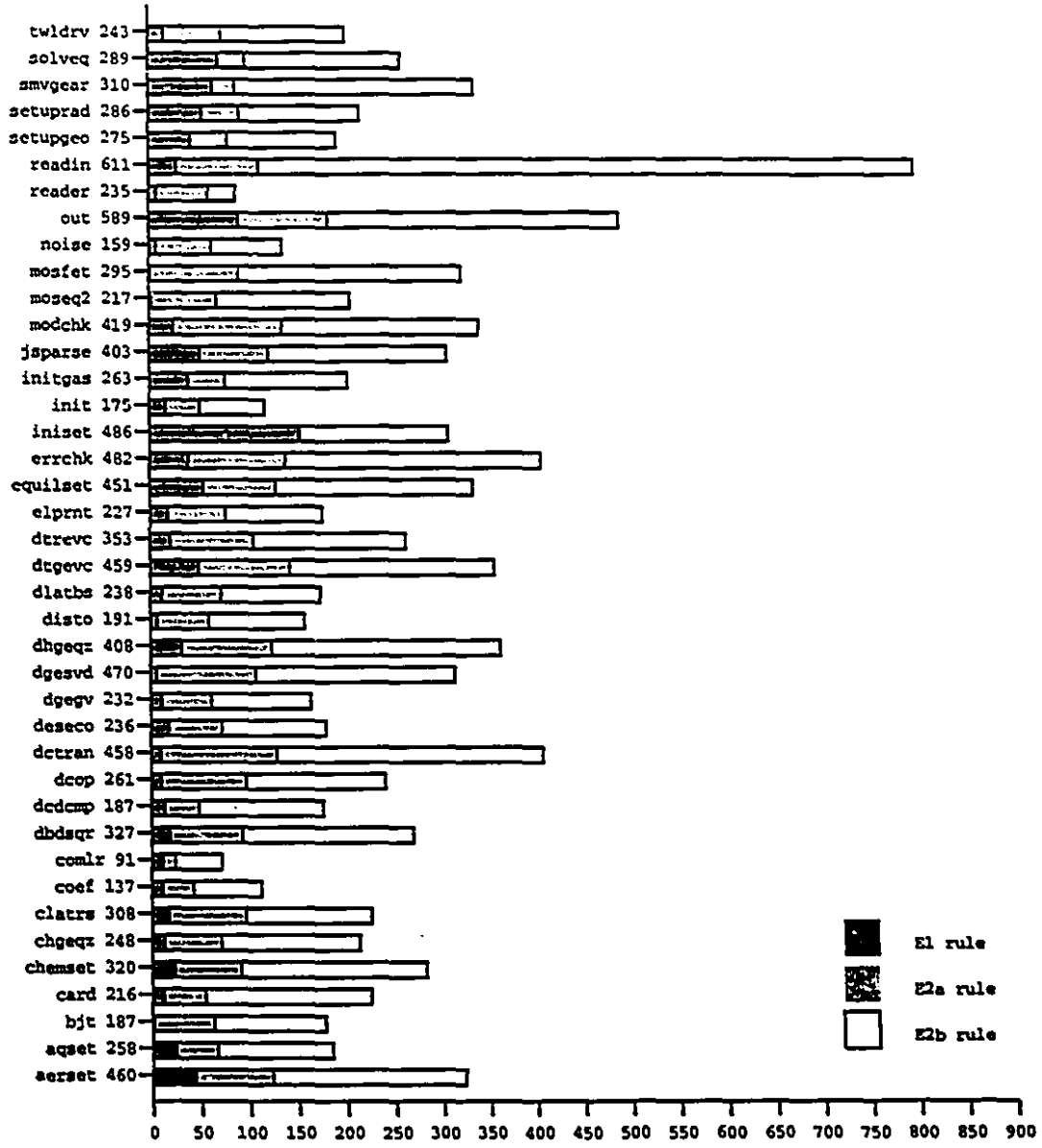


Figure 10.8: A profile of the number of $\mathcal{E}$-*rules* applied.

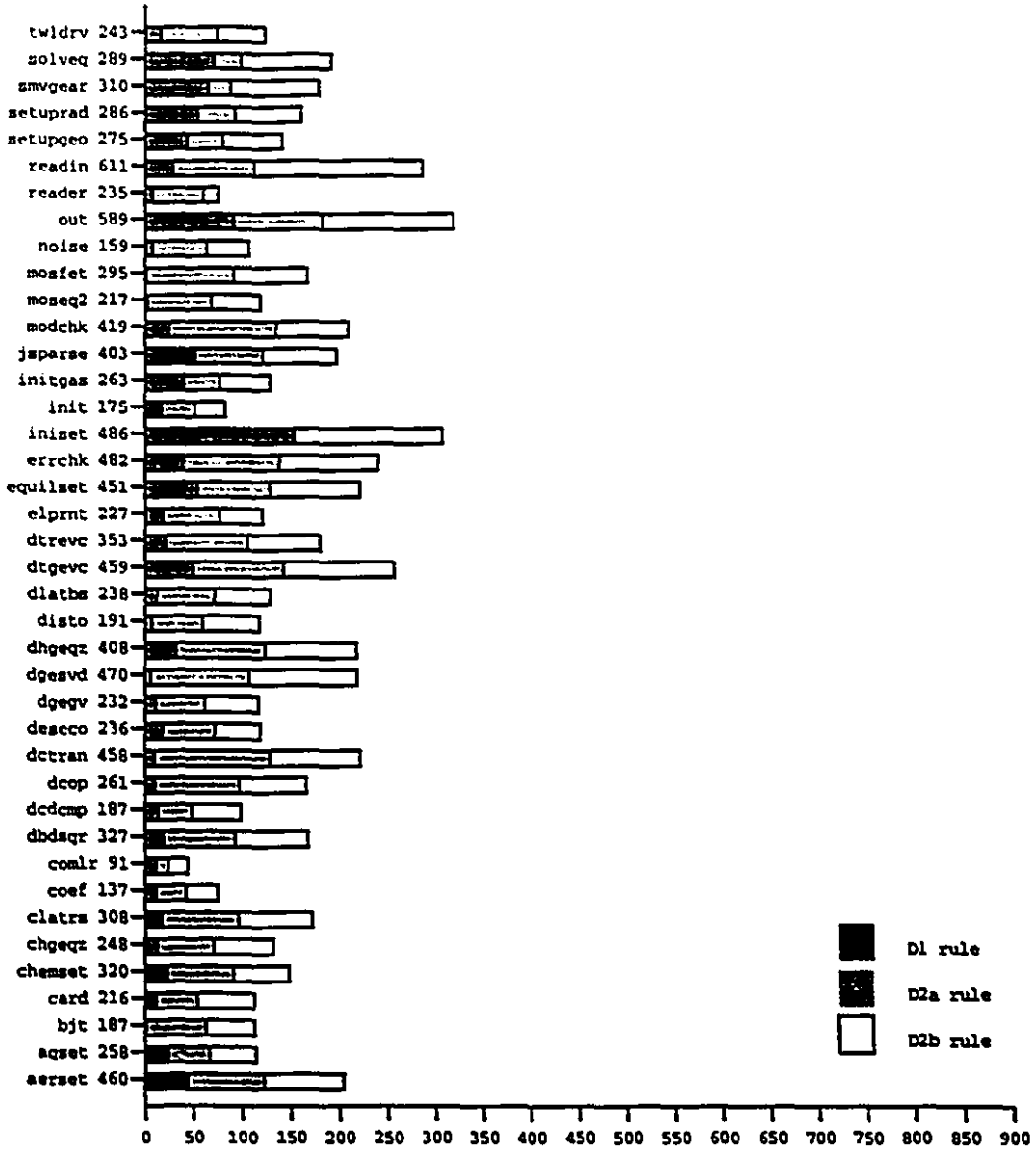


Figure 10.9: A profile of the number of *D-rules* applied.

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 1.28, indicating that delayed elimination method should, ideally, be faster than eager elimination by a factor of 1.28 (if we ignore the overhead of bookkeeping and nonprofitable path compressions in delayed elimination).

In Figure 10.10 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 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 10.1 and from the plot shown in Figure 10.10, 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

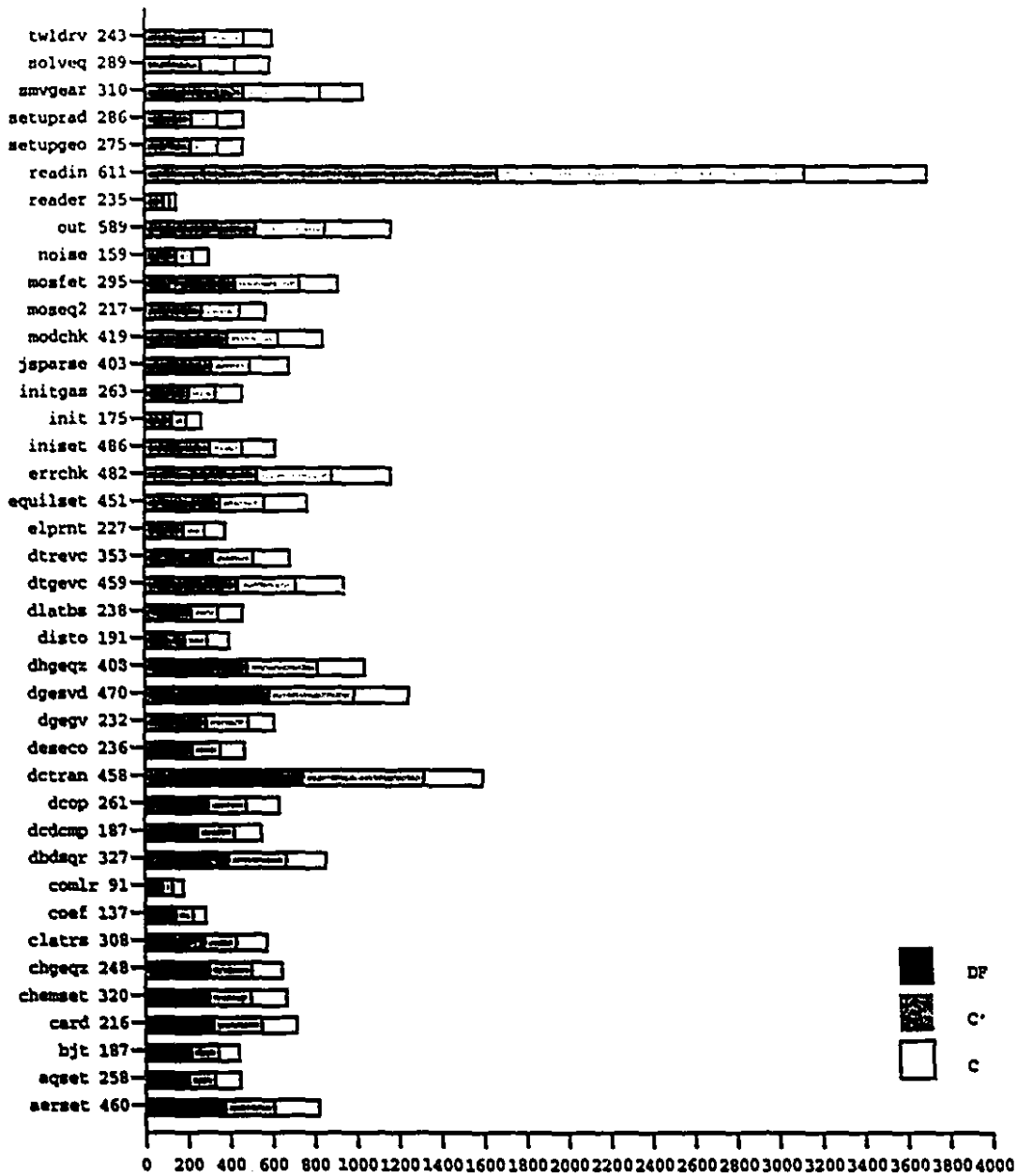


Figure 10.10: A comparison of $|DF_c|$ with C and C'.

use $|DF_e|$ and $|E_f|$ from Table 10.1 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.

10.9.2 Execution Performance

Table 10.2 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. The columns $S_{i/e}$ and $S_{i/d}$ gives the speedups of eager method over iterative method and delayed method over iterative method, respectively.

For the data given in Table 10.2, we observed the following characteristics.

- For iteration method we observed that the execution time is linearly proportional to the product $|E| \times |G| \times nI$, where $|E|$ is the number of flowgraph edge, $|G|$ is the total number of downward exposed definitions, and nI is the number iteration required for convergence of the iteration algorithm (Figure 10.11).

- For both eager and delayed elimination methods we observed that the execution time is linearly proportional to the product $O(|E| \times |G|)$ (Figure 10.12.

From the execution characteristics we can see that the eager elimination method is competitive with the 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.

Theoretically, the eager elimination method is worse than the delayed elimination method in terms of time complexity. However, our empirical results demonstrate that eager method is very competitive when compared with delayed

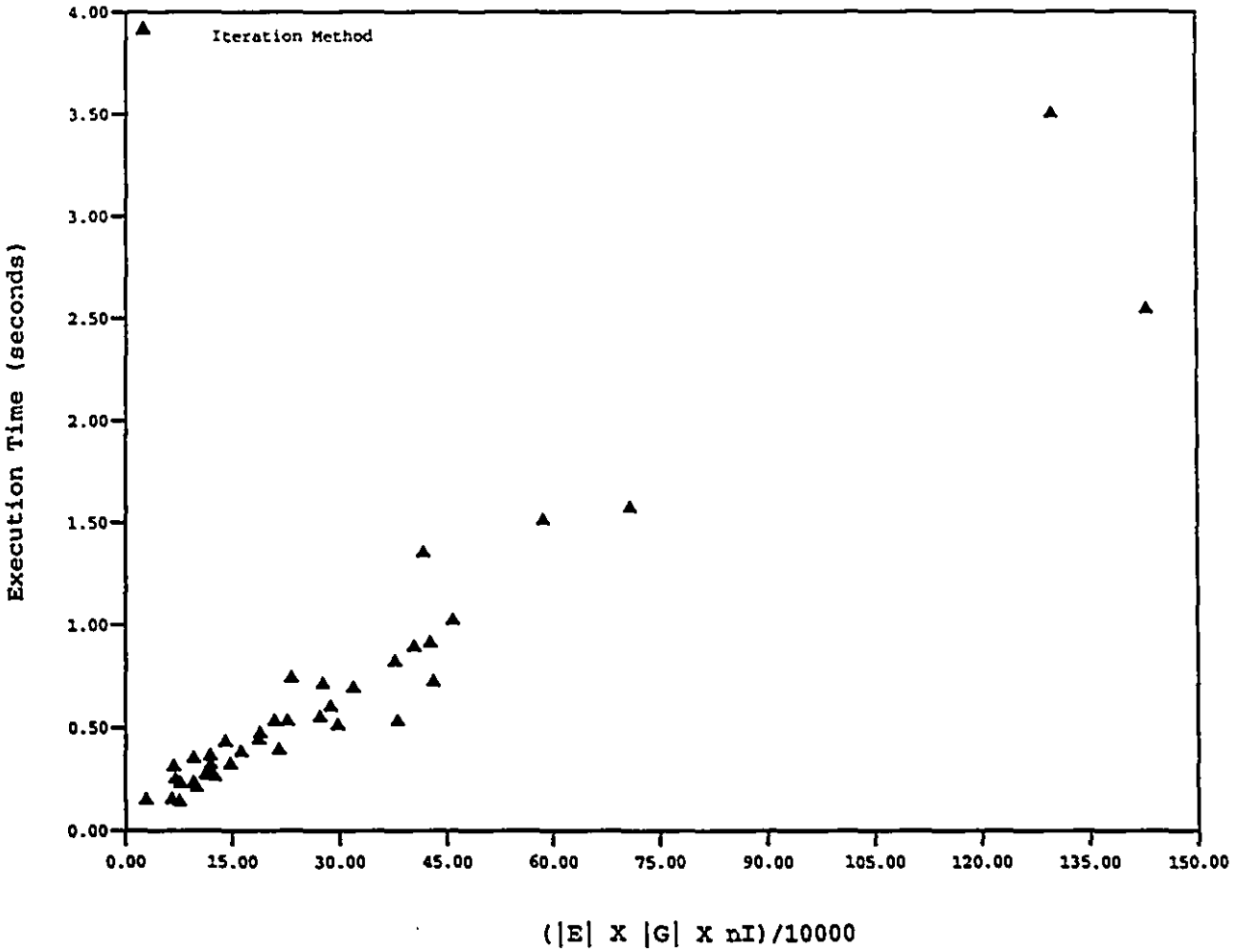


Figure 10.11: Execution characteristics of the iterative method on our test suites.

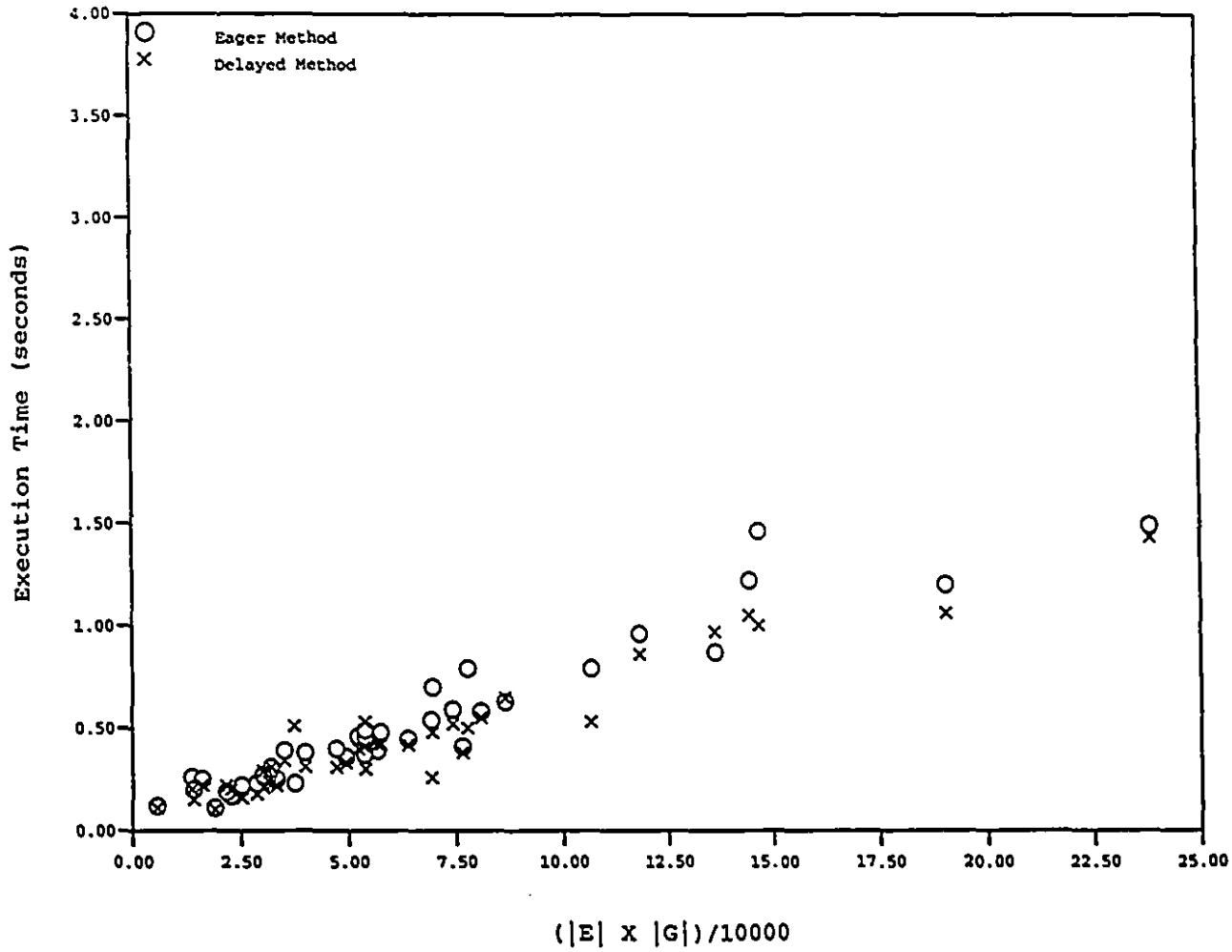


Figure 10.12: Execution characteristics of eager and delayed methods on our test suites.

method. From a pragmatic point view, we thus recommend that one implement the eager method. Not only is it simple and easy to implement, but also it is amenable to incremental data flow analysis (Chapter 11 and [SG95a]).

We found several "outstanding" procedures when we examined the data reported in Table 10.1 and 10.2. 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 10.2, 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.

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.

10.10 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.

Name	$\|E\|$	nI	T_i	T_e	T_d	$S_{i/e}$	$S_{i/d}$	$S_{e/d}$
aerset	460	6	1.02	0.41	0.38	2.49	2.68	1.08
aqset	258	4	0.23	0.11	0.10	2.09	2.30	1.10
bjt	187	3	0.36	0.38	0.31	0.95	1.16	1.23
card	216	5	0.31	0.26	0.20	1.19	1.55	1.30
chemset	320	4	0.29	0.24	0.21	1.21	1.38	1.14
chgeqz	248	4	0.27	0.23	0.18	1.17	1.50	1.28
clatrs	308	3	0.32	0.36	0.33	0.89	0.97	1.09
coef	137	6	0.35	0.25	0.22	1.40	1.59	1.14
comlr	91	5	0.15	0.12	0.11	1.25	1.36	1.09
dbdsqr	327	4	0.47	0.40	0.31	1.17	1.52	1.29
dcdcmp	187	5	0.25	0.20	0.15	1.25	1.67	1.33
dcop	261	3	0.21	0.25	0.22	0.84	0.95	1.14
dctran	458	4	0.71	0.54	0.26	1.31	2.73	2.08
deseco	236	4	0.53	0.39	0.42	1.36	1.26	0.93
dgegv	232	3	0.14	0.22	0.16	0.64	0.88	1.38
dgesvd	470	2	0.53	1.20	1.06	0.44	0.50	1.13
dhgeqz	408	4	0.91	0.79	0.53	1.15	1.72	1.49
disto	191	4	0.43	0.39	0.34	1.10	1.26	1.15
dlatbs	238	3	0.23	0.31	0.31	0.74	0.74	1.00
dtgevc	459	6	1.57	0.96	0.86	1.64	1.83	1.12
dtrevc	353	4	0.53	0.46	0.40	1.15	1.32	1.15
elprnt	227	4	0.26	0.27	0.24	0.96	1.08	1.13
equilset	451	5	0.89	0.58	0.55	1.53	1.62	1.05
errchk	482	4	0.51	0.59	0.52	0.86	0.98	1.13
iniset	486	2	0.55	0.87	0.97	0.63	0.57	0.90
init	175	5	0.27	0.17	0.20	1.59	1.35	0.85
initgas	263	4	0.32	0.27	0.29	1.19	1.10	0.93
jsparse	403	5	0.72	0.63	0.65	1.14	1.11	0.97
modchk	419	5	0.60	0.48	0.42	1.25	1.43	1.14
moseq2	217	3	0.38	0.45	0.30	0.84	1.27	1.50
mosfet	295	3	0.74	0.79	0.50	0.94	1.48	1.58
noise	160	3	0.15	0.19	0.22	0.79	0.68	0.86
out	590	6	2.54	1.49	1.43	1.70	1.78	1.04
reader	235	5	0.44	0.23	0.51	1.91	0.86	0.45
readin	611	4	1.51	1.46	1.00	1.03	1.51	1.46
setupgeo	275	4	0.39	0.37	0.41	1.05	0.95	0.90
setuprad	286	5	0.69	0.45	0.42	1.53	1.64	1.07
smvgear	310	6	1.35	0.70	0.48	1.93	2.81	1.46
solveq	289	7	0.82	0.49	0.53	1.67	1.55	0.92
twldrv	243	9	3.50	1.22	1.05	2.87	3.33	1.16
Average	312	4	0.66	0.50	0.44	1.27	1.45	1.15

Table 10.2: Timings and speedups.

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 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.[10] 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

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

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.

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 exit 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.

Chapter 11

A New Framework for Elimination-Based Data Flow Analysis: Incremental Analysis

One of my favorite philosophical tenets is that people will agree with you only if they already agree with you. You do not change people's minds.

—Frank Vincent Zappa

In this chapter we present a new approach for incremental data flow analysis based on elimination methods. Our approach is based on incrementalizing our eager elimination method (Chapter 10). Compared to previous elimination-based incremental data flow analyses, our approach can handle arbitrary non-structural and structural changes, including irreducibility. To incrementally update data flow solutions we use properties of dominance frontiers and iterated dominance frontiers, and these properties are valid for both reducible and irreducible flowgraphs. In the next section we introduce and motivate the problem of incremental data flow analysis. In Section 11.2, we briefly review our eager elimination method. In Section 11.3, we introduce the concept of initial and final data flow equations, which are central to our approach. In Section 11.4 and Section 11.5, we give algorithms for updating the final flow equations for non-structural changes and for structural changes, respectively. Once the final flow equations have been updated, we next show how to update the final data flow solutions for both structural and non-structural changes, in Section 11.6. In Section 11.7, we prove the correctness of our approach and also analyze its time complexity. Finally, in

Section 11.8, we compare our work with other related work.

11.1 Introduction and Motivation

There are two classical approaches to incremental data flow analysis: one based on iteration methods [PS89], and ancther based on elimination methods [RP88, CR88]. Marlowe and others have extensively studied the relative merits of one approach over the other [Mar89, RMP88, MR90b, BR90]. Marlowe has also proposed a hybrid scheme that combines the two approaches [MR90b, Mar89].

In this chapter we present a new approach for incremental data flow analysis that is based on our eager elimination method. Compared to previous elimination-based incremental data flow analyses, our approach can handle arbitrary non-structural and structural program changes, including irreducibility. A novel aspect of our approach is that we use simple properties of dominance frontiers and iterated dominance frontiers for updating the data flow solutions. In Chapter 8 we showed how to use such properties in the context of incremental dominator tree update problem. In this chapter we will go one step further and show how to exploit them in the context of incremental data flow analysis.

In Chapter 10 we proposed a new approach for elimination-based data flow analysis that uses DJ graphs for reduction and variable elimination. We proposed two variations of our approach: (1) eager elimination method, and (2) delayed elimination method. Both approaches perform reduction and variable elimination on DJ graphs in a bottom-up fashion, ordered by the levels of the nodes on the dominator tree. Unlike the eager elimination method, the delayed elimination method also compresses the dominator tree to improve the worst-case time complexity of the eager elimination method (Chapter 10 and see also [SGL95]).

In this chapter we show how to incrementalize our eager elimination method. Incrementalizing the delayed elimination method involves incrementally maintaining the compressed dominator tree, and doing this is a complex process and may not be worth the effort. Ryder and Paull have similarly shown that incrementalizing other delayed approaches, such as the Hecht-Ullman algorithm, also involves maintaining auxiliary structures (e.g., 2-3 tree) while updating the data flow solutions. Maintaining such structures may out-weigh the benefits of

incremental data flow analysis [RP88].

The major features of our algorithm are as follows:

- Unlike many of the previous incremental elimination algorithms, our algorithm can handle arbitrary non-structural and structural program changes, including irreducibility.
- We use simple properties of dominance frontiers and iterated dominance frontiers for updating the data flow solutions. These properties were introduced for constructing Static Single Assignment form and other other Sparse Evaluation Graphs [CFR+91, CCF91]. In this chapter we will show how to exploit such properties in the context of incremental data flow analysis.

11.2 Exhaustive Eager Elimination Method: An Overview

In this section we briefly review our exhaustive eager elimination method (Chapter 10). Recall that our exhaustive eager elimination method consists of three phases:

1. Reduce the DJ graph to its dominator tree in a bottom-up fashion using *$\mathcal{E}$-rules*.
2. Reduce the system of data flow equations by eliminating variables.
3. Propagate the final data flow solutions in a top-down manner on the dominator tree.

Rather than reducing a DJ graph to a single node we only eliminate J edges in a bottom-up fashion, preserving the structure of the dominator tree for the entire duration of the algorithm. During the bottom-up reduction we apply the *$\mathcal{E}$-rules* for eliminating flow variables and reducing the DJ graph to its dominator tree. The *$\mathcal{E}$-rules* are always applied to a J edge $y \rightarrow z$ such that y is a *non-join* node, and there are no other J edges whose source node is greater than $y.level$, the level number of node y. Each application of *$\mathcal{E}$-rules* transforms some reduced DJ graph $\mathcal{G}^i$ to $\mathcal{G}^{i+1}$, until the DJ graph reduces to its dominator tree.

The E1 rule eliminates a self-loop, and computes closure of its recursive equation.

Definition 11.1 (E1 rule) *Let $\mathcal{G}^i = (N, E)$ be the ith reduced DJ graph. Let y be a non-join node such that y contains a self-loop. Let $H_y : O_y = f(O_y)$ be the flow equation at node y. The E1 rule is given below:*

- (i) Graph reduction:

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

  $E1(< H_y : O_y = f(O_y) >) = < O_y = f^*(O_y) >$

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

Definition 11.2 (E2 rules) *Let $\mathcal{G}^i = (N, E)$ be the ith reduced DJ graph. Let y be a non-join node such y do not contain a self-loop. Let $y \rightarrow z$ be a J edge, and let $x = idom(y)$. Let $H_y : O_y = kO_x + m$ be the equation at node y, such that the parameters k and m does not contain any variables. Finally, let $H_z : O_z = aO_y + b$ be the equation at node z, where a and b does not contain the variable O_y. There are two cases:*

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

- (i) Graph reduction:

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

  $E2a(< H_z : O_z = aO_y + b >) = < H_z : O_z = a(kO_x + m) + b >$

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

- (i) Graph reduction:

  $E2b(< \mathcal{G}^i, N, E, y \xrightarrow{J} z >) = < \mathcal{G}^{i+1}, N, (E - \{y \xrightarrow{J} z\}) \cup \{x \xrightarrow{J} z\} >$ [1]
- (ii)Variable elimination:

  $E2a(< H_z : O_z = aO_y + b >) = < H_z : O_z = a(kO_x + m) + b >$

The newly inserted edge $idom(y) \rightarrow z$ in E2b is called the **derived edge** of $y \rightarrow z$. An important point to note here is that before an E2 rule is applied to an edge $y \rightarrow z$, we first eliminate the self-loop $y \rightarrow y$, if it exists, using an E1 rule.

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

Handling Irreducibility

In our exhaustive eager elimination method irreducibility is detected whenever we cannot apply any of the $\mathcal{E}$-*rules*, although there are J edges that originate at the current level. If this is the case, we apply Tarjan's Strongly Connected Component (SCC) algorithm and collapse every non-trivial component to a single node. In applying the Tarjan's algorithm to the reduced DJ graph, we visits only the nodes via J edges whose source and destination nodes are at the current level. 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. We compute closure of equations of all nodes in S, if needed. We also express the flow equations at all nodes at level i in terms of flow variable of their immediate dominator node. This is possible because we are eliminating the flow variables in the topological order of the SCC. Next we eliminate all J edges whose source and destination nodes are at level i. Finally we apply E2b rule to all J edges whose source nodes are at level i and destination nodes are at levels less i. Once this is done, all the J edges whose source nodes are at this level are eliminated, and so we can continue to apply $\mathcal{E}$-*rules* to nodes at levels less than i.

11.3 Problem Formulation

In the next several sections we present our approach for incremental data flow analysis. In this section we will set the stage for our approach. Specifically we will introduce the concept of *initial* and *final* flow equations in Section 11.3.1, and introduce the steps involved in the updating the final data flow solutions in Section 11.3.2. We will also show how DF graphs, introduced in Chapter 9, is related to the concept of "derived edges" introduced in Chapter 10 (Section 11.3.3). In this chapter we will use DF graphs for updating data flow solutions.

11.3.1 Initial and Final Flow Equations

Our exhaustive eager elimination method consists of three steps: (1) reduce the DJ graph to its dominator tree in a bottom-up fashion, (2) eliminate the variables by substitution, and (3) propagate the solution of root node to all other nodes, determining their corresponding solution. In Chapter 10 we showed that at the

end of elimination phase (step 2), the flow equation at each node depends only on the output flow variable of its immediate dominator node. In other words, let y be a node and let $x = idom(y)$, the flow equation at node y, at the end of elimination phase, would resemble:

$$H_y^F : f^F(O_x) = O_y = P_y^F O_x + G_y, \tag{11.1}$$

where P_y^F and G_y^F are final parameters of the equation at node y. We will call the set of equations at the end of elimination phase as the **final flow equations**. In contrast, we will call the set of flow equations prior to DJ graph reduction and variable elimination as the **initial flow equations**. We will denote the initial flow equation at a node y as follows:

$$H_y^0 : f^0(I_y) = O_y = P_y^0(\bigwedge_{z \in Pred_f(y)} O_z) + G_y^0 \tag{11.2}$$

where P_y^0 and G_y^0 are initial parameters of the initial flow equation at node y, $\wedge$ is merge (union) operator, and $Pred_f(y)$ is a set of predecessors in the corresponding flowgraph.

Example 11.1

Consider a forward data flow problem with union as the merge operation (e.g., Reaching Definitions). The initial flow equation at each node for the example flowgraph shown in Figure 11.1 is as follows:

$$\begin{aligned}
O_S &= G_S^0 \\
O_1 &= P_1^0 O_S + G_1^0 \\
O_2 &= P_2^0 O_1 + G_2^0 \\
O_3 &= P_3^0(O_1 + O_8) + G_3^0 \\
O_4 &= P_4^0(O_2 + O_3 + O_7) + G_4^0 \\
O_5 &= P_5^0 O_4 + G_5^0 \\
O_6 &= P_6^0 O_4 + G_6^0 \\
O_7 &= P_7^0(O_5 + O_6) + G_7^0 \\
O_8 &= P_8^0 O_7 + G_8^0
\end{aligned}$$

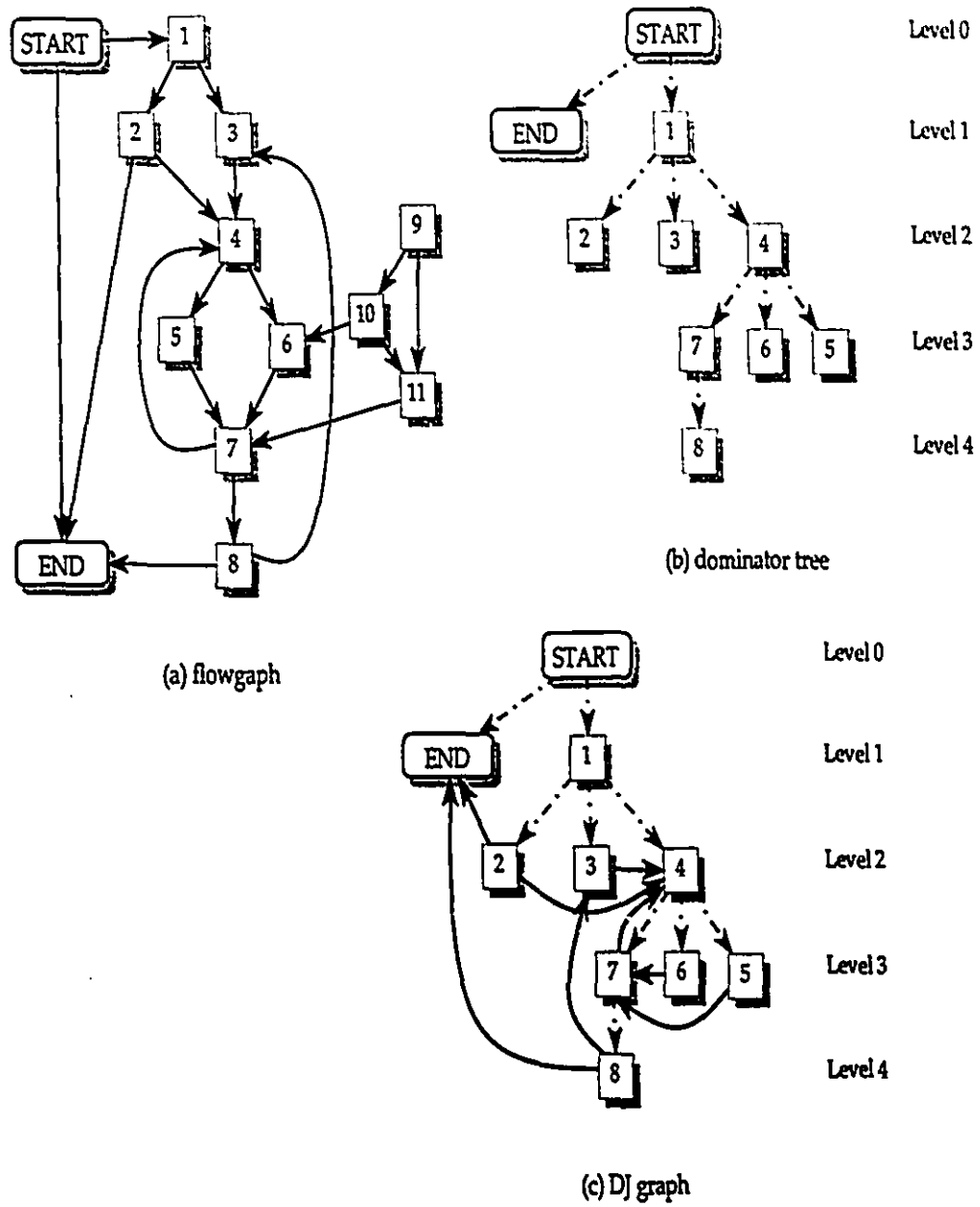


Figure 11.1: Another example of a flowgraph, its dominator tree, and its DJ graph.

$$O_E = P_E^0(O_S + O_2 + O_8) + G_E^0$$

The corresponding final flow equation at each node would resemble:

$$\begin{aligned}
H_S^F : O_S &= G_S^0 \\
H_1^F : O_1 &= P_1^0 O_S + G_1^0 \\
H_2^F : O_2 &= P_2^0 O_1 + G_2^0 \\
H_3^F : O_3 &= P_3^0 O_1 + (P_3^0 P_8^0 P_7^0 P_5^0 + P_3^0 P_8^0 P_7^0 P_6^0) O_4 + P_3^0 P_8^0 P_7^0 G_6^0 + \\
&\quad P_3^0 P_8^0 P_7^0 G_5^0 + P_3^0 P_8^0 G_7^0 + P_3^0 G_8^0 + G_3^0 \\
H_4^F : O_4 &= P_4^0 P_2^0 O_1 + (P_4^0 P_3^0 P_8^0 P_7^0 P_5^0 + P_4^0 P_3^0 P_8^0 P_7^0 P_6^0) O_4 + \\
&\quad P_4^0 P_3^0 P_8^0 P_7^0 G_5^0 + P_4^0 P_3^0 P_8^0 P_7^0 G_6^0 + P_4^0 P_3^0 P_8^0 G_7^0 + P_4^0 P_3^0 G_8^0 + \\
&\quad P_4^0 G_3^0 + G_4^0 \\
H_5^F : O_5 &= P_5^0 O_4 + G_5^0 \\
H_6^F : O_6 &= P_6^0 O_4 + G_6^0 \\
H_7^F : O_7 &= (P_7^0 P_5^0 + P_7^0 P_6^0) O_4 + P_7^0 G_5^0 + P_7^0 G_6^0 \\
H_8^F : O_8 &= P_8^0 O_7 + G_8^0
\end{aligned}$$

In the above system of equations, H_3^F and H_4^F are mutually recursive, and the final equations at nodes 3 and 4 is the closure of the two equations. Once the closure is determined, the final flow equations at these nodes are expressed in terms of their immediate dominator node (see Chapter 10).

11.3.2 Basic Steps

Given the notion of initial and final flow equations we are ready to lay the foundation of our approach for incremental data flow analysis. The problem of incremental data flow analysis can be concisely stated as follows [PS89]:

> *Given a program and a correct solution to a data flow problem over that program, update the affected parts of the current solution to reflect a change in the program without unnecessary reinitialization and recalculation of the entire data flow solution.*

To simplify the presentation, we will consider only the following two types of incremental changes:

- Non-structural change: The parameters of the initial flow equation H_y^0 at a node y are modified.
- Structural change: A flowgraph edge $x \rightarrow y$ is either inserted or deleted in the flowgraph.

One can easily extend and implement other types of incremental changes using the results of this chapter. It is important to remember that all incremental algorithms rely on having correct solutions at all nodes prior to incremental changes. Once an incremental change is effected, incremental algorithms will update (ideally) only those solutions that are affected due to the incremental change [Mar89]. Let us denote the data flow solution at each node y, prior to incremental change, as α_y^F. We will call α_y^F as the **final flow solution** at a node y. Let $x = idom(y)$, then one can easily show the following input-output relation will hold at node y prior to an incremental change.

$$H_y^0 : \alpha_y^F = f^F(\alpha_x^F) = P_y^F \alpha_x^F + G_y^F \tag{11.3}$$

The first step in our approach is to associate with each node y: (1) the initial flow equation H_y^0, (2) the final flow equation H_y^F, and (3) the final flow solution α_y^F. Now supposing we induce an incremental change (such as updating the parameter of the initial flow equations, inserting a new flowgraph edge, or deleting an existing flowgraph edge), our incremental data flow analysis will update the data flow solutions in two steps:

- Update the final data flow equations; and
- Update the final data flow solutions.

We will handle structural and non-structural changes separately. The complete algorithm for incremental data flow analysis is given below:

Algorithm 11.1 *The following algorithm updates the data flow solution when the corresponding flowgraph is subjected to structural and non-structural incremental changes.*

```
MainIDFA()
{
306:  if (IncrementalChange == Non-Structural) then
307:    Update final flow equations for non-structural changes (Section 11.4)
308:  else
309:    Update final flow equations for structural changes (Section 11.5)
310:  endif
311:  Update final data flow solutions (Section 11.6).
}
```

To effectively handle the incremental changes we will use **Dominance Frontier (DF) graph** introduced in Chapter 9 (Section 9.5). Recall that a DF graph is nothing but the dominator tree of a flowgraph augmented with edges $x \rightarrow y$ such that $y \in DF(x)$. We will use DF graph for updating the final flow equations.

11.3.3 DF Graphs Revisited

In our exhaustive eager elimination method when we apply the E2b rule to an edge $y \rightarrow z$, we eliminate $y \rightarrow z$ and insert $x \rightarrow z$, where $x = idom(y)$. In other words we "derive" the edge $x \rightarrow z$ from $y \rightarrow z$ by applying the E2b rule, and so we call $x \rightarrow z$ as a derived edge of $y \rightarrow z$. Now if $x.level > z.level$, we will (subsequently) apply E2b rule to $x \rightarrow z$ to derive another edge $w \rightarrow z$, where $w = idom(x)$. We will continue to apply E2b rule to the "derived edges" as long as its source and destination nodes are at the same level, at which point we either apply an E1 rule or an E2a rule, eliminating the derived edge.

An astute reader may observe the relation between derived edges and dominance frontiers. To see this let $y \rightarrow z$ be the original J edge in the initial DJ graph, then $x \rightarrow z$ will be a derived edge if $z \in DF(x)$. This is interesting because given a dominator tree we can augment the dominator tree with edges $x \rightarrow y$ such that $y \in DF(x)$. The resulting graph is the Dominance Frontier (DF) graph. Figure 11.2(b) gives the DF graph for our example flowgraph shown in Figure 11.1.

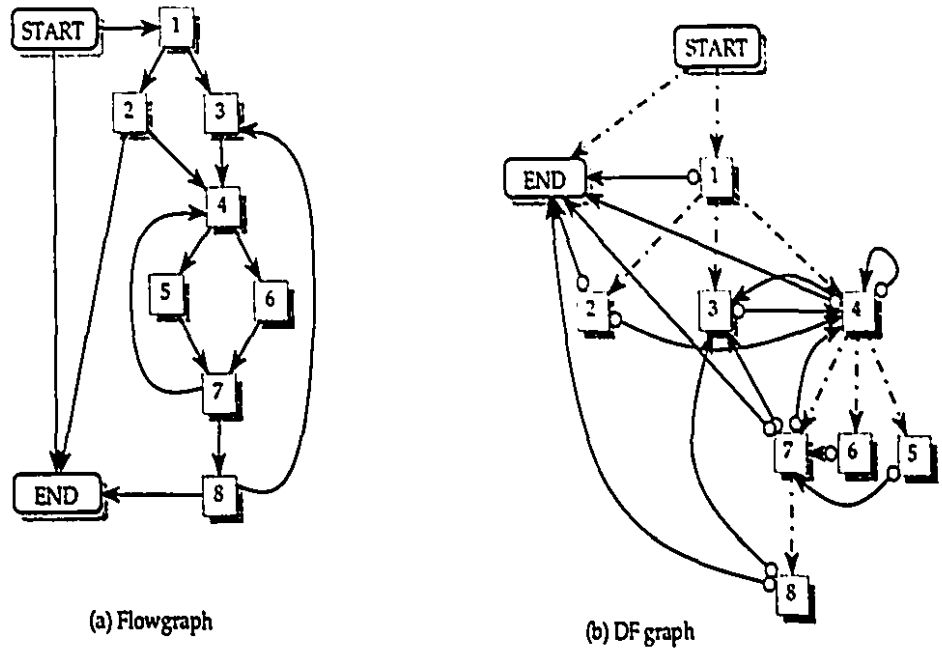


Figure 11.2: A flowgraph and its DF graph.

11.4 Updating Final Data Flow Equations: Non-Structural Changes

In this section we will show how to update the final data flow equations for a non-structural change at a node y in a flowgraph. An example of a non-structural change at a node y is addition or deletion of a new definition. Because of this update, the reaching definition information at other nodes may be affected. By adding or deleting a definition, we are essentially changing the parameters P_y^0 and G_y^0 of the initial flow equation at node y.

Given an incremental change at a node y, the first step is to determine the set of nodes whose final flow equations is affected. To determine the set of affected nodes we made one key observation. Consider the final flow equation of node 7 of the example flowgraph shown in Figure 11.1.

$$\begin{aligned} H_7^F : O_7 &= (P_7^0 P_5^0 + P_7^0 P_6^0) O_4 + P_7^0 G_5^0 + P_7^0 G_6^0 \\ &= P_7^F O_4 + G_7^F \end{aligned}$$

We notice that the final parameters P_7^F and G_7^F on the RHS of the equation are made up of the initial parameters P_5^0, G_5^0, P_6^0, and G_6^0 of nodes 5 and 6, respectively. It is important to remember that the parameters of both the initial flow equation and the final flow equation at a node are constant values. We will use the term *appears* to mean that the parameters of the final flow equations are computed from the parameters of initial flow equations during the reduction and variable elimination phase of the eager elimination algorithm. Given this notion, the key question to ask is: how are the parameters of the initial flow equations related to the parameters of the final flow equations? We found a surprisingly simple relation between the initial and the final parameters of flow equations.

Claim 11.1 *Let P_w^F and G_w^F be the parameters of the final flow equations at node w. P_u^0 and G_u^0 will appear in P_w^F and G_w^F if and only if either $w = u$ or $w \in IDF(u)$.*

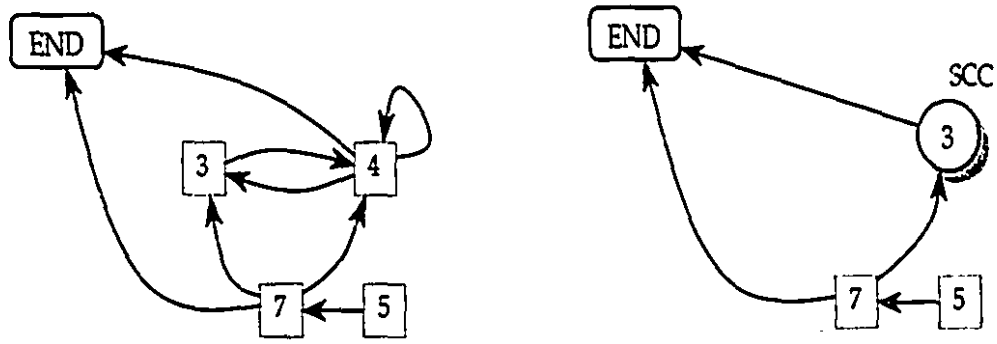
We will prove this claim later in Section 11.7. The above claim has an important implication in our incremental algorithm. Supposing we make a non-structural change to a node y, thereby affecting P_y^0 and G_y^0. From Claim 11.1 we should update the final flow equation of all nodes that are in the $IDF(y)$. Now since we are changing P_y^0 and G_y^0, the final flow equation H_y^F at node y should also be updated. Let FEqAffected(y) be the set of all nodes whose final flow equation change due to a non-structural change at y (i.e., FEqAffected(y) is the set of "affected" nodes). To determine the set nodes whose final flow equations have to be updated, we will use the following key result:

Claim 11.2 *Let the parameters P_y^0 and G_y^0 of the initial flow equation at node y be updated. Then the final flow equation at a node w is* **affected** *(i.e., w is in* FEqAffected(y)*) if and only if $w \in \{y\} \cup IDF(y)$.*

Therefore, from the above key result, we can see that the first step in updating the final flow equations is to compute the set $IDF(y)$. Computing $IDF(y)$ is much simpler using DF graphs than DJ graphs. We can easily show that a node w is in $IDF(y)$ if and only if there exists a path P from y to w in DF graph that does not contain D edges. Therefore to compute the $IDF(y)$ we determine all nodes that are reachable from y without visiting any D edges.

Once we compute the $IDF(y)$, we next construct a **Projection Graph** $Proj(y)$ of the DF graph with respect to the nodes in $\{y\} \cup IDF(y)$. The projection graph

$Proj(y)$ consists of nodes in set $\{y\} \cup IDF(y)$, and we insert an edge $p \rightarrow q$ between any two nodes in $Proj(y)$ if and only if $q \in DF(p)$ in DF graph.[2] For example, let us construct the projection graph $Proj(5)$. First we compute $IDF(5)$, which consists of nodes $\{3, 4, 7, \text{END}\}$. The next step is to insert edges $u \rightarrow w$ from the DF graph, such that $w \in DF(u)$ and both u and w are in $Proj(5)$. The resulting projection graph $Proj(5)$ is shown in Figure 11.3(a).



(a) Proj(5) (b) DAG

Figure 11.3: The projection graph $Proj(5)$ graph, and its dag.

Given the projection graph we will next show how to update the final flow equations of the affected nodes. It is important to remember that a node w is affected (because of a non-structural change at node y) if and only if w is in $Proj(y)$. It is also important to note that $Proj(y)$ need not be acyclic. So the first step is to apply Tarjan's Strongly Connected Component (SCC) algorithm and process each component in the topological order of the dag of SCCs. For example, the $Proj(5)$ in Figure 11.3(a) is not acyclic, and so we determine its dag of SCCs, which is shown in Figure 11.3(b). Now if a SCC in the dag of $Proj(y)$ contains a cycle, we compute the closure of the equations of the nodes in the SCC (as in the exhaustive case). We can easily show that if an SCC contains more than one node, then all the nodes will be at the same level in the DF graph (see Chapter 6). Now given the dag of SCCs of $Proj(y)$, *we process each SCC in the*

[2]Note that the set $DF(p)$ is same in both the original flowgraph and its DF graph.

dag in topological order. It is important to remember that the only variable on the RHS of the final flow equation at a node should be the output flow variable of its immediate dominator. Given this we next show how to construct the final flow equations of all the affected nodes. The projection graph $Proj(y)$ helps provide an ordering in which we can update the final flow equations at the affected nodes. As discussed above, we first apply Tarjan's Strongly Connected Component (SCC) algorithm to the projection graph. Then we determine a topological ordering on the strong components. Finally, we visit the strong components S in topological order to update their final flow equations. There are three cases to consider:

Case 1: S is a single node and has no self-loop. Let $Pred_f(S)$ be the set of predecessor of node S in the corresponding flowgraph. Assume that the final flow equation at every predecessor $p \in Pred_f(S)$ is correct (either previously updated or unaffected by the incremental change). To updated final flow equation H_S^F at node S, we start with its initial flow equation. That is, we first construct the following equation at node S:

$$H_S : O_S = P_S^0(\bigwedge_{t \in Pred_f(S)} O_t) + G_S^0, \tag{11.4}$$

Starting from this equation, we eliminate variables from it in a bottom-up fashion, as in our exhaustive eager elimination, until the only variable in H_S is that of its immediate dominator node. Recall that during the elimination process we create and delete derived edges. The topological ordering of the SCCs ensures that the final equation at source nodes of these derived edges are in its final form. At the end of the elimination process the equation at node S will be in its final form. Notice that the above (incremental) update is nothing but a "selective" exhaustive eager elimination process, but restricted to the equation at node S.

Case 2: S is a single node and has a self-loop. Here we assume that the final flow equation at every predecessor of node S, excluding S itself, is correct. We next perform a "selective" exhaustive variable elimination starting from the initial flow equation H_S of node S. At the end of this selective variable elimination, the only variable that will remain in H_S is O_S and $O_{idom(S)}$.

At this point we compute the closure of the recursive equation to break the dependency of O_S on itself (*á la* E1 rule) and obtain the updated final flow equation H_S^F.

Case 3: S contains more than one node. In this case we have irreducibility, and so we have to simultaneously determine the final flow equation of all nodes in S. These nodes have the same immediate dominator. In this case we will assume that the final flow equation at all nodes in $N - S$ are correct (where N is a set of all nodes in the DF graph). As before, we again perform "selective" variable elimination, for each equation at w nodes in S, until the only variables remaining in the system of equations are those of nodes in S and the output variable their immediate dominator node. Finally we perform fixed-point iterations over all the mutually recursive equation and determine their closure.

The complete algorithm is given below.

UpdateFlowEq(y)

{

312: Compute $IDF(y)$

313: Determine $Proj(y)$

314: Apply Tarjan's SCC algorithm determine the SCCs in $Proj(y)$.

315: For each SCC S in topological order **do**

316: **Switch** (S)

317: **Case 1:** S is a single node and does not contain a self-loop.
Compute the final flow equation as described in the main text.

318: **Case 2:** S is a single node and contain a self-loop.
Compute the final flow equation as described in the main text.

319: **Case 3:** S contains more than one node.
Compute the final flow equation as described in the main text.

320: **EndSwitch**

}

Example 11.2

Consider the previous example where we induce a non-structural change to node 5. The corresponding $Proj(5)$ and its dag is shown in

Figure 11.3. We will process the nodes in the dag order. So we will first process node 5. Node 5 is a single node with no self-loop, and so corresponds to case 1 (step 317). The only predecessor of node 5 (on the flowgraph) is node 4. The new final flow equation of node 5 is

$$H_5^{nF} = P_5^{n0} O_4 + G^{n0},$$

where the superscript n means *new* equation (or parameters).

Next we process node 7, and this also corresponds to case 1 in the algorithm (step 317). The two predecessor nodes of node 7 are 5 and 6. The initial flow equation of node 7 depends only on equations at node 5 and node 6, and is given below:

$$O_7 = P_7^0(O_5 + O_6) + G_7^0$$

Note that the equation at node 6 is unaffected, but at node 5 it is affected (and updated). After eliminating O_5 and O_6 in above equation we get the new final flow equation for node 7:

$$\begin{aligned} H_7^{nF} : O_7 &= (P_7^0 P_5^{n0} + P_7^0 P_6^0) O_4 + P_7^0 G_5^{n0} + P_7^0 G_6^0 \\ &= P_7^{nF} O_4 + G_7^{nF} \end{aligned}$$

Next we process the non-trivial SCC 3, which corresponds to case 3 in the algorithm (step 319). The SCC 3 consists of nodes 3 and 4. We first set up the intial flow equations for these two nodes:

$$\begin{aligned} O_3 &= P_3^0(O_1 + O_8) + G_3^0 \\ O_4 &= P_4^0(O_2 + O_3 + O_7) + G_4^0 \end{aligned}$$

Next we perform variable elimination, and reduce the equation to the following form:

$$\begin{aligned} H_3^{nF} : O_3 &= P_3^0 O_1 + (P_3^0 P_8^0 P_7^0 P_5^{n0} + P_3^0 P_8^0 P_7^0 P_6^0) O_4 + P_3^0 P_8^0 P_7^0 G_6^0 + \\ &\quad P_3^0 P_8^0 P_7^0 G_5^{n0} + P_3^0 P_8^0 G_7^0 + P_3^0 G_8^0 + G_3^0 \\ H_4^{nF} : O_4 &= P_4^0 P_2^0 O_1 + (P_4^0 P_3^0 P_8^0 P_7^0 P_5^{n0} + P_4^0 P_3^0 P_8^0 P_7^0 P_6^0) O_4 + \\ &\quad P_4^0 P_3^0 P_8^0 P_7^0 G_5^{n0} + P_4^0 P_3^0 P_8^0 P_7^0 G_6^0 + P_4^0 P_3^0 P_8^0 G_7^0 + \\ &\quad P_4^0 P_3^0 G_8^0 + P_4^0 G_3^0 + G_4^0 \end{aligned}$$

Notice that the only variables in the above two equations are those of the nodes in the SCC 3 and their immediate dominator node. Next we compute the closure of the two equations using fixed-point iterations, and express them only in terms of O_4, the immediate dominator of the nodes in SCC 3.

11.5 Updating Final Data Flow Equations: Structural Changes

Next we will show how to update the final flow equations for structural changes (i.e, insertion and deletion of a flowgraph edge). Our algorithm for structural changes consists of the following steps:

- Update the dominator tree of the flowgraph.
- Update the dominance frontier relation of the flowgraph.
- Update the final flow equations.

In Chapter 8 we gave a simple algorithm for updating the dominator tree of a flowgraph, and in Chapter 9 we gave a simple algorithm for updating dominance frontiers of a flowgraph. We will use these two results in this chapter. Recall that $p \rightarrow q$ is a DF edge in DF graph iff $q \in DF(p)$. Therefore the problem of updating DF graphs is isomorphic to the problem of updating dominance frontiers.

Once we have updated the dominance frontier relation, we will next show how to update the final flow equations at all the 'affected' nodes. Again, let the edge $x \rightarrow y$ be the structural update (either inserted or deleted). Now the key question to ask is: at which nodes are the final data flow equations affected when a new edge $x \rightarrow y$ is updated. The answer to the above question is given in the following claim:

Claim 11.3 *Let $x \rightarrow y$ be the edge that is updated in the flowgraph. The final data flow equation at a node w is affected if and only if $w \in \{y\} \cup IDF(y)$.*

This is very interesting. Recall in Section 11.4 we made a similar claim (Claim 11.2) for non-structural updates. This means that we can essentially use

the same algorithm as given in Section 11.4 for updating the final data flow equations even for structural changes. But rather than using the 'old' DF graph we will have to use the updated DF graph, to compute the new set of final flow equations. In other words we first compute the $IDF(y)$ on the new DF graph and, as before, construct the projected graph $Proj(y)$. Once the projected graph is constructed, we update the final flow equations at all nodes in $Proj(y)$, using the same strategy described in Section 11.4.

11.6 Updating Final Data Flow Solutions

In this section we will show how to update the final data flow solutions α^F, once the final data flow equations have been updated. The first key question to ask is: at which nodes the final data flow solutions are affected because of an incremental change. As before let $Proj(y)$ be the set of nodes where data flow equations have been updated. Once the data flow equations have been updated at these nodes, we may have to update their final data flow solutions. It is important to remember that the final data flow equation at each node will depend only on the output flow variable of its immediate dominator node.

Let w be a node in $Proj(y)$ whose *new* final data flow equation is $f_w^{Fnew}(I_w)$. Let α_w^{Fold} be the old final solution at this node (which may not be a correct solution). Let $u = idom(w)$ whose data flow solution α_u^{Fcor} is a 'correct" solution.[3] If the following relation holds at node w

$$\alpha_w^{Fold} = f_w^{Fnew}(\alpha_u^{Fcor}),$$

then we need not update the final solutions of the children nodes of w. Otherwise, we have to compute α_w^{Fcor} and mark the immediate dominee of w (i.e., children of w on the dominator tree) as potentially being affected, and repeat the process for each child node. Since the flow equation at node w is depends only on its immediate dominator node, and since the solution of its immediate dominator is correct, we can compute the new correct final solution at w as follows:

$$\alpha_w^{Fcor} = f_w^{Fnew}(\alpha_u^{Fcor})$$

[3]By correct solution we mean that either its original solution was unaffected because of the incremental change, or some how has been correctly updated.

It is important to observe that before we can update the final solution at a node w, we have to ensure that the final solution of its immediate dominator node is correct. Also, once (and if) we update w final solution we have to mark all its children nodes to be potentially affected, and so their final solutions have to be updated. Therefore we order the nodes in $Proj(y)$ in terms of the levels of the nodes on the dominator tree, and process the nodes in a top-down fashion. We will use a data structure akin to *OrderedBuckets* to keep track nodes where the final solutions are possibly affected. *OrderedBuckets* is an array of buckets ordered by levels of the nodes on the dominator tree. When a node x is inserted in the bucket, it will be inserted at the bucket $OrderedBuckets[x.level]$. We will initially insert all the nodes whose final flow equation was updated into the *OrderedBuckets*. We then iterate by picking out one node at a time in a *top-down* fashion and updating its solution. We check if the old solution at the node is consistent with the final data flow equation at the node, if so we pick the next node from the *OrderedBuckets*. Otherwise we compute the new final solution and insert all its children nodes into the *OrderedBuckets*. The complete algorithm is given below.

UpdateFlowSol()

```
{
321:  Insert the set of nodes Proj(y) in the OrderedBuckets
322:  foreach i = 1 to NumLevel − 1 do
323:    while((u = IdfaGetNode(i)) ≠ NULL) do
324:      w = idom(u)
325:      if(α_u^Fold ≠ f_u^Fnew(α_w^Fcor)) then
326:        α_u^Fcor = f_u^Fnew(α_w^Fcor)
327:        Insert the children nodes of u in the OrderedBuckets.
328:      else
329:        α_u^Fcor = α_u^Fold /* Old solution is the correct solution */
330:      endif
331:    endwhile
332:  endfor
}
```

The function **IdfaGetNode(i)** returns a node x if one exists in the ith bucket,

otherwise it returns $NULL$. Notice at step 329 if the old solution is "correct" then the children nodes are not affected (unless they are already in the $OrderedBuckets$).

11.7 Correctness and Complexity

In this section we prove the correctness of our incremental algorithm and analyze its time complexity. The main theorem which establishes the correctness of Algorithm 11.1 is Theorem 11.1. The proof of Theorem 11.1 relies on Lemma 11.6 and Lemma 11.5. Lemma 11.5 establishes the correctness of updating final flow equations for both structural and non-structural changes. To update the final flow equation of a node we must first know that it is affected. Lemma 11.3 and Lemma 11.4 gives the necessary and sufficient condition to determine the *exact* set of nodes whose final flow equations have to be updated for non-structural and structural changes, respectively. The validity of these two lemmas is based on another key lemma, Lemma 11.2, which establishes a relation between the parameters of initial flow equations and the parameters of the final flow equations. To prove Lemma 11.2 we will use another lemma, Lemma 11.1, which relates the concept of dominance frontiers and derived edges (see Definition 10.5).

The validity of Lemma 11.5 for structural changes relies on the correctness of dominance frontier update algorithm (Theorem 9.1). Finally, Lemma 11.6 establishes the correctness of updating final flow solutions once the final flow equations have been updated.

In our chain of proofs, we begin with Lemma 11.1, that relates the concept of dominance frontiers and derived edges.

Lemma 11.1 *In the exhaustive eager elimination method, a derived edge $u \rightarrow w$ will be created and processed at some stage in the elimination phase iff $w \in DF(u)$.*

Proof:

The "if" part: From Lemma 4.1 we know that if $w \in DF(u)$ then there exists a J edge $t \rightarrow w$ such that u dom t and $w.level \leq u.level$. Now if $t \rightarrow w$ is a J edge then this edge will be processed during

some stage in the elimination phase (i.e. one of $\mathcal{E}$-rules will be applied to this edge and the edge will be eliminated). Now if $t = u$, then we are done (i.e, $u \rightarrow w$ is a derived edge). Otherwise we will apply one or more E2b rules to the derived edges of $t \rightarrow w$ until the source node of the derived edge is at the same level as the destination. Since u dom t, and $w.level \leq u.level$, eventually a derived edge $u \rightarrow w$ will created and processed.

The "only if" part: If $u \rightarrow w$ is a derived edge then by Definition 10.5 $u \rightarrow w$ was created and processed at some stage during the elimation phase. In other words, it was derived from some J edge $t \rightarrow w$ such that u dom t, and the level of u is greater than or equal to the level of w. But from Lemma 4.1 we know that if $t \rightarrow w$ is a J edge such that u dom t and $u.level \geq w.level$, the $w \in DF(u)$. Hence the result.

■

Next we prove Lemma 11.2. This lemma is exactly the same as Claim 11.1. This lemma states that the parameter P_u^0 and G_u^0 in the initial flow equation of node u will appear in P_w^F and G_w^F if and only if either $w = u$ or $w \in IDF(u)$. It is important to remember that the parameters of both the initial flow equation and the final flow equation at a node are constant values. We use the term *appears* to mean that the parameters of the final flow equations are computed from the parameters of the initial flow equations during the reduction and variable substitution phase of the eager elimination algorithm.

Lemma 11.2 *Let P_w^F and G_w^F be the parameters of the final flow equations at node w. P_u^0 and G_u^0 will appear in P_w^F and G_w^F if and only if either $w = u$ or $w \in IDF(u)$.*

Proof:

It is obvious to see that if $w = u$ then P_u^0 and G_u^0 will appear in P_w^F and G_w^F, and vice versa. So let us assume that $w \neq u$.

The "if" part: We want to show that if $w \in IDF(u)$, then P_u^0 and G_u^0 will appear in P_w^F and G_w^F.

First we will show that if $w \in DF(u)$ then P_u^0 and G_u^0 will appear in P_w^F and G_w^F. Using Lemma 11.1 we can see that if $w \in DF(u)$ then

$u \rightarrow w$ is a derived edge. When this edge is processed during the elimination phase, we will eliminate O_u in the flow equation H_w at node w, by substituting it with $P_u^F O_r + G_u^F$, where $r = idom(u)$. The parameters P_u^F and G_u^F will contain P_u^0 and G_u^0 and so will appear in H_w, and hence H_w^F, the final flow equation of node w.

Now to show that if $w \in IDF(u)$, then P_u^0 and G_u^0 will appear in P_w^F and G_w^F, we can use inductive definition of iterative dominance frontiers. Now, if $w \in IDF(u)$, then there exist nodes $t_0, \ldots, t_k$ such that $w = t_0$, $u = t_k$, and $t_i = DF(t_{i+1})$, where $0 \leq i \leq k-1$. We can use induction on i to show the result.

The "only if" part: We want to show that if P_u^0 and G_u^0 appear in P_w^F and G_w^F, then $w \in IDF(u)$.

Now assume to the contrary that $w \notin IDF(u)$. Then either w is not reachable from u or there exists a node s that strictly dominates w and $s \in IDF(u)$. But if w is not reachable then P_u^0 and G_u^0 will not appear in P_w^F and G_w^F contradicting our assumption. Assume that w is reachable from u but is not in $IDF(u)$. Now we know that there exists a node s closest to w that strictly dominates w and $s \in IDF(u)$ (follows from Lemma 8.3). Therefore all paths from u to w must pass through s. Now if s stdom w then a derived edge will never be created between s and w (since s can never be in $DF(w)$), and so P_u^0 and G_u^0 will never propagate to w via node s, and since all paths from u to w must pass through s, P_u^0 and G_u^0 can never appear in P_w^F and G_w^F, contradicting our initial assumption. Therefore w must be in $IDF(u)$.

∎

Next we will prove Lemma 11.3, which is same as the Claim 11.2.

Lemma 11.3 *Let a non-structural change be induced at a node y. The final flow equation at a node w is* **affected** *(i.e., w is in* FEqAffected(y)*) if and only if $w \in \{y\} \cup IDF(y)$.*

Proof:

First of all observe that when we induce a non-structural change at a node y, we are essentially changing the parameters P_y^0 and G_y^0 of

the initial flow equation at y. From Lemma 11.2 we know that if $w \in IDF(y)$ then P_y^0 and G_y^0 will appear in P_w^F and G_w^F, and vice-versa. Therefore if P_y^0 and G_y^0 are updated we have to update the parameters P_w^F and G_w^F of all final flow equations that appear in $y \cup IDF(y)$. The converse is also true i.e., if a node is in $\{y\} \cup IDF(y)$ then its final flow equation is affected. ∎

Next we will establish the correctness of Lemma 11.4, which which is same as the Claim 11.3.

Lemma 11.4 *Let $x \rightarrow y$ be the edge that is updated in the flowgraph. The final data flow equation at a node w is affected if and only if $w \in y \cup IDF(y)$.*

Proof:

First of all notice that insertion and/or deletion of an edge does not affect the set of initial flow equations. Given this, the rest of the proof is based on the following observation. When $x \rightarrow y$ is updated we are essentially changing the input flow information of node y. Therefore the parameters of the final flow equation at node y is affected. This situation exactly corresponds to non-structural updates, except that we do not change the parameters of the initial flow equation of node y. The rest of the proof is exactly same as in the proof of Lemma 11.3 ∎

Recall that once we identify the set of nodes whose final flow equations are affected we need to proceed to re-evaluate their new final flow equation. Next we will show that steps 314 to 320 correctly re-evaluates the final flow equation for both structural and non-structural changes.

Lemma 11.5 *The steps 314 to 320 correctly updates the final flow equations for both structural and non-structural changes.*

Proof:

From Lemmas 11.3 and 11.4 we know exactly at which nodes the final flow equations are affected. For non-structural updates we do not change the structure of the DF graph. For structural changes we first update the DF graph, and then update the final flow equations. From Theorem 9.1 we know that DF graph is correctly updated. Therefore

in the rest of the lemma we will not distinguish between the two incremental changes.

First of all observe that we are processing SCCs in topological order of the dag obtained by collapsing the non-trivial SCCs in $Proj(y)$ (step 315). Therefore when processing an SCC S we are ensured that final flow equations at all the nodes u such that $u \rightarrow w$ is an edge in DF graph, $u \notin S$, and $w \in S$ are correct (either previously updated or is unaffected). This topological order also ensures that the algorithm will terminate in finite time.

Given this it is enough to show that the final flow equation at affected node that is derived by eliminating variables from the corresponding initial equation is the correct final flow equation. The derivation of the final equation depends on the type of SCC S, and we will handle them seperately.

Case 1: S is a single node and does not contain a self-loop. Its initial equation is given by

$$H_S^0 : O_S = P_S^0(\bigwedge_{t \in T} O_t) + G_S^0, \tag{11.5}$$

where $T = \{t | t \in Pred_f(S)\}$. To eliminate variables from the above equation we perform selective exhaustive eager elimination. Since we are processing the nodes in the topological order, we are ensured that the final flow equation at the destination node of every derived edge (generated and processed during the selective eager elimination process) is correct (either updated or unaffected). Since we showed the correctness of the exhaustive eager elimination (Theorem 11.1), the correctness of the selective elimination directly follows from it.

Case 2: S is a single node and contains a self-loop The only difference between this case and Case 1 is that we also compute closure of the recursive equation. In the E1 rule we also compute closure whenever there is a self-loop at a non-join node. Once the closure is computed, the equation at node s is the final flow equation.

Case 3: S contains more than one node. In this case we first form a set of mutually recursive equation by eliminating all output flow variables O_p such that p is not in s. This situation corresponds to the irreducible case in our exhaustive elimination method. As in the exhaustive case we determine the closure of all the mutually dependent equations, and then express the final flow equation at all nodes in s in terms of their immediate dominator.

In each case we have established the correctness of the derivation of the final flow equations. ∎

Next we will show that the algorithm for updating final flow solution is correct. First of all observe that the data flow solutions at all nodes whose final flow equation is updated is potentially affected. So we may have to update their solution. Now let α_u^{Fold} be the old solution at a node u, i.e., solution of node u prior to incremental change. Let $w = idom(u)$, and assume that its solution α_w^{Fcor} is correct. If $\alpha_u^{Fold} = f_u^{Fnew}(\alpha_w^{Fcor})$ then we need not update the solution at node u. Otherwise we have to update its solution and mark the solutions of all its children node as being affected.

Lemma 11.6 *The procedure* **UpdateFlowSol()** *correctly updates the final flow solutions at all the nodes whose final solutions are incorrect.*

Proof:

First of all notice that if a node u is returned by IdfaGetNode() then it is possibly affected. A node u is returned by IdfaGetNode() if and only if it was previously inserted in *OrderedBuckets*; and a node u is inserted in *OrderedBuckets* if it either in $Proj(y)$ (step 321) or the solution of its parent node was previously updated (step 327). Since we are processing the nodes in a top-down manner we will eventually update the final flow solutions at all the affected nodes.

∎

Finally we prove the correctness of our incremental data flow analysis.

Theorem 11.1 *The Algorithm 11.1 correctly updates the data flow solutions for both structural and non-structural changes to flowgraphs.*

Proof:

Follows from Theorem 9.1, Lemma 11.5, and Lemma 11.6. ∎

Next we will analyze the time complexity of our approach.

Theorem 11.2 *The worst case time complexity of Algorithm 11.1 is* $O(|E| \times |N|)$.

Proof:

(1) For both structural and non-structural updates selective eager elimination could, in the worst case, be performed over all nodes. And so the worst case time complexity for selective elimination is $O(|E| \times |N|)$ (follows from Theorem 10.2).

(2) the worst case time complexity of updating both dominance frontiers and dominator trees is bounded by $O(|E| \times |N|)$ (see Chapter 8 and Chapter 9).

(3) The worst-case time for updating the final solution is $O(|N|)$, since we are propagating the solution on the dominator tree.

Combining (1), (2), and (3) we can see that the worst case time complexity of Algorithm 11.1 is $O(|E| \times |N|)$. ∎

For both non-structural changes and for insertion of an edge our algorithm is expected to behave linearly in practice since the size of the dominance frontier is linear in practice [CFR+91]. The cost for updating the data flow solution for deletion case is dominated by the cost for updating the dominator tree. Since we use Purdom and Moore algorithm for this step, the time complexity is quadratic in the worst-case for the deletion.

11.8 Discussion and Related Work

In this chapter we proposed a new approach for incremental data flow analysis based on elimination methods. Previous work most relevant to ours is due to Carroll and Ryder [CR88]. We will first give a detailed comparison of our approach with theirs, and then compare with other related work.

Carroll and Ryder's algorithm is based on the notion of *reduce and borrow* concept for updating the data flow solutions [CR88]. They *reduce* a monotone

data flow problem to an attributed (dominator) tree problem, and then *borrow* the well-known Reps's attribute update algorithm for updating the data flow solutions [Rep82, RTD83]. They use Graham-Wegman elimination algorithm as a starting point for mapping data flow problems to attributed dominator tree problems [GW76]. They decorate each node in the dominator tree with its (1) initial flow equation (2) final flow equation, and (3) the correct solution. These decorations are treated as attributes of the dominator tree. Once they construct an attributed dominator tree, they modify the well-known Reps's algorithm for updating the attributed dominator tree [Rep82, RTD83]. Reps's original algorithm can only handle updates to attributed parse tree, which are derived from attributed grammars. Since dominator trees are not parse trees, Carroll and Ryder generalize Reps's algorithm for handling updates to arbitrary trees.

In an attributed parse tree problem, we associate with each node a *semantic function* which defines the value of that attribute in terms of values of other attributes [ASU86]. Given an attributed parse tree, if the attribute at node y uses the value of attribute node at x, then we say that node (attribute) y depends on node (attribute) x. The *dependency graph* of a set of attributes A is a graph whose nodes are the elements of A, and there is an edge $u \rightarrow w$ if w depends on u. The value of an attribute is *consistent* if it equals the value returned by the attribute's semantic function. A *solution* to an attributed parse tree is a set of consistent values for all its attributes [ASU86].

Reps's original algorithm can handle updates only if the dependence graph of the attributed tree is acyclic [Rep82, RTD83]. Carroll and Ryder show that if the original flowgraph is reducible then the dependence graph of the attributed dominator tree is also acyclic. Presence of irreducibility in the original flowgraph introduces cycles in the dependence graph of the attributed dominator tree, and so we cannot use Reps's algorithm for updating such trees.

Reps's algorithm basically consists of replacing the affected sub-parse tree with a correct sub-parse tree and propagating the attribute values of the new subtree to all other nodes that depend on it. To ensure optimality the attributes are propagated on a projected graph of the dependence graph, called the sub-ordinate and superior *characteristic graph* [Rep82, RTD83]. Carroll and Ryder show how to construct these characteristic graphs for attributed dominator problem, and use them for updating and propagating final data flow solutions.

The sub-parse tree replacement in Reps's algorithm corresponds to restructuring of the dominator tree in Carroll and Ryder's algorithm. Carroll and Ryder also propose an algorithm for updating the dominator tree of the flowgraph. While updating the dominator tree they compute, what they call as *representative edges*, which are central to their update algorithm. These representative edges are then used for updating both the dominator tree and the attributes of the dominator tree. Projection of these representative edges with respect to the root of the affected sub-tree corresponds to the characteristic graphs in Reps's algorithm. For reducible flow graphs the projection of the representative edges form a dag, and so they can update the attributes of the dominator tree in the dag order of the projection graph.

In our algorithm too we "reduce" the problem to an attributed tree problem (since we are annotating the DJ graph with initial flow equation, final flow equation, and the final flow solution). But, unlike Carroll and Ryder's approach, we use simple properties of dominance frontiers and iterated dominance frontiers for updating the final data flow solution, and these properties are valid for both reducible and irreducible flowgraphs.

Although we do not use Reps's update algorithm for updating data flow solutions, we will show how the notion of dominance frontiers and iterated dominance frontiers are tied to the notion of dependence graphs and characteristic graphs used in Reps's algorithm. As in Carroll and Ryder's algorithm the dominator tree in our algorithm corresponds to the parse tree in Reps's algorithm. The DF graph in our algorithm correspond to the dependence graph in Reps's algorithm. Interestingly enough, the representative edges used in Carroll and Ryder's algorithm are nothing but DF edges in our DF graph. The superior characteristic graphs in Reps's and Carroll and Ryder's algorithm correspond to the projection graph $Proj(y)$ in our algorithm, although $Proj(y)$ can contain cycles.[4] Recall that $Proj(y)$ is derived from $IDF(y)$, the iterated dominance frontiers of y. This suggests that the concept of iterated dominance frontiers is deeply related to the concept of superior characteristic graphs used in Reps's algorithm. These two concepts were developed independently, and for different problems—dependency graphs and the characteristic graphs were introduced in the context of attributed grammars, whereas dominance frontiers and iterated

[4] Assuming that a non-structural change is induced at node y or we update an edge $x \rightarrow y$.

dominance frontiers were introduced in the context of Static Single Assignment form [CFR+91]. An interesting direction for future research would be to further explore the possible relation between these two concepts.

Burke proposes an algorithm for elimination-based incremental data flow analysis that use interval graphs for updating and propagating data flow solutions [Bur90]. His algorithm can only handle structural changes (to flowgraphs) that does not change the depth-first spanning tree of the flowgraph. Marlowe and Ryder propose a hybrid incremental algorithm that combines iteration and elimination methods [MR90b]. They first identify strongly connected components in the flowgraph, and they use iteration method within each component, but propagate the solutions to other components using elimination-like method. Although they can handle program arbitrary updates, their incremental algorithm is more coarse-grained; they update and propagate solutions to a much larger set of nodes than our algorithm or Carroll and Ryder's algorithm.

Chapter 12

Conclusions and Future Work

One of the symptoms of an approaching nervous breakdown is the belief that one's work is terribly important.

—Bertrand Russell

As stated in the introduction, the goal of this dissertation was to demonstrate the effectiveness of using DJ graphs for program analysis. To this end, we have presented a number of algorithms for solving simple problems such as loop detection to sophisticated analysis techniques, such as exhaustive and incremental analysis, including construction of sparse evaluation graphs. In this dissertation we have demonstrated that how a simple representation like DJ graphs can be used for solving sophisticated problems. We have also demonstrated that our solution methods are simple, efficient, and general (i.e., can handle arbitrary program structures). We have provided empirical results for many algorithms and compared them with existing ones for similar problems. Our empirical results show that the algorithms presented here are indeed efficient and practical, and can be easily incorporated in a production compiler.

There are other interesting and important open problems that can be solved using DJ graphs. Here we will highlight some of them.

Incremental Computation of Static Single Assignment Form and Sparse Evaluation Graphs

Static Single Assignment (SSA) form and Sparse Evaluation Graphs (SEGs) are intermediate representations that are well suited for solving many data flow and

optimization problems [CLZ86, RWZS8, AWZ88, WZ85, Bri92, CBC93]. In Chapter 7 we gave a simple algorithm for constructing a single SEG in linear time. Although we now have a linear time algorithm, maintaining correct SEGs and SSA form can be expensive throughout a multi-pass compilation process. In a recent report Choi et al. proposed an algorithm for incrementally maintaining the correct SSA form for restricted types of program changes [CSS94]. In particular they do not allow arbitrary insertion and deletion of edges in the corresponding flowgraph. In [CG93], Cytron and Garbshbein show how to efficiently accommodate may alias information in the SSA form. Their algorithm consists of iteratively refining both the alias information and SSA form in a round robin fashion until the two information are sufficiently accurate. The refinement process consists of incrementally updating the SSA form and alias information. The incremental algorithm proposed in the paper does not allow structural changes to flowgraphs.

In this dissertation we proposed efficient algorithms for maintaining dominator trees and dominance frontiers, both of which are fundamental to the construction of the SSA form and SEGs. An interesting and important future work would be to come up with an incremental algorithm for maintaining these sparse representations for arbitrary program changes. Based on our experience with DJ graphs, we believe that DJ graphs are well suited for solving this problem.

Improving the Dominator Update Algorithm for the Deletion of an Edge

In Chapter 8 we gave an algorithm for updating the dominator tree when the corresponding flowgraph was subjected to incremental changes. Although our algorithm for insertion of an edge achieves linear time complexity, the incremental algorithm for the deletion case is quadratic in the worst-case. For the deletion case we modified the Purdom and Moore algorithm to first compute the dominance relation for all (possibly) affected nodes, and then updated the dominator tree using that information. The worst-case quadratic time complexity for the deletion case is because of the quadratic time complexity of Purdom and Moore's algorithm. Now the open question is: Can we compute the exact set of affected nodes and the corresponding new immediate dominators in linear time, even for the deletion case?

Empirical Study of Elimination Methods in the Context of Interprocedural Data Flow Analysis

In this dissertation we gave empirical results of our elimination methods for solving the intraprocedural reaching definition problem. An interesting future work would to study our approach in the context of interprocedural data flow analysis. Tarjan, Rosen, Burke, and others have shown that elimination methods can be used for solving general monotone data flow problems [Tar81, Bur90, Ros80, Ros82, Mar89]. We are not aware of any empirical results on how elimination methods perform compared to iteration methods on real programs for interprocedural analysis. It would be an interesting and worthwhile exercise to implement elimination algorithms for solving interprocedural data flow analysis and quantitatively study them on real benchmark programs.

Empirical Study of Incremental Algorithms in the Context of a Real Compiler

In this dissertation we proposed a number of algorithms for incremental analysis. We are not aware of any published literature that quantitatively evaluates the benefits of incremental analysis in the context of a real optimizing compiler. An interesting and important direction for future work would be to empirically study the benefits of incremental analysis in the context of an optimizing compiler. Most optimizing compilers perform aggressive program transformation, and so it is important that data flow information and other program properties are correctly maintained throughout the entire compilation process. Current methods in most optimizing compilers recompute the data flow information after every change in the program, even within a single optimization phase of the compiler. We expect that incremental analysis would speed-up the compilation process of an aggressive optimizing compiler.

Parallel Data Flow Analysis Using DJ Graphs

In this dissertation we proposed two approaches for elimination based data flow analysis using DJ graphs. An interesting direction for future research would be to parallelize our elimination algorithm. Lee et al proposed a *region* partition scheme for parallel data flow analysis [LRF94]. Lee et al. define a region to a connected subgraph of a flowgraph such that all the incoming edges from other

parts of the flowgraph to the region enter into its head node. Incidentally, region head nodes are join nodes in our DJ graphs. An interesting future work would be to use their region partition algorithm to partition DJ graphs for parallelizing our elimination algorithms.

Thy right is to work only; but never to its fruits; let not the fruit of action be thy motive, nor let thy attachment be to inaction.

—Bhagavad Gita

Appendix A

A Data Flow Analysis Framework

Recall that algorithms for data flow analysis take a program and estimate properties of the program statically. The nature of these properties depends on the data flow problem being solved. Most interesting data flow problems can be expressed within a framework called the *monotone data flow framework*. In this appendix we will briefly discuss this framework. This framework was first introduced by Kildall, and subsequently revised and redefined by others [KU77, Tar81, Mar89]. Here we will essentially follow the notation given by Marlowe [Mar89].

The data flow information in a data flow problem is represented by elements of a *lattice* $\mathcal{L}$, having a commutative, associative, and idempotent meet operation $\wedge$. Intuitively, the operation $l \wedge m$, for $l, m \in \mathcal{L}$, represents information common to both l and m. Given $\mathcal{L}$, we can define a relation $\sqsubseteq$ on $\mathcal{L}$ such that $l \sqsubseteq m$ iff $l \wedge m = l$. In other words, $\sqsubseteq$ defines a partial order on elements of $\mathcal{L}$. Intuitively, the relation $l \sqsubseteq m$ means that l contains less information than m. Dually, we also define $\sqsupseteq$ relation; if $l \sqsubseteq m$ then $m \sqsupseteq l$. If $l \sqsubseteq m$ and $l \neq m$ then $l \sqsubset m$. We can similarly define the dual $\sqsupset$ relation. Finally, the lattice also contains two distinguished elements $\top$ and $\bot$, called the top and bottom elements, satisfying the following relation:

$$\begin{aligned} l \wedge \top &= l \\ l \wedge \bot &= \bot \end{aligned}$$

Typically the $\wedge$ operation is used for merging information at join nodes. Let $X = \{x_1, x_2, \ldots\}$ be a subset of $\mathcal{L}$. We will use the notation $\bigwedge X$ to mean $x_1 \wedge x_2 \wedge \ldots$.

We will impose another restriction on $\mathcal{L}$ called the *descending chain condition*,

in which every chain[1]

$$l_1 \sqsupset l_2 \sqsupset \ldots$$

is finite. Most data flow problems have this property.[2] An important consequence of the descending chain condition is that every meet $\bigwedge X$ can be computed using meet of a finite subset. Note that $\mathcal{L}$ or $X \subseteq \mathcal{L}$ need not be finite. In other words even an infinite lattice can have finite height. Throughout this dissertation we will assume lattices that satisfy descending chain condition.

Next we will discuss the effects of a node (a sequence of instructions) on flow information. These effects are modeled by flow functions $\mathcal{F} \subseteq \{f : \mathcal{L} \to \mathcal{L}\}$. For most interesting data flow problems the flow functions are monotone.[3] In this dissertation we will assume flow functions to be monotone. A function $f \in \mathcal{F}$ is monotone if for any two elements $l, m \in \mathcal{L}$ such that $l \sqsubseteq m$, then $f(l) \sqsubseteq f(m)$. Let f and g be any two functions in $\mathcal{F}$ we will use the notation $f \wedge g$ to mean $(f \wedge g)(l) = f(l) \wedge g(l)$. We will also assume that $\mathcal{F}$ contains an identity function ι.

Let f and g be any two functions in $\mathcal{F}$, we will use the notations $f \circ g$ to denote function composition, and $f^*(l) = \{\bigwedge_{i=0}^{\infty} \{f^i(l) | i \geq 0\}$, the iterated composition of f and $f^0 = \iota$, to denote the reflexive transitive closure of $f(l)$.

One can define classes of data flow problems depending on the finiteness properties of flow functions and lattices. We will define a few of them (for a detailed discussion please see [Mar89]).

k-Boundedness: Let $f^k(l) \sqsupseteq \bigwedge \{f^i(l) | 0 \leq i \leq k\}$ for all $f \in \mathcal{F}$ and $l \in \mathcal{L}$. In this case, we can compute f^k using only $k-1$ meet iterations. An example of a k-bounded problem is the Formal Bound Set problem.

Rapidity and Fastness: Many of the classical bit-vector problems have nicer function properties. Let $f \in \mathcal{F}$, if $f \circ f \sqsupseteq f \wedge \iota$ then f is fast. Notice that fastness is equivalent to 2-boundedness. An important consequence of fastness is that we need at most two iterations for fixed-point computation in a loop. Also, the notion of "loop breaking" in [RP86] is applicable only for fast problems.

---

[1] I.e., a linear order

[2] Two classical examples that do not possess this property are pointer analysis for recursive data structures and type checking [Mar89].

[3] There are practical data flow problems in which flow functions are not monotone; for example, the Generalized Common Sub-Expression Elimination due to Fong [Fon77, Tar81]. Another example where flow functions are not monotone is alias analysis in the presence of dynamic data structures that do not use k-limited graph for approximation [LH88].

A function $f \in \mathcal{F}$ is *rapid* if $f(l) \sqsupseteq l \wedge m \wedge f(m)$ for all $l, m \in \mathcal{L}$. One can easily show that if a function is rapid it is also fast, but not the converse. Another interesting property to observe is that rapidity implies *separability.* An example of a data flow problem that is rapid is reaching definition. An example of a problem that is fast but not rapid is constant propagation.

Distributive: A function $f \in \mathcal{F}$ is *distributive* if $f(l \wedge m) = f(l) \wedge f(m)$. An example of a problem that is distributive is copy constant propagation; and example of a problem that is not distributive is constant propagation.

Given the previous background we are ready to define *monotone data flow framework.* A data flow framework is a tuple $(G, \mathcal{L}, \mathcal{F}, M)$, where

- $G =< N, E, \text{START} >$ is a flowgraph,
- $\mathcal{L} =< S, \bot, \top, \sqsubseteq, \wedge >$ is a lattice,
- $\mathcal{F} \subseteq \{f : \mathcal{L} \rightarrow \mathcal{L}\}$ is a set of monotone functions.
- $M \subseteq \{e \rightarrow f | e \in E,\ f \in \mathcal{F}\}$ is a set of functions mapping edges to flow functions. Sometimes it is convenient to map nodes, rather than edges, to flow functions.

Depending on the nature of $\mathcal{F}$ and $\mathcal{L}$ we can define various classes of the monotone data flow framework, such as *bounded* monotone framework, distributive framework, etc. [Mar89].

Bibliography

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

[AHM+95] APPELBE, B., HARMON, R., MAY, P., WILLS, S., AND VITALE, M. Hoisting branch conditions—Improving superscalar processor performance. In *Eighth International Workshop on Languages and Compilers for Parallel Computing*, pages 20.1–20.14, Columbus, Ohio, 1995.

[AHR+90] ALPERN, B., HOOVER, R., ROSEN, B., SWEENEY, B. K., AND ZADECK, F. K. Incremental evaluation of computational circuits. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 32–42, 1990.

[Amm92] AMMARGUELLAT, Z. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–250, 1992.

[ASU86] AHO, A. V., SETHI, R., AND ULLMAN, J. D. *Compilers–Principles, Techniques, and Tools*. Addison-Wesley Publishing Co., 1986.

[AWZ88] ALPERN, B., WEGMAN, M. N., AND ZADECK, F. K. Detecting equality of variables in programs. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[BG95] BODIK, R., AND GUPTA, R. Array data flow analysis for load-store optimizations in superscalar architectures. In *Eighth International Workshop on Languages and Compilers for Parallel Computing*, pages 1.1–1.15, Columbus, Ohio, 1995.

[BMO90] BALLANCE, R. A., MACCABE, A. B., AND OTTENSTEIN, K. J. The program dependence web: A representation supporting control-, and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 257–271, 1990.

[BR90] BURKE, M. G., AND RYDER, B. G. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Transactions on Software Engineering*, 16(7):723–728, July 1990.

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

[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.

[CBC93] CHOI, J. D., BURKE, M. G., AND CARINI, P. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[CCF91] CHOI, J.-D., CYTRON, R., AND FERRANTE, J. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, 1991.

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

[CF93] CYTRON, R., AND FERRANTE, J. Efficiently computing ϕ-nodes on-the-fly. In *Languages and Compilers for Parallel Computing*, 1993.

[CFR+89] CYTRON, R., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. An efficient method for computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, Texas, January 11–13, 1989. ACM SIGACT and SIGPLAN.

[CFR+91] 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.

[CFS90a] CYTRON, R., FERRANTE, J., AND SARKAR, V. Experiences using control dependence in PTRAN. In GELERNTER, D., NICOLAU, A., AND PADUA, D., Eds., *Languages and Compilers for Parallel Computing*, pages 186–212. The MIT Press, Cambridge, Massachusetts, 1990.

[CFS90b] CYTRON, R., FERRANTE, J., AND SARKAR, V. Compact representations for control dependence. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 337–351, White Plains, New York, June 20–22, 1990. ACM SIGPLAN. Also in *SIGPLAN Notices*, 25(6), June 1990.

[CG93] CYTRON, R., AND GARSHBEIN, R. Efficient accomodation of may-alias information in SSA form. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 36–45, Albuquerque, NM, June 23–25, 1993. ACM SIGPLAN. Also in *SIGPLAN Notices*, 28(6), June 1993.

[CLR90] CORMAN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. The MIT Press; McGraw-Hill Book Co., Cambridge, New York, 1990.

[CLZ86] CYTRON, R., LOWRY, A., AND ZADECK, K. Code motion of control structures in high-level languages. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 70–85, St. Petersburg Beach, Florida, January 13–15, 1986. ACM SIGACT and SIGPLAN.

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

[CSS94] CHOI, J. D., SARKAR, V., AND SCHONBERG, E. Incremental computation of static single assignment form. Unpublished manuscript, 1994.

[DGS95] DUESTERWALD, E., GUPTA, R., AND SOFFA, M. L. Demand-driven computation of interprocedural data flow. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 37–48, January 1995.

[Ero95] EROSA, A. A goto-elimination method and its implementation for the McCAT C compiler. Master's thesis, McGill University, May 1995.

[Fon77] FONG, A. C. Generalized common subexpressions in very high languages. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 48–57, January 1977.

[FOW87] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Gup92] GUPTA, R. Generalized dominators and post-dominators. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 246–257, 1992.

[Gup95] GUPTA, R. Generalized dominators. *Information Processing Letters*, 53:193–200, 1995.

[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.

[Har85] HAREL, D. A linear time algorithm for finding dominators in flow graphs and related problems. In *Symposium on Theory of Computing*. ACM, May 1985.

[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.

[Joh94] JOHNSON, R. *Dependence Based Program Analysis*. PhD thesis, Cornell University, Ithaca, New York, August 1994.

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

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

[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.

[KU77] KAM, J. B., AND ULLMAN, J. D. Monotone data flow analysis frameworks. *Acta Informatica*, 7(3):305–317, 1977.

[LH88] LARUS, J. R., AND HILFINGER, P. N. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, Atlanta, Georgia, June 22–24, 1988. *SIGPLAN Notices*, 23(7), July 1988.

[LRF94] LEE, Y., RYDER, B. G., AND FIUCZYNSKI, M. E. Region analysis: A parallel elimination method for data flow analysis. In *Proceedings of IEEE 1994 International Conference on Computer Languages*, pages 31–42, May 1994. Toulouse, France.

[LT79] LENGAUER, T., AND TARJAN, R. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[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.

[MR90a] MARLOWE, T., AND RYDER, B. Properites of data flow frameworks: A unified model. *Acta Informatica*, 28:121–163, 1990.

[MR90b] MARLOWE, T. J., AND RYDER, B. G. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–196. ACM SIGACT and SIGPLAN, January 1990.

[PB95] PINGALI, K., AND BILARDI, G. *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.

[Pin95] PINGALI, K., 1995. Personal Communication.

[PM72] PURDOM, JR., P. W., AND MOORE, E. F. Algorithm 430: Immediate predominators in a directed graph. *Communications of the ACM*, 15(8):777–778, August 1972.

[PS89] POLLOCK, L., AND SOFFA, M. L. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 11(4), April 1989.

[Ram93] RAMALINGAM, G. *Bounded Incremental Computation*. PhD thesis, University of Wisconsin-Madison, Madison, Wisconsin, August 1993.

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

[RLP90] RYDER, B., LANDI, W., AND PANDE, H. Profiling an incremental data flow analysis algorithm. *IEEE Transactions on Software Engineering*, 16(2), February 1990.

[RMP88] RYDER, B. G., MARLOWE, T. G., AND PAULL, M. C. Conditions for incremental iteration: Examples and counterexamples. *Science of Computer Programming*, 11(1):1–15, October 1988.

[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.

[Ros94] ROSEN, B., 1994. Personal Communication.

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

[RP88] RYDER, B. G., AND PAULL, M. C. Incremental data-flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, January 1988.

[RR94] RAMALINGAM, G., AND REPS, T. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, January 1994.

[RT82] REIF, J. H., AND TARJAN, R. Symbolic program analysis in almost linear time. *SIAM Journal of Computing*, 11(1):81–93, February 1982.

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

[RWZ88] ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. Global value numbers and redundant computations. In *ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, pages 12–27, January 1988.

[SG94] SREEDHAR, V. C., AND GAO, G. R. Computing ϕ-nodes in linear time using DJ-graphs. Technical Report ACAPS Memo 75, School of Computer Science, McGill University, January 1994.

[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 to appear in Journal of Programming Languages.

[SGL95] SREEDHAR, V. C., GAO, G. R., AND LEE, Y. Efficient data flow analysis using DJ graphs: Elimination methods revisited. Technical Report ACAPS Memo 93, McGill University, June 1995.

[SS70] SHAPIRO, R. M., AND SAINT, H. The representation of algorithm. Technical Report CA-7002-1432, MCA, 1970.

[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.

[Ste93] STEENSGARD, B. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Microsoft Research, October 1993.

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

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

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

[TH92] TJIANG, S. W. K., AND HENNESSY, J. L. Sharlit—a tool for building optimizers. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 82–93, San Francisco, California, June 17–19, 1992. *SIGPLAN Notices*, 27(7), July 1992.

[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.

[WCES94] WEISE, D., CREW, R. F., ERNST, M., AND STEENSGAARD, B. Value dependence graphs: Representation without taxation. In *ACM SIGPLAN-SIGACT Symposium on the Principles of Programming Languages*, January 1994.

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

[Wol89] WOLFE, M. J. *Optimizing Supercompilers for Supercomputers*. Pitman, London and MIT Press, Cambridge, Massachusetts, 1989.

[WZ85] WEGMAN, M., AND ZADECK, K. Constant propagation with conditional branches. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 291–299. ACM SIGACT and SIGPLAN, January 1985.

Index