NODE LISTING BASED GLOBAL DATA FLOW ANALYSIS

B.E Project Report
Submitted in partial fulfillment of the requirements of the degree of

Bachelor of Computer Engineering
of
University of Pune

Rahul U. Joshi
Vinay V. Kakade
Medha G. Trivedi

Under the guidance of

Dr. Uday P. Khedker
Reader, Department of Computer Science
University of Pune

and

Prof. A. A. Sawant
(Internal Guide)

Department of Computer Engineering
Government College of Engineering

Shivajinagar, Pune – 411005, India

2000 – 2001

Government College of Engineering
Shivajinagar, Pune

Certificate

This is to certify that

- Rahul U. Joshi (Exam No. B2014221)
- Vinay V. Kakade (Exam No. B2014222)
- Medha G. Trivedi (Exam No. B2014242)

of B.E Computer Engineering class have submitted their project write-up on *Node Listing Based Global Data Flow Analysis*. This is a result of their own efforts in collecting information from various sources, assimilating it, deriving new information from experimentation and reasoning and presenting it in this format. We certify that they have completed the project work to our satisfaction.

Prof. V. K. Kokate	Prof. A. A. Sawant
Head of Department	Internal Guide

Contents

Abstract **ix**

Acknowledgments **xi**

1 Introduction **1**

1.1 Code Optimization 1
1.2 An Organization for an Optimizing Compiler 2
1.3 An Outline 2

I BACKGROUND AND KNOWN RESULTS **4**

2 Flow Graphs and Their Properties **5**

2.1 Basic Blocks 5
2.2 Flow Graphs 6

3 Data Flow Analysis **12**

3.1 Advantages of using Iterative Algorithms 12
3.2 Disadvantages of using Iterative Algorithms 13
3.3 Representative, Basic Data Flow Analysis Problems 13
3.4 Brief description of each problem 14
3.5 A Taxonomy 14
3.6 Iterative Algorithms 15

4 Introduction to Node Listing **16**

4.1 Depth First Ordering 16
4.2 Motivation of Node Listing 17
4.3 Some Known Results Regarding Node Listings 19
4.4 Node Listing Based Data Flow Analysis 20

4.5 Algorithm by Aho and Ullman 21
4.6 Heuristic Node Listings 21
4.7 Majority Merge Heuristics 23

II NEW RESEARCH AND THEORETICAL RESULTS 24

5 Density of a Graph 25
5.1 Definitions 25
5.2 Maximum Density and Length of the Node listing 26
5.3 Effect of Repetitions on the Flow Graph Structure 27
5.4 A Sureshot Method of Increasing the Density 28
5.5 Densities of Spiral Graphs 30
5.6 Two Methods of Finding Density 31

6 Maximal Reducible Flow Graphs 33
6.1 Motivation For Maximal Reducible Flow Graphs 33
6.2 Maximal Reducible Flow Graphs 37
6.3 Paths in Maximal Reducible Flow Graphs 43
6.4 Regions in Maximal Reducible Flow Graphs 45
6.5 Partial Ordering in Reducible Flow Graphs 48
6.6 Binary Parsable Reducible Flow Graphs 52
6.7 Weak Node Listings for Maximal Rfg's 53
6.8 Applications of Maximal Reducible Flow Graphs 54
6.9 Density, Node Listings and Maximal Rfg's 56

7 Properties of Spiral Graphs 57
7.1 Spiral Graphs are Maximal Reducible Flow Graphs 57
7.2 Properties of Spiral Graphs 59
7.3 Reduction of a Flow Graphs into a Spiral Graph 62
7.4 Algorithm For Computing Sequences of Regions 63

8 Miscellaneous Results 67
8.1 About Spiral Graphs 67
8.2 Generalization of Lemma 2 in Aho and Ullman 71
8.3 Some More Results 73

9 Minimal Reducible Flow Graphs 76

9.1 Transformations and Their Effect on Density 76
9.2 Minimal Reducible Flow Graphs 79

10 Conclusion and Future Work 81
10.1 Conclusion 81
10.2 Future Work 81

III SOFTWARE ENGINEERING 83

11 Organization of The Programs 84

IV IMPLEMENTATION DETAILS 86

12 Brute Force Implementation on PARAM 10000 Supercomputer 87
12.1 Sequential Brute Force Algorithm 87
12.2 Implementation with Parallel Virtual Machine 88
12.3 Implementation with Message Passing Interface 90
12.4 Some Optimization Heuristics 96

13 GUI Front End using GTK+/ GNOME 99
13.1 The Main Window 99
13.2 Working With Graphs 101
13.3 Programming Details 103

14 Graphical User Interface using Qt 104
14.1 About Qt 104
14.2 Qt Object Model 104
14.3 Signals and Slots 105
14.4 Meta Object Information 107
14.5 GUI for Node Listing Based Data Flow Analysis 108

15 Software Used 110

16 Program Manual 111
allbackpaths 113
acyclic 114
allpaths 115

brute . . . 116
depth . . . 117
depth-max . . . 118
dfnize . . . 119
dominance . . . 120
elimpaths . . . 121
Exe.sh . . . 122
graphinput . . . 123
heursitic . . . 124
ismax . . . 125
isspiral . . . 126
matrix . . . 127
max-RFG . . . 128
mmheuristic . . . 129
nl2b . . . 130
reduce . . . 131
reducible . . . 132
sheuristic . . . 133
spiral . . . 134
sub_of_spiral . . . 135
subgraph . . . 136
verify-elimpaths . . . 137
verifynl . . . 138
vernl-trie . . . 139

17 Algorithms 140

V REFERENCES 146

Appendix A - Some Referenced Papers 147
Node Listings Applied to Data Flow Analysis . . . 148
Node Listings for Reducible Flow Graphs . . . 165

Appendix B - Papers Written During the Project 177

Appendix C - PVM and MPI Function List 183

About the Project Report 184

References 186

Index 190

List of Algorithms

1 Division into basic blocks 6
2 Iterative Data Flow Analysis 17
3 Node Listing Based Data Flow Analysis 20
4 Heuristic Node Listing Constructor 22
5 Majority Merge 23
6 Finding the matrix of levels for max rfg 51
7 Computing the sequences of regions 64
8 Finding the sequences of regions 65
9 Sequential brute force algorithm 88
10 Master Program for MPI Implementation 94
11 Slave program for MPI Implementation 95
12 Skipping Redundant Permutations 96
13 Heuristics for reducing the number of prefixes 97
14 Finding the acyclic ordering of a flow graph 140
15 Checking reducibility by T_1-T_2 transformations 141
16 Generate all paths in a flow graph 142
17 Generate all paths in a flow graph that begin with a back edge 143
18 Simplified heuristic node listing 144
19 Dominator computing algorithm 145

Abstract

Data flow analysis is one of the important activities carried out by an optimizing compiler before applying any optimizations. It involves the collection of information about how the various data items in the program are defined and used. This information can then be used by the optimizer to apply various transformations. Global data flow analysis is performed at the level of the entire program rather than the basic blocks. Currently, the iterative methods are being widely used for performing this analysis. The iterative data flow analysis has a time complexity of $O((d + 2)n)$, where d is the depth of the flow graph. In the worst case $d = n - 1$, so that iterative data flow analysis in worst case is $O(n^2)$. Also, this method involves some extra overheads.

Ken Kennedy, in 1975, had proposed another approach towards global data flow analysis, in which we construct an intermediate representation of the flow graph, called *node listing* and then apply the data flow equations to the nodes in that order. The significance of this approach towards data flow analysis is that if the node listing for a flow graph can be found quickly, it will enable us to do the various kinds of global data flow analyses quickly. Shortly after that, in 1976, Al Aho and J.D.Ullman, gave a method of constructing node listings for reducible flow graphs. Their algorithm produced a node listing of length $n + 2.01n \log n$ in time $O(n \log n)$. Their method was based on the theme of converting a reducible flow graphs into spiral graph, finding the node listing of the spiral graph, and from that, derive the node listing of the original graph.

However, it has been experimentally observed that for all reducible flow graphs, the length of the minimal node listing is very small than the one proposed by Aho and Ullman. In particular, it has been seen that for every reducible flow graph of n nodes, there exists a node listing of length $n + n \log n$. To prove this result, we propose a new concept called *density of a graph*, denoted as δ, which is the maximum number of times a node appears in some node listing. Thus, if δ is the density of a graph, there exists a node listing for the graph of length $L \leq (\delta + 1)n$. We then propose to show that $\delta \leq \lfloor \log n \rfloor$ for any reducible flow graph, so that for all reducible flow graph of n nodes, there exists a node listing of length $n + n \log n$.

Towards proving this result, we have derived many intermediate results. We have developed the concept of a *maximal reducible flow graph* which are special type of reducible flow graphs of which every reducible flow graph is a subgraph. Thus, we can use these graphs to find or prove the upper or lower bounds on the properties of reducible flow graphs. *Spiral graphs* and *binary parsable flow graph* are special types of maximal reducible flow graphs for whom we have proved that $\delta \leq \log n$. Based on this result, we also have an "intuitive" yet informal proof for $\delta \leq \log n$ for all reducible flow graphs.

A study of the properties of maximal reducible flow graphs and some experimentation also brings out the significance of the *depth* and the *dominator tree* of a flow graph and motivates us to change

our original conjecture $\delta \leq \log n$ to $\delta \leq \log{(d+1)}$.

Given all the non-redundant acyclic paths in a flow graph, finding its density manually may not always give correct results, as a node listing with smaller density may exist. Brute force method remains the only choice and its is *computationally very hard* because good bounding functions are not known. So, we have implemented brute force programs to find the density of the given reducible flow graph on *PARAM 10000 Supercomputer* using the *Parallel Virtual Machine* and the *Message Passing Interface* libraries at the *Center for Development of Advanced Computing's National PARAM Supercomputing Facility*.

We have also written several papers regarding our findings and we wish to publish them in some journals.

Apart from these theoretical developments, we also have implemented a library of tools on a Linux system using such tools as *lex*, *yacc* and *shell scripts*. These tools are the implementations of known algorithms, the algorithms developed by us, as well as many "utility" programs. These tools help us with experimenting with flow graph and carrying out common operations. For user friendliness, we have also provided a GUI front end to these tools using both the *Qt library* as well as the *GTK+/GNOME library*.

❑❑❑

Acknowledgments

First and foremost, we would like to express our indebtedness to **Dr. Uday Khedker** Reader, Department of Computer Science, University of Pune for guiding us, allowing us to explore new ideas and allowing us to participate in his research. Secondly, we would like to thank our internal guides, **Prof. A. A. Sawant** and **Prof. P. P. Kajave** for taking keen interest in the project and helping us from time to time. We would like to thank **Prof. V. K. Kokate**, Head of the Department of Computer Engineering, Govt. College of Engineering, Pune for taking interest and allowing us to do this project.

We would also like to thank the **Center for Development of Advanced Computing (CDAC)** for allowing us to use the **PARAM 10000 Supercomputer** at the **National PARAM Supercomputing Facility** for implementing brute force programs to verify our results. In particular, we wish to thank **Dr. P. K. Sinha** for allowing us to carry out the experiments, **Dr. Sunder Rajan** for guiding us and **Dr. Anbarasu** for helping us implement the brute force program on a Parallel Virtual Machine. We would like to thank **Mrs. Akshara** for helping us re-implement the programs using the Message Passing Interface.

We would also like to thank **Dr. A. A. Diwan, Dept. Of Computer Science, Indian Institute of Technology (IIT), Powai** for guiding us and discussing the problem with us.

We would like to thank **Jennifer Kilcoyne** and **Chris Smith** of the **Academic Press** for allowing us to print the reference papers from the **Journal of Computer and Systems Science** in our report.

Last but not the least, we would like to thank all the people involved in the development of Linux and other free software that we used heavily in our project. We would also like to thank the members of the *Pune Linux User's Group* mailing list for solving our difficulties from time to time.

Once again our deepest gratitude to all mentioned above and those whom we might have unknowingly forgotten to mention.

Rahul U. Joshi
Vinay V. Kakade
Medha G. Trivedi

CHAPTER 1

Introduction

In this introductory chapter, we will see what data flow analysis is and how it is useful for performing various program optimizations. More information on these optimizations can be found in [5].

1.1 Code Optimization

Ideally, compilers must produce code that is as good as can be written by hand. However, this goal is very difficult to achieve. However, the code produced by straightforward compiling can made to run faster or take less space, or both by applying certain program transformations called *optimizations* . Compilers that apply such code improving transformations are called *optimizing compilers*. The optimizations that are applied by such compilers can be divided into two categories,

Machine Dependent Here, the specific characteristics of the underlying machine for which the code is generated are taken into consideration.

Machine Independent Here, the transformations are applied without considering any properties of the target machine.

The best program transformations are those that yield the most benefit for the least efforts. Some of the properties that the program transformations must have are

1. It must preserve the meaning of the program.
2. It must speed up the program by a measurable amount.
3. It must be worth the effort i.e. a long and complex optimization that speeds up the program by only a small amount is not worth applying.

1.2 An Organization for an Optimizing Compiler

The code improvement phase of a compiler consists of *control flow analysis* and *data flow analysis* followed by the application of the transformations, as shown in Figure 1.1

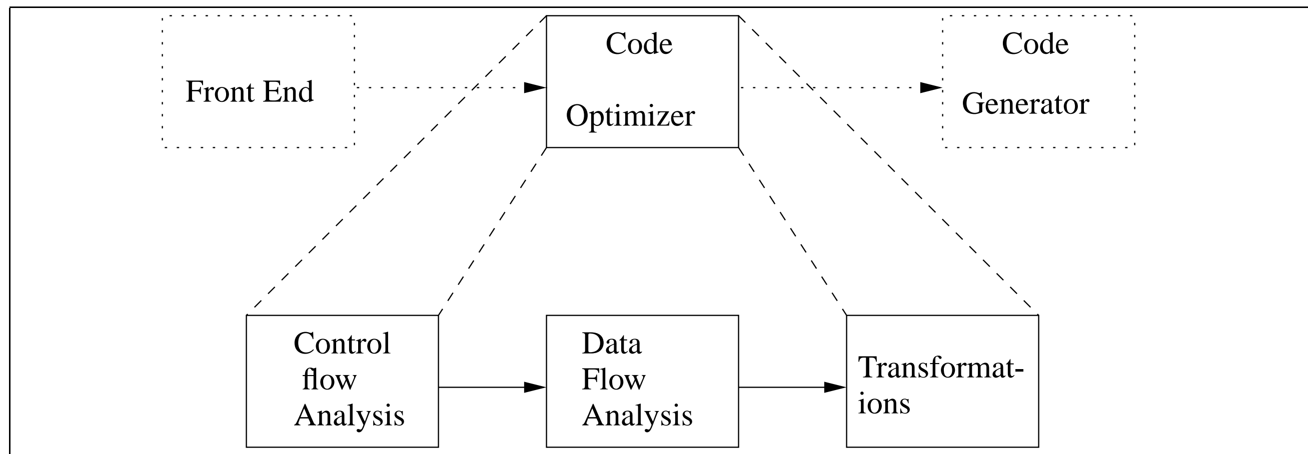


Figure 1.1: Organization of the code optimizer

Control flow analysis traces the patterns of possible execution in a program. For this purpose, a program is represented by a *control flow graph* or simply a *flow graph*. In a flow graph, the edges indicate the flow of control and the nodes represent the basic blocks, as will be discussed later. The construction, structure, representation and properties of such graphs is a part of control flow analysis.

Data flow analysis traces the possible definitions and uses of data along the potential control flow paths and collects the information about certain attributes of the data items. For example, data flow analysis of a program may indicate that for all possible paths the program takes, the value of a certain variable say i at a certain point in the program is always 1. This fact can then be used by the optimizer to speed up the code. Data flow analysis is generally performed by solving a system of simultaneous equations. If the analysis is performed by looking only at the statements in a basic block, it is called *local data flow analysis*. On the other hand, if the data flow analysis is performed by looking at all the the basic blocks in a program, it is called *global data flow analysis*. Normally, local data flow analysis is performed before the global data flow analysis. There are many types of data flow analysis like *unidirectional* and *bidirectional*, *inter-procedural* and *intra-procedural*. Furthermore, data flow analysis can be performed by using various methods viz. *iterative, exhaustive, incremental* etc. We will be discussing these terms in later chapters.

1.3 An Outline

This project report is divided into five parts. The first part aims at introducing data flow analysis and node listing. It contains an explanation and definitions of commonly used concepts in data flow analysis and it also explains the known algorithms for data flow analysis. It describes the node listing approach to data flow analysis and the algorithm due to Aho and Ullman. The second part describes the research that we have done during the project and the results that we have obtained. The third part deals with software engineering. The fourth part contains details of various implementations that

were carried out during the project. The fifth part contains some of the papers that we referred during the project.

Chapter 2 describe *flow graphs* and states some of their properties of our interest. Chapter 3 described *data flow analysis* in detail. It also explains the terms such as *forward*, *backward*, *unidirectional* and *bidirectional* data flow analysis. It also explains the *iterative*, *incremental*, *inter-procedural* and *intra-procedural* data flow analysis. Chapter 4 explains the *node listing* approach for data flow analysis, its significance and current limitations. It also briefly describes the algorithm given by Aho and Ullman for constructing the node listings for reducible flow graphs.

Chapter 5 introduces the concept of the *density* of a graph and explains various results that we have derived regarding the densities of reducible flow graphs. Chapter 6 describes *maximal reducible flow graphs* that we developed during the project. Chapter 7 described various properties of spiral graphs in relation to the maximal reducible flow graphs. It also describes a new algorithm for the reduction of a flow graph into a spiral graph. Chapter 8 is a cornucopia of some smaller but important results that were found during the project. Chapter 9 described *minimal reducible flow graphs* that are basically a motivation towards grouping all the flow graphs having the same density. Chapter 10 concludes and also examines what more can be done towards continuing the research.

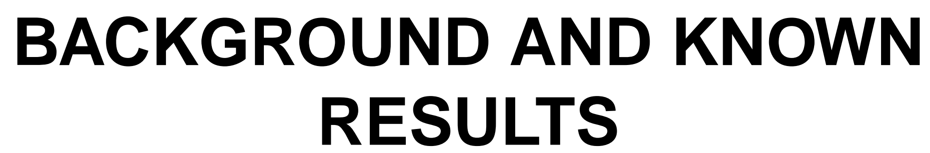
Chapter 11 comments about the organization of the programs developed during the course of the project and the tools used therein.

Chapter 12 describes the details of the brute force implementation for finding exact densities of flow graphs. This implementation was carried out on the *PARAM 10000 Supercomputer* at CDAC using the *Parallel Virtual Machine* and the *Message Passing Interface* libraries. Chapter 15 lists the various softwares that were used during the project. Chapter 16 provides a detailed description of all the programs developed and serves as a manual. Chapter 17 shows some of the algorithms that were developed and implemented. Chapter 14 describes the GUI front end for the programs that was developed using the Qt Library under Linux and a demonstration of the programs developed. Chapter 13 describes another GUI front end developed using the GTK+/GNOME libraries.

Appendix A includes some of the papers that we referred during the project. Appendix B presents the papers that were written during the project. Finally, we list all the books, papers, reports etc. that were some time or the other used during the project.

❑❑❑

I



BACKGROUND AND KNOWN RESULTS

CHAPTER 2

Flow Graphs and Their Properties

In an optimizer, a program is represented as a *flow graph*. In this chapter we will see what these flow graphs are and also examine some properties and algorithms concerning flow graphs.

2.1 Basic Blocks

A *control flow graph* or simply a *flow graph* is a graph representation of the intermediate code generated by the intermediate code generator. Nodes in the flow graph represent computations whereas the edges represent the flow of control. A flow graph is extensively uses as a vehicle for collecting information about the intermediate program.

Definition 2.1 (Basic Block) A *basic block* is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branch except at the end.

Thus, we can see that a basic block is nothing but any sequence of program statements not containing transfer of control instructions (i.e. goto's) except at the end. Algorithm 1 can be used to partition a sequence of intermediate code statements into basic blocks,

Example 2.1 For the intermediate code in Figure 2.1, the leaders are the statements (1), (2), (3), (4), (6) and (7). Therefore the basic blocks are $\{(1)\}$, $\{(2)\}$, $\{(3)\}$, $\{(4), (5)\}$, $\{(6)\}$ and $\{(7)\}$.

```
(1) if a <= b goto (3)
(2) max := a
(3) goto (5)
(4) max := b
(5) if c <= max goto (7)
(6) max := c
(7) return max
```

Figure 2.1: Division into Basic Blocks

Input A sequence of intermediate code statements
Output A list of basic blocks
Method

1. We first determine the set of *leaders*, the first statements of basic blocks. The rules we use are the following.
   (a) The first statement is a leader.
   (b) Any statement that is the target of a conditional or unconditional goto is a leader.
   (c) Any statement that immediately follows a goto or conditional goto statement is a leader.
2. For each leader, its basic block consists of the leader and all statements upto but not including the next leader or the end of the program.

Algorithm 1: Division into basic blocks

2.2 Flow Graphs

We can add the flow-of-control information to the set of basic blocks making up a program by constructing a directed graph called a *flow graph* [5, p. 532]. The nodes in the flow graph are the basic blocks. One node is distinguished as *initial*; it is the block whose leader is the first statement. There is a directed edge from block B_1 to block B_2 if B_1 immediately follows B_2 in some execution sequence; that is, if

1. there is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2.
2. B_2 immediately follows B_1 in the order of the program, and B_1 does not end in an unconditional jump.

We say that B_1 is a *predecessor* of B_2 and B_2 is a *successor* of B_1.

When a program is converted into a flow graph, the jump statements at the end of a block are made to point to the block rather that to the actual quadruples. Secondly, in a flow graph, an edge from block B to block B' does not specify the condition under which the control flows from B to B'.

2.2.1 Dominators

The dominator relationships are one of the most important characteristics of flow graphs that enable us to detect loops apart from many other things.

Definition 2.2 (Dominators) A node d in a flow graph *dominates* node n, written as d *dom* n, if every path from the initial node of the flow graph to n goes through d.

From the above definition, it can be easily seen every node dominates itself. Also, it can be seen that the initial node n_0 dominates all the nodes in the flow graph. Finding the dominator relationship

in a flow graph involves some kind of data flow analysis. An algorithm for finding dominators in given in Chapter 17 (Algorithm 19).

A useful way of presenting the dominator information of a flow graph is in a tree, called the *dominator tree*. In a dominator tree,

1. The initial node is the root of the tree.
2. Each node d dominates only the descendents in the tree.

The existence of the dominator trees follows from a property of dominators; each node n has a unique *immediate dominator* m that is the last dominator of n on any path from the initial node to n, that is, the immediate dominator m has the property that if $d \neq n$ and d *dom* n, then d *dom* m.

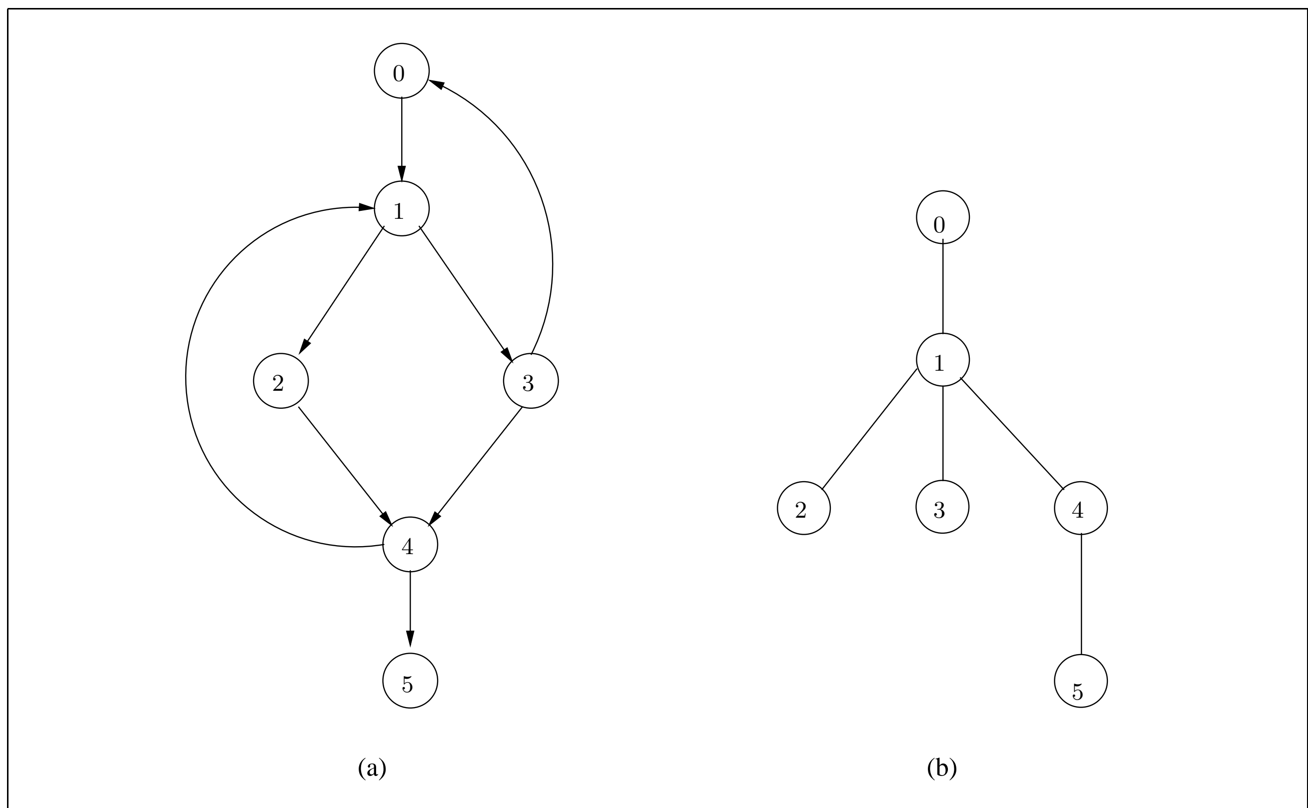


Figure 2.2: Flow Graph and its dominator tree

Example 2.2 Figure 2.2–a shows a flow graph and Figure 2.2–b shows its dominator tree. The initial node 0 dominates all the other nodes and hence it is the root of the dominator tree. Node 1 dominates node 2 since control can flow to node 2 only after passing through node 1. Similarly, node 1 also dominates nodes $3, 4$ and 5. Nodes 2 and 3 dominate only themselves. Node 4 dominates node 5.

Definition 2.3 (Back Edge) An edge in a flow graph is called a *back* edge if its head dominates its tail.(If $n \rightarrow m$ is an edge in a flow graph, then m is the head and n is the tail).

Thus, if $n \rightarrow m$ is a back edge if m dominates n. For example, in the flow graph of Figure 2.2, the edge $4 \rightarrow 1$ is a back edge as 1 dom 4. Similarly the edges $3 \rightarrow 0$ and $5 \rightarrow 3$ are back edges.

Observation

1. Since the initial node n_0 dominates all the other nodes in the flow graph, all the edges in the flow graph of the form $n \to n_0, n \neq n_0$ are back edges.

2. Since each node dominates itself, self loops of the form $n \to n$ are back edges.

Definition 2.4 (Forward Edge) An edge in a flow graph is called a *forward* edge if it is not a back edge.

Observation Any edge from the initial node to some other node i.e. any edge of the form $n_0 \to n, n \neq n_0$ is a forward edge since n does not dominate n_0.

The dominator relationship can be used to find the loops in a flow graph. From the principles of structured programming, we know that a loop must have a single entry point, called its *header*. This entry point dominates all the nodes in the loop. Also, there must be at least one way to iterate the loop i.e. at least one path back to the header. The loops in a flow graph can be defined in terms of back edges.

Definition 2.5 (Natural Loop) Given a back edge $n \to d$, we define the *natural loop* of the edge to be d plus the set of nodes that can reach n without going through d. Node d is the header of the loop.

2.2.2 Reducible Flow Graphs

Reducible flow graphs [32] are special types of flow graphs.Almost all the flow graph that one encounters in practice are *reducible*. In particular, use of *structured programming* principles results in program whose flow graphs are reducible. Any program not containing goto's will be reducible. One important property of reducible flow graph is that there are no jumps into the middle of loops, the only entry into a loop is through its *header*. There are many equivalent definitions of reducible flow graphs. We will see them one by one.

Definition 2.6 (Reducible Flow Graph) A flow 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.

2. The back edges consists only of edges whose heads dominates their tails.

Thus, if a flow graph is reducible, then removing all the back edges in the flow graph gives us an acyclic graph. This condition can be used to test for reducibility of flow graphs.

Example 2.3 Consider the flow graph in Figure 2.2. Removing the back edges $3 \to 0, 4 \to 1$ and $5 \to 3$, it can be easily seen that the resulting flow graph is acyclic. Hence the flow graph in Figure 2.2 is reducible.

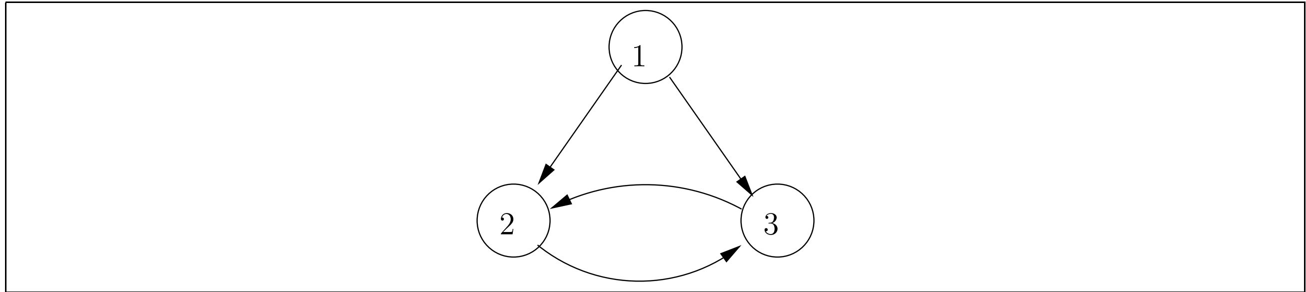


Figure 2.3: A non reducible flow graph

Example 2.4 As another example, consider the flow graph in Figure 2.3. Here 1 is the initial node, so edges $1 \rightarrow 2$ and $1 \rightarrow 3$ are forward edges. Now, neither 2 dominates 3 not does 3 dominates 2. So both the edges $2 \rightarrow 3$ and $3 \rightarrow 2$ are also forward edges. Thus, all the edges in this graph are forward edges. Clearly, this graph has a cycle $(2, 3, 2)$. Hence the graph is not reducible.

The flow graph in Figure 2.3 is not reducible because the nodes 2 and 3 form a loop and one can enter this loop either through the edge $1 \rightarrow 2$ or $1 \rightarrow 3$ i.e. the loop is not a single entry loop.

We now state some more properties of reducible flow graphs [32].

2.2.3 T_1 - T_2 Analysis

We first define the two transformations as follows

Definition 2.7 (T_1 Transformation) If n is a node in a flow graph with a loop i.e. an edge $n \rightarrow n$, then a T_1 *transformation* is defined as the deletion of the loop.

Definition 2.8 (T_2 Transformation) If there is a node n, not the initial node, that has a unique predecessor m , then a T_2 *transformation* is defined as the consumption of node n by node m, that is, delete node n and make all successors of n (including m possibly) as successors of m.

We now state an equivalent definition of reducibility [32].

Result 2.1 *A flow graph is reducible if and only if it can be transformed into a single node by repeated applications of T_1 and T_2 transformations.*

Example 2.5 Figure 2.4 shows how a flow graph is reduced into a single node by repeated applications of T_1 and T_2 transformations. (a) is the original flow graph. Node d has the only predecessor c, so we apply a T_2 transformation to get (b). Then we remove the self loop on the node cd by applying a T_1 transformation to get (c). Now b has the only predecessor a, so applying a T_2 transformation gives (d). Now cd has the only predecessor ab, so again applying a T_2 transformation gives (e). Thus, we have transformed the original graph into a single node by applying T_1 and T_2 transformations. Hence, the flow graph in (a) is reducible.

We now state a theorem regarding irreducible flow graphs [32]. We first define a ($\star$)-flow graph.

Definition 2.9 (($\star$)-flow graph) A ($\star$)-flow graph is defined as any of the flow graphs represented in Figure 2.5, where the dashed lines denote node disjoint (except for endpoints, of course) paths; nodes a, b, c, and n_0 are distinct, except that a and n_0 may be the same.

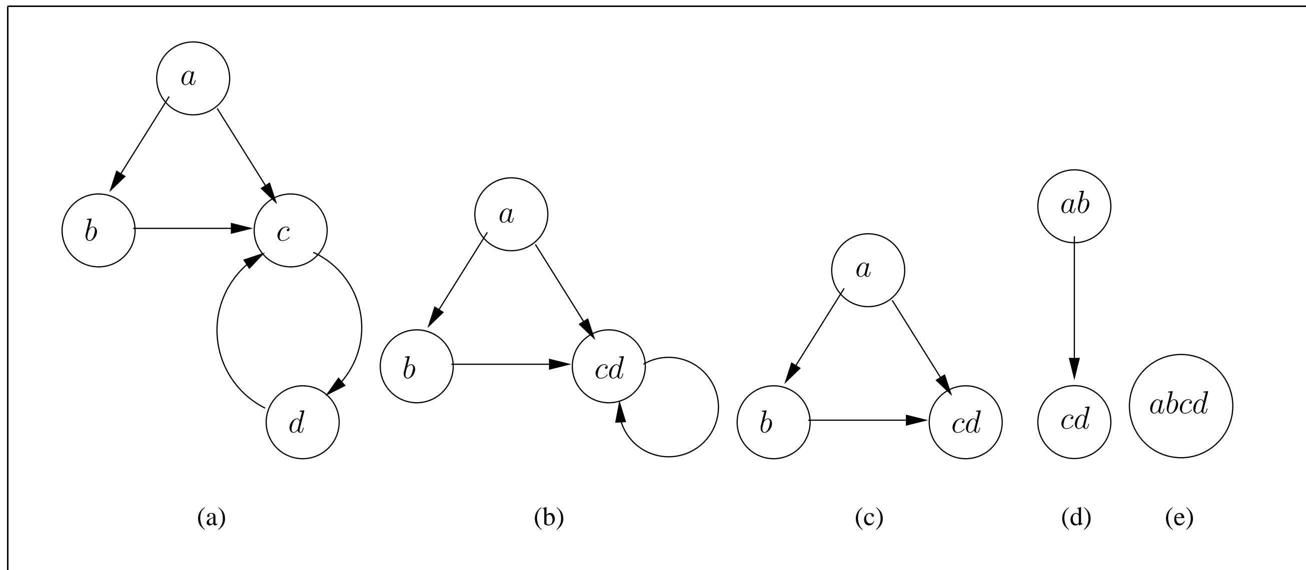


Figure 2.4: T_1 - T_2 Analysis

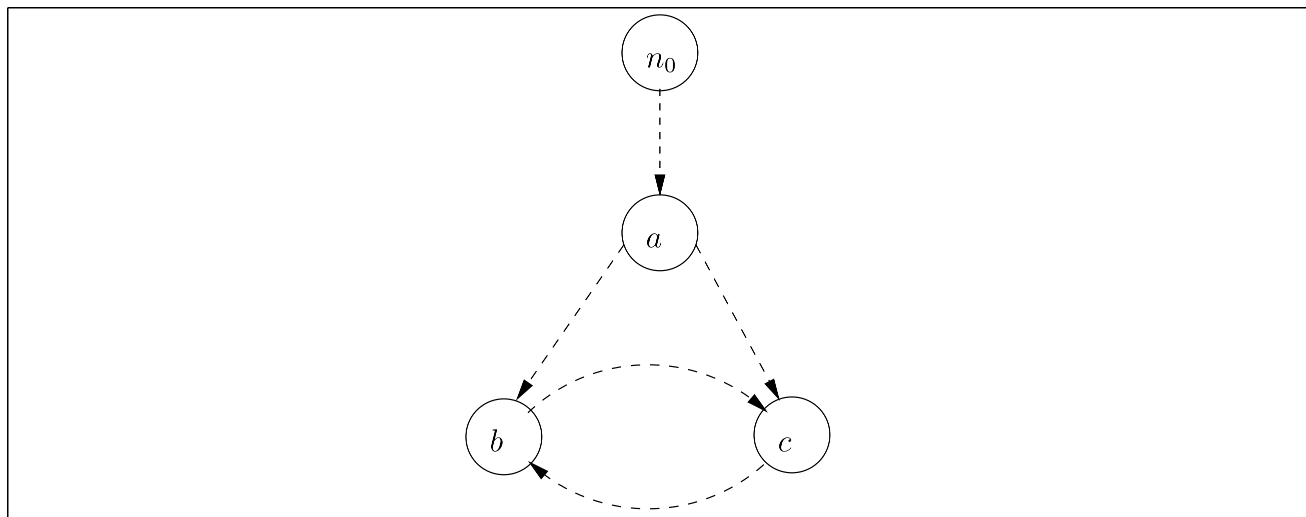


Figure 2.5: The (✶)-subgraph

Result 2.2 (The (✶)-Characterization Theorem) *A flow graph is irreducible if and only if it contains (✶).*

2.2.4 Regions

The division of a flow graph into regions serves to put an hierarchical structure on a flow graph.

Definition 2.10 (Region) We define a portion of a flow graph called a *region* to be a set of nodes N that include a *header*, which dominates all other nodes in the region.

Another equivalent definition of a region from [4] is:

Definition 2.11 (Region) A *region with header* h of a flow graph $G = (N, E, n_0)$ is a set of nodes N' and edges $E' \subseteq N' \times N'$, with h in N', such that if $m \rightarrow n$ is in E, then

1. if n is in N' and $n \neq h$, then m is in N', and
2. if m and n are in N' and $n \neq h$, them $m \rightarrow n$ is in E'.

That is, the only way to enter a region from outside is through its header.

We now state the definition of the dag of a flow graph [32].

Definition 2.12 (DAG of a Flow Graph) A *dag of a flow graph* $G = (N, E, n_0)$ is an acyclic flow graph $D = (N, E', n_0)$ such that $E' \subseteq E$ and for any edge e in $E - E'$, $(N, E' \cup \{e\}, n_0)$ is not a dag. That is, D is a maximal acyclic sub-flow graph.

The following result regarding DAG of a reducible flow graph is from [32].

Result 2.3 *If $G = (N, E, n_0)$ is a reducible flow graph, then $D = (N, E - B, n_0)$ is a dag of G, where B is the set of back edges in G.*

The reason for dividing a flow graph into regions is to impose a hierarchical structure on the flow graph. Formally, this structure is defined in terms of the "parse" of a flow graph. We first state the following result from [4].

Result 2.4 *Let $R = (N', E')$ be a region of some flow graph G represented by some node during the reduction of G, with N' not a singleton. Then N' can be partitioned into two nonempty disjoint sets of nodes N_1 and N_2, such that (N_1, E_1) and (N_2, E_2) are regions, where $E_1 = E' \cap N_1 \times N_1$ and $E_2 = E' \cap N_2 \times N_2$.*

Thus every non singleton region can be divided into two regions. We call this division as "parse" in this report, though the actual definition is somewhat different. Thus a parse is nothing but a sequence showing how a region and its parses are further parsed.

❑❑❑

CHAPTER 3

Data Flow Analysis

Iterative algorithms are used for practical intra-procedural data flow analysis.There are several variations of iterative algorithms. The following three are the most common:

1. Worklist version
2. Round Robin version
3. Node listing version

The worklist versions maintain a set containing "work-to-be-done" that is initialized, updated on-the-fly as the algorithm executes and eventually exhausted. The worklist contains information to be propagated whose "influence" may not have been recorded yet. Nodes may be "visited" in an arbitrary order.

The round-robin version propagates information by starting with an initial estimate of the desired information to nodes by repeatedly visiting the nodes in a round-robin fashion until information flow stabilizes(i.e., a fixed point is reached).

The Node listing version first preprocesses the flow graph to obtain a list of nodes (with repetitions, in general) then propagates information by visiting nodes in the order in which they occur on the list. The node listing has the property that visiting nodes in the indicated order suffices to propagate information.

3.1 Advantages of using Iterative Algorithms

1. Worklist and round-robin versions of the iterative algorithms apply to *all* known data flow analysis.
2. These algorithms are very easy to program. No graph reductions are necessary, as in interval analysis algorithm and Ullman's algorithm. Therefore the iterative algorithms are oblivious to the reducibility of the underlying flow graph.

3.2 Disadvantages of using Iterative Algorithms

1. The disadvantage of the node listing version of the iterative algorithm is that the preprocessing necessary to compute a good node listing is usually nontrivial [4, 22].

2. The most important undesirable quality of the worklist and round-robin versions is the fact that the worst case time complexity of these algorithms is not good.

   For example, with sparse reducible flow graphs on "bit vector problems" these algorithms require $O(n^2)$ bit vector steps in the worst case, whereas Ullman's algorithm requires at most $O(n\log n)$ bit vector steps.

3. Another undesirable quality is that because the iterative algorithms do not analyze the underlying flow graphs, nothing is known about the loop structure of the graph, should this information be desired.

Nevertheless, the ease of programming and generality make the iterative algorithm excellent for practical use when a flow graph is necessary, until of course another solution is discovered!

The sections to continue describe in brief variations of the iterative algorithm applied to a class of very simple data flow analysis problems that are called *"bit vector frameworks"* .
Note : Considering just bit vector frameworks is sufficient because

1. it simplifies the exposition in that these problems are easier to understand than more general problems.

2. such problems do occur often enough to justify a separate treatment.

3.3 Representative, Basic Data Flow Analysis Problems

There is an important subclass in intra-procedural data flow analysis problems each of which can be formulated as a collection of set equations, reminiscent of equations for conservation of flow and the sets involved with each flow graph node, each bit position corresponds to a program variable (or expression), and a bit indicates nonexistence or possible existence of an attribute of that variable (or expression) at that node.

There are four representative problems of this subclass.These problems are called

1. "available expressions"

2. "reaching definitions"

3. "live variables"

4. "very busy expressions"

These problems are very similar in that almost any algorithm to solve one of these problems can, with slight modification, be used to solve the other problems.

The attention is restricted to intra-procedural rather than inter-procedural problems. For this reason we assume that for for the procedure under scrutiny :

1. there is a control flow graph
2. all relevant local data flow information is available
3. all variable aliasing is known, can be handled, and thus can be ignored and
4. the procedure is isolated in that we may ignore where and how it is called.

3.4 Brief description of each problem

3.4.1 Available Expressions (AE)

An expression such as $X + Y$ is *available* at a point p in a flow graph $G = (N, A, s)$ iff every sequence of branches that the program may take to p cause $X + Y$ to have been computed after the last computation of X or Y. By determining the set of available expressions at the top of each node in G, we know which expressions have already been computed prior to each node. Thus, we may be able to eliminate the redundant computation of some expressions within each node.

3.4.2 Reaching Definitions (RD)

A *definition* of a variable x is a statement that assigns, or may assign, a value to x. A definition d *reaches* a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path. Intuitively, if a definition d of some variable a reaches point p, then d might be the place at which the value of a used at p might last have been defined.We kill a definition of a variable a if between two points along the path there is a definition of a.

3.4.3 Live Variables (LV)

A number of code improving transformations depend on the information computed in the direction opposite to the flow of control in a program. In *live variable* analysis we wish to know for variable x and point p whether the value of x at p could be used along some path in the flow graph starting at p. If so, we say x is *live* at p; otherwise x is *dead* at p.

3.4.4 Very Busy Expressions (VBE)

An expression e is *very busy* at a point p in a flow graph iff it is always used before it is killed. This means that no matter what path is taken from p, the expression e will be evaluated before any of its operands are defined.

3.5 A Taxonomy

The following table summarizes the four types of basic data flow analysis problems.

	Set Intersection, "and" problems	Set union, "or" problems
Top–Down Problems (Operation over predecessors)	Available Expressions (AE)	Reaching Definition (RD)
Bottom–Up Problems (Operation over successors)	Very Busy Expressions (VBE)	Live Variables (LV)

The AE and VBE use the set intersection operation and require a largest solution, whereas, RD and LV use the set union operation and require a smallest solution.For AE and RD the operation is over predecessors, whereas, for VBE and LV it is over successors. We call AE and RD *top down* (or forward) *problems* because information is propagated in the same direction as control flow to solve these problems. Conversely, we call VBE and LV *bottom up* (or backward) *problems* because information must be propagated in the opposite direction of control flow to solve these problems.

3.6 Iterative Algorithms

3.6.1 Worklist Version

There are three worklist versions of the iterative algorithm.

1. the segregated version,
2. the integrated version, and
3. Kildall's version.

In the segregated version we process one expression at a time.In the integrated version and Kildall's version we intermix the processing of expressions, but in slightly different ways. Detailed algorithms of each version is beyond scope of this project report.

3.6.2 Round-Robin Version

This version uses bit vectors and the bit vector operations *bvand* and *bvor* .Instead of maintaining a worklist, we repeatedly visit each node in round-robin fashion and propagate $0's$ forward for AE ($1's$ backward for LV). The algorithm terminates when one iteration fails to change any bit of any bit vector.It follows the "iterate-until-stabilization" paradigm.

❑❑❑

CHAPTER 4

Introduction to Node Listing

In order to give a more efficient data flow analysis method, **Ken Kennedy** in 1975 proposed a new approach towards global data flow analysis [22]. In his paper, he also conjectured a node listing constructor with time complexity $O(n \log n)$[1]. Immediately after that, an $O(n \log n)$ node listing constructor was given by **Aho** and **Ullman** [4]. The length of the node listing was $n + 2.01n \log n$. In this chapter, we introduce the node listing based approach to data flow analysis and see how it can lead to efficient incremental data flow analysis of flow graphs.

4.1 Depth First Ordering

In all the problems of data flow analysis considered before, it can be observed that any *event of significance at a node will be propagated to that node along an acyclic path.* Thus, any cyclic path does not contribute towards the data flow analysis [5, p 672]. If all the useful information propagates along acyclic paths, we have an opportunity to tailor the order in which we visit nodes in iterative data flow algorithms so that after relatively few passes through the nodes, we can be sure that the information has passed along all the acyclic paths.In particular, if we apply the data flow equations to the nodes in the depth first order, the number of iterations required are bounded by the *depth* of the graph. In that case, the algorithm 2 gives a general algorithm for iterative data flow analysis.

It can be observed in this case that if d is the depth of the graph, then $d+1$ iterations are sufficient to propagate the data flow values along acyclic paths. However, the above algorithm requires one more pass to detect the fact that all the data flow variables have been propagated. Thus, if d is the depth of the graph, then the data flow analysis can be performed in time $O((d + 2)n)$. In the worst case, $d = n - 1$ and hence in general, data flow analysis by iterative methods still remains an $O(n^2)$ problem.

It can be seen that there are two major areas of inefficiencies in the iterative approach towards data flow analysis [22].

1. First, an extra pass through the program is required to discover that none of the data flow

[1] All logarithms are to the base 2 unless otherwise stated

```
1:  for i := 0 to n − 1 do /* Initialize the data flow variables */
2:     Initialize in() and out() for block i ;
3:  end for
4:  change := true ;
    /* Apply equations till the in's and out's stabilize */
5:  while change == true do
6:     change := false ;
7:     for each block B in depth first order do
8:        apply data flow equations to block B ;
9:        update change;
10:       if there is a change in in() or out() of block B then
11:          change := true ;
12:       end if
13:    end for
14: end while
```

Algorithm 2: Iterative Data Flow Analysis

variables change. This extra pass and the testing for changed data flow variables (sets) on each pass results in a lot of unnecessary work that can be avoided if we could somehow know when to halt the iteration.

2. Second, iteration over every node in each pass seems to be unnecessary. The problem is to iterate exactly enough times to transmit information along any acyclic path in the program.

The node listing method of data flow analysis attempts to overcome both the above inefficiencies of the iterative data flow analysis method. The node listing is an intermediate representation of the flow graph that facilitates the propagation of data flow variables along the acyclic paths.

4.2 Motivation of Node Listing

To gain some motivation behind the node listing, consider the flow graph given in Figure 4.1–a.

The depth d of the graph in Figure 4.1–a. is 3. As a result, using the iterative data flow analysis algorithm requires 5 iterations over all the nodes in the flow graph. If however, the data flow equations are applied to the nodes of this flow graph in the order $(1, 2, 3, 4, 3, 2, 1)$, then data flow analysis would be correctly performed and fewer than 2 iterations over the graph will be required. Thus, the node listing specifies the order in which the equations are to be applied to the nodes of the flow graph.

To formally define a node listing, we first define simple and basic paths in a flow graph.

Definition 4.1 (Simple Path) A *simple* path in a flow graph is a path that does not include the same edge twice. That is, a simple path is nothing but an acyclic path.

Definition 4.2 (Basic Path) A *basic* path in a flow graph is a simple path $(x_1, x_2, \ldots, x_k)$, such that there is no shorter simple path from x_1 to x_k which is a subsequence of $(x_1, x_2, \ldots, x_k)$.

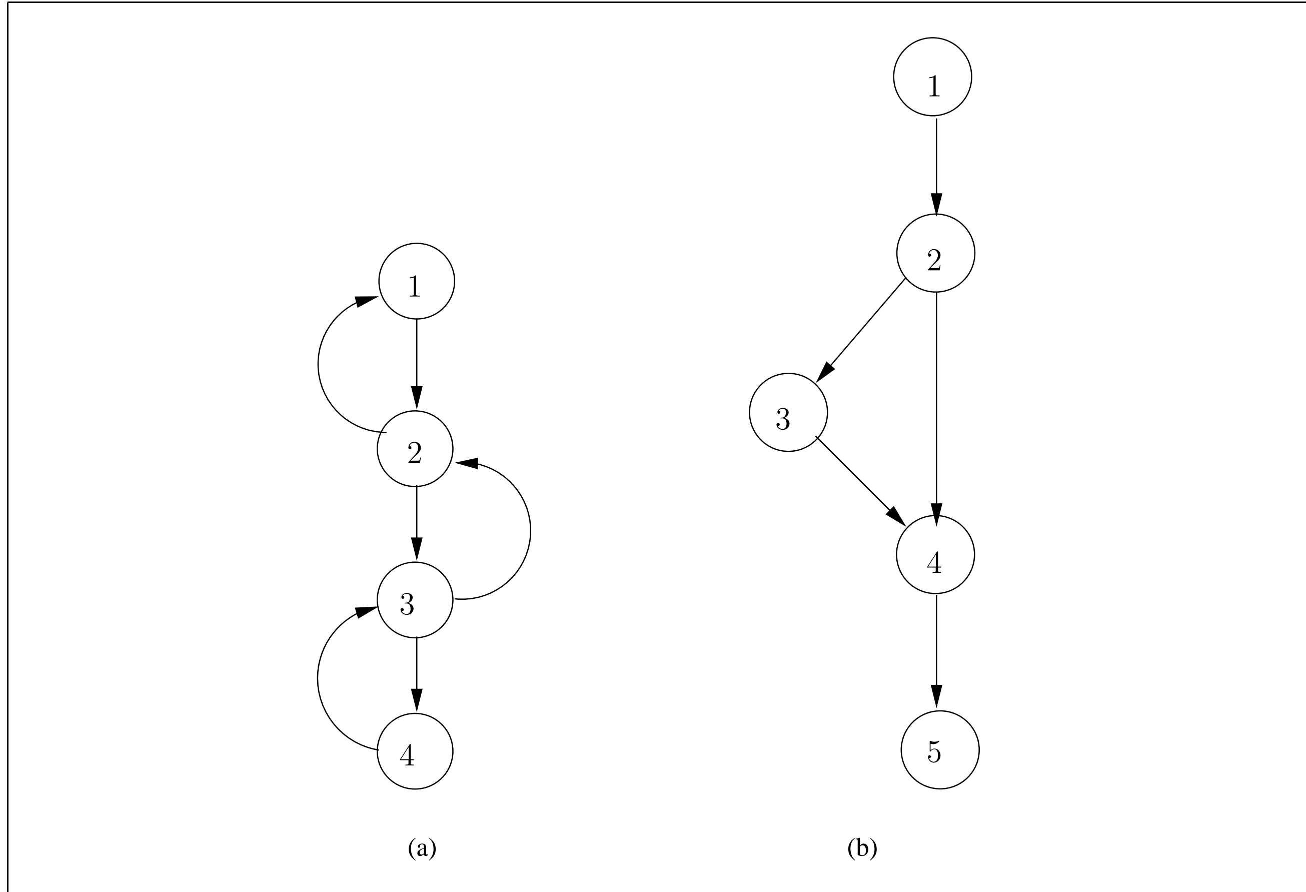


Figure 4.1: Example Flow Graphs

Example 4.1 In the flow graph shown in Figure 4.1–b the path $(1, 2, 3, 4, 5)$ is a simple path. The path $(1, 2, 4, 5)$ is also a simple path. The path $(1, 2, 3, 4, 5)$ is *not* a basic path because there is a simple path $(1, 2, 4, 5)$ which is a subsequence of $(1, 2, 3, 4, 5)$. The path $(1, 2, 4, 5)$ is a basic path.

It can be seen that in case of "there exists" problems like live variables and reaching definitions, propagation of the data flow variables along the basic paths is sufficient. However, in case of "for all" problems like available expressions, propagations along the basic paths is not sufficient, we need to propagate the values along all the simple paths in the flow graph.

4.2.1 Strong Node Listing

Definition 4.3 (Strong Node Listing) A *strong node listing* for a flow graph $G = (N, E, n_0)$ is a sequence $(n_1, n_2, \ldots, n_m), m \geq n$, of nodes from N, in which nodes may be repeated more than once, such that all the simple paths in G are (not necessarily contiguous) subsequences thereof.

Example 4.2 For the flow graph in Figure 4.1, $(1, 2, 3, 4, 3, 2, 1)$ is a strong node listing.

Definition 4.4 (Minimal Node Listing) A strong node listing for a flow graph is called *minimal* iff there is no shorter node listing for that flow graph.

The following three results about strong node listings are known

- Every reducible flow graph has a node listing of length at most $n + 2.01n \log n$.
- There exists reducible flow graphs for which no strong node listing is shorter than $\frac{1}{2}n \log n$.
- A strong node listing for a reducible flow graph with $e = O(n)$, e being the number of edges in the graph, can be constructed in $O(n \log n)$ time.

4.2.2 Weak Node Listing

Definition 4.5 (Weak Node Listing) A *weak node listing* for a flow graph $G = (N, E, n_0)$ is a sequence $(n_1, n_2, \ldots, n_m), m \geq n$, of nodes from N, in which nodes may be repeated more than once, such that all the basic paths in G are (not necessarily contiguous) subsequences thereof.

Since every basic path is also a simple path, every strong node listing for a flow graph is also a weak node listing.

For the purpose of data flow analysis, it can be seen that for such problems as live variables and reaching definitions, wherein propagation along basic paths is sufficient, a weak node listing suffices for data flow analysis. However, for such problems as available expressions, wherein propagation along all the simple paths is necessary, a strong node listing is necessary to perform the data flow analysis. As a result, henceforth, we will concentrate only on strong node listings. Henceforth, when we mean node listing, it is implicitly understood to be a strong node listing.

4.3 Some Known Results Regarding Node Listings

In his paper "Node Listings Applied to Data Flow Analysis," Ken Kennedy [22] gives many results regarding the node listings for reducible flow graphs. In this section, we state some of them that are of our interest.

Result 4.1 *Let $G = (N, E, n_0)$ be an acyclic flow graph, that is, there does not exists a sequence $(x_1, \ldots, x_k)$ of nodes in G such that $x_1 = x_k$ and $(x_i, x_{i+1}) \in E, 1 \leq i < k$. Then there exists a node listing of length $|N|$ for G. Furthermore, this node listing contains every path, not merely the basic paths.*

Observation The above result really says that for acyclic flow graph, the length of the node listing if $|N|$ and that a node listing for an acyclic flow graph can be obtained by applying a "topological" sort to the flow graph. Also, if the flow graph is not acyclic, then after the removal of the back edges, it becomes acyclic and the node listing for such an acyclic flow graph is also called its *acyclic ordering*.

Result 4.2 *For every reducible flow graph of n nodes, there exists a node listing of length $(d + 1)n$ where d is the depth of the graph*

Proof:

In the previous section, we have seen that if during the data flow analysis, the nodes are visited in depth first order, then $d + 2$ iterations are sufficient for the data flow analysis. Of these, one iteration is used just to discover that the data flow analysis is complete, and hence it is an overhead. Therefore,

in reality, only $d+1$ iterations are necessary for data flow analysis. Hence, if we repeat the depth first ordering of the reducible flow graphs $d+1$ times, the resulting sequence covers all the acyclic paths in the flow graph. This exactly is the node listing with length $(d+1)n$. □

Result 4.3 *For every acyclic reducible flow graph, there exists a node listing in which each node appears exactly once.*

Proof:
It can be easily seen that a topological sort [54] of an acyclic flow graph will have all the acyclic paths in the flow graph as its subset. Thus, the topological sort of the acyclic graph is its node listing. □

From a lemma in [4], we know that any acyclic path that enters a region R from a node outside the region cannot traverse any back edge. Thus, it can be easily seen that if a path enters a region from outside, it traverses a subsequence of the topological sort of the flow graph obtained after the removal of the back edges. We define this as acyclic ordering.

Definition 4.6 (Acyclic Ordering) If G is a reducible flow graph, then its *acyclic ordering* is defined as any topological sort of the flow graph obtained from G by removing all the back edges.

We follow the convention of indicating the acyclic orderings by writing a “hat”i.e. if A is the node listing for a certain flow graph, then its acyclic ordering will be denoted as $\hat{A}$. Since, in general, an acyclic graph can have more than one topological sort, there are, in general, more than one acyclic orderings for a reducible flow graph.

4.4 Node Listing Based Data Flow Analysis

Once we have obtained the node listing for the given flow graph, performing the data flow analysis is quite simple. We simply traverse the node listing, applying the data flow equations to each node as we visit it. Algorithm 3 shows how data flow analysis is done using the node listing method.

```
1: Find the node listing, say L, for the flow graph ;
2: for i := 0 to n − 1 do /* Initialize */
3:    Initialize in() and out() for node i ;
4: end for
5: for each node x in L do /* Iterate */
6:    Apply the data flow equations to node x ;
7: end for
```

Algorithm 3: Node Listing Based Data Flow Analysis

It can be seen that the node listing based algorithm has the following advantages over the iterative algorithm:

1. The algorithm makes sufficient number of passes over the nodes in the flow graph so that all data flow variables are stabilized. Extra pass just to detect that data flow variables have now stabilized is not required, since we now know when to stop the process (the end of the node listing).

2. We don't have to keep a track of the changes in in() and out() of the various nodes in the flow graph. Thus, the space for variable "change" along with the overhead of storing the old values of in() or out() and checking for a change is avoided.

Thus, it can be seen that node listing based variation of the data flow analysis is indeed much more efficient than the iterative algorithm. However, this method requires the construction of the node listing of the flow graph which is not trivial. In fact, currently known algorithms take much longer time ($O(n \log n)$) to construct the node listing. It is this initial pre-processing overhead because of which the node listing method is not used in practice. Also the iterative algorithm is much more simpler to implement.

There are two variants of this node listing algorithm [31]. They are as follows,

1. For "there exists" problem in which the confluence operator is set union (such as Reaching Definitions and Live Variable), a weak node listing is sufficient to perform the data flow analysis. On the other hand, for "for all" problems in which the confluence operator is set intersection, (such as Available Expressions and Very Busy Expressions), a strong node listing is required to perform the data flow analysis.

2. For *forward* data flow problems (such as Available Expressions), we apply the data flow equations to each node in the node listing in their actual order i.e. the node listing is processed as it is. However for *backward* data flow problems (such as Live Variables), the equations are applied to the nodes in the node listing in reverse order i.e. the node listing is processed in reverse order.

4.5 Algorithm by Aho and Ullman

Aho and Ullman [4] have given an algorithm for finding the node listings for reducible flow graphs. This algorithm has a time complexity of $O(n \log n)$ and produces a node listing of length bounded by $n + 2.01n \log n$. In this section, we briefly describe this algorithm. Detailed information about it can be found in [4], which has been included in this report in Appendix A.

4.6 Heuristic Node Listings

In this section, we will explain a heuristic algorithm for finding the node listings of reducible flow graphs. Note that the time complexity of the algorithm is very large i.e. $O(n2^n)$. However, these algorithms are suitable for experimentation with graphs having limited number of nodes.

We know that if the acyclic ordering of a reducible flow graph is repeated $d + 1$ times, the resulting sequence of nodes is a node listing. This is the basic principle behind the heuristic node listing constructor. We first define a *span* in an acyclic path.

Definition 4.7 (Span) If P is a acyclic path in a reducible flow graph, then a *span* is defined as a sequence S of nodes in P having the following properties:

1. the nodes in S are consecutive in P i.e. S is a *proper* subsequence of P,

2. the nodes in P are in acyclic order, and

3. S is not a subsequence of any other span in P.

In simple words, if we partition an acyclic path such that the nodes in each part are connected by forward edges and the part themselves are connected by back edges, then each of these parts in a span. It is easy to see that if an acyclic path contains d back edges, then it can be partitioned into $d+1$ spans. This is illustrated as follows,

$$\underbrace{S_{(1,1)}, S_{(1,2)}, \ldots, S_{(1,k_1)}}, \underbrace{S_{(2,1)}, S_{(2,2)}, \ldots, S_{(2,k_2)}}, \ldots, \underbrace{S_{(d+1,1)}, S_{(d+1,2)}, \ldots, S_{(d+1,k_{d+1})}}$$

Here, each underbrace indicates a span with span i having k_i nodes and $d+1$ such spans. $S_{(i,j)}$ is the jth node in the ith span. The spans are connected by back edges so that each edge of the form $S_{(i,k_i)} \rightarrow S_{(i+1,1)}$ is a back edge.

Now to find the heuristic node listing for a given reducible flow graph, we first find all the acyclic paths in that flow graph and divide each of the paths into spans. The node listing consists of $d+1$ levels, one for each span, each level being a set. When adding the path P to the partially constructed node listing, we add the first span in P to level 1 in the node listing, add the second span in P to level 2 and so on. In other words, we add each span to the corresponding level. After all the paths are processed, we list the nodes in level 1 in their acyclic ordering, followed by those in level 2 and so on. The heuristic node listing constructor is showing in Algorithm 4.

1: Let $NL_i, 1 \leq i \leq d+1$ be an initially empty set for level i ;
2: **for** Each acyclic path P in G **do** /* Process each acyclic path */
3: d := depth of the path P ;
4: **for** i := 1 to $d+1$ **do** /* Process each span */
5: S := Span i of path P ;
6: NL_i := $NL_i \cup S$;
7: **end for**
8: **end for**
/* Print the node listing */
9: **for** i := 1 to $d+1$ **do**
10: print nodes in NL_i in acyclic order ;
11: **end for**

Algorithm 4: Heuristic Node Listing Constructor

In the program "Exe" given by Dr. Khedker as well as the program *heuristic*, each acyclic path P in the flow graph is processed in the *reverse* order i.e. the highest span is first added, then the next highest span and so on. There is really no necessity of this reverse processing as it does not always reduce the length of the heuristic listing. Also, the processing of the paths in the reverse order increases the complexity of the program.

As a result, we have devised a new algorithm for finding the heuristic node listing which we call as the *simplified heuristics*. This is a straightforward algorithm which processes each path in the normal order, adding the lowest span first, then the next one and so on. This simplifies heuristic has been implemented as *sheuristic* and the wrapper shell script *Exe.sh*.

4.7 Majority Merge Heuristics

In this section, we state another algorithm for finding the node listings for flow graphs. This is a general algorithm and does not depend upon the properties of the graph for finding the node listing. It is a heuristic for a general *Shortest Common Supersequence* problem. In fact, the problem of constructing a node listing is the problem of constructing a shortest common supersequence where the strings are the paths in the graph.

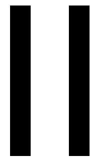
The *Majority Merge* algorithm builds a node listing starting from an empty node listing as follows: It looks for the first node of every path in the graph, appends the most frequent node say n to the partially constructed node listing and then removes node n from the front of the paths. This process is repeated until all the paths in the graph are exhausted. Thus, the algorithm for majority merge is as in Algorithm 5.

```
1:  L = φ;
2:  while all paths not exhausted do
3:     for all nodes n do
4:        frequency(n) = 0;
5:     end for
6:     for all paths p do
7:        Increase the frequency of the front node of p by 1;
8:     end for
9:     maxNode = Node with the maximum frequency;
10:    add node maxNode to the end of the node listing L;
11:    for all paths p with maxNode as the front node do
12:       Delete the front node of p;
13:    end for
14: end while
```

Algorithm 5: Majority Merge

This majority merge algorithm has been implemented as the program *mmheuristics* and works as well as the other heuristics.

❑ ❑ ❑



NEW RESEARCH AND THEORETICAL RESULTS

CHAPTER 5

Density of a Graph

In this chapter, we will introduce the concept of the density of a graph, methods of finding out density of the given graph and implications of the density towards lengths of node listings.

5.1 Definitions

Definition 5.1 (Density of A Node Listing) The *density of a node listing* L is defined as the maximum number of times a node is repeated in that node listing and is denoted as δ_L.

Thus, if $C_0, C_1, \ldots, C_{k-1}$ is the number of times nodes $n_0, n_1, \ldots, n_{k-1}$ respectively appear in L, then

$\delta_L = max(C_0, C_1, \ldots, C_{k-1}) - 1.$

Definition 5.2 (Density of a Graph) The *density of a graph* G is defined as the minimum of the densities of all the node listings for that graph.

Thus, if $(L_0, L_1, \ldots, L_k)$ is the set of all the node listings of a flow graph G and the respective densities of these node listings are $(\delta_0, \delta_1, \ldots, \delta_k)$, then the density δ of G is given by,

$\delta = min(\delta_0, \delta_1, \ldots, \delta_k)$

It can be easily seen that if the density of a graph is δ, then there exists a node listing for that graph in which some node appears $\delta + 1$ times and no node appears more that $\delta + 1$ times. In the worst case, it may be the case that all the n nodes in the graph appear $\delta + 1$ times in the node listing. Thus, if δ is the density of a graph G, then there exists a node listing of G whose length is,

$L \leq (\delta + 1)n$

Thus, the density of a graph gives us an upper bound on the length of the node listing. Many a times, not all nodes in the flow graph appear $\delta + 1$ times in the node listing. So the actual length of the node listing may be less than the upper bound of $(\delta + 1)n$.

5.2 Maximum Density and Length of the Node listing

First, let us see an extract from the paper "Node Listings applied to Data Flow Analysis" by K. Kennedy [22]. It states that,

Result 5.1 *For any flow graph there exists a node listing of length* $\leq n^2$ *where* $n = |N|$*, where* N *is the set of nodes of the flow graph.*

Proof:

Suppose $x_1, x_2, \ldots, x_n$ be all the nodes of the graph. Then,

$l = (x_1, \ldots, x_n, x_1, \ldots, x_n, \ldots, x_1, \ldots, x_n)$

with n repetitions of $(x_1, \ldots, x_n)$ is certainly a node listing. □

We make the above result stronger by the following statement:

Result 5.2 *For any (reducible or irreducible) flow graph, there exist* n *node listings (where* n *is the number of nodes in the flow graph). These node listings are:*

- $x_n, (x_1, \ldots, x_n), (x_1, \ldots, x_n), \ldots, (x_1, \ldots, x_n)$
- $(x_1, \ldots, x_n), x_{n-1}, (x_1, \ldots, x_n), \ldots, (x_1, \ldots, x_n)$
  ...
  ...
  ...
- $(x_1, \ldots, x_n), (x_1, \ldots, x_n), \ldots, (x_1, \ldots, x_n), x_1$

In all these node listings, there are $n - 1$ *repetitions of* $(x_1, \ldots, x_n)$*. So, the maximum length of node listing is* $n^2 - n + 1$*. Also, maximum value of density of graph is* $n - 1$*.*

Proof:

1. We consider only complete graph with n nodes as it will have the maximum possible length of the node listing as well as the maximum possible density amongst the graphs with n nodes.

2. In a complete graph with n nodes, each acyclic path will consist of exactly n nodes and there will be $n!$ such paths (including the paths starting with forward edges).

3. For each acyclic path except the path $(x_n, \ldots, x_1)$, there exist two consecutive nodes (x_j, x_k) such that $j \neq k$, $1 \leq$ (index of node x_j in that acyclic path) $\leq n - 1$, and (x_j, x_k) is a subsequence of $(x_1, \ldots, x_n)$.

4. So, all acyclic paths, except the path $(x_n, \ldots, x_1)$ is covered in the list of nodes

   $L = (x_1, \ldots, x_n), \ldots, (x_1, \ldots, x_n)$,

   where there are $n - 1$ repetitions of $(x_1, \ldots, x_n)$.

5. To cover the path $(x_n, \ldots, x_1)$, either the node x_n or node $x_{n-1}, \ldots$ or node x_1 is to be inserted at appropriate position in L. Depending on which node is inserted, one of the above mentioned node listing will be generated. The length of the node listing will be $n^2 - n + 1$, and density of graph will be $n - 1$.

Hence proved. □

Result 5.3 *If d is the depth of a reducible flow graph, then the density of that flow graph is $\delta \leq d$.*

Proof:
We know that if we repeat the depth first order (i.e. the acyclic order) of a reducible flow graph $d + 1$ times, then the resulting sequence is a node listing for the flow graph. In this sequence, each node appears $d + 1$ times. So, each node is repeated d times. Hence, for reducible flow graphs, the density (δ) is given by

$$\delta \leq d.$$

□

5.3 Effect of Repetitions on the Flow Graph Structure

Result 5.4 *If the structure of a flow graph is repeated twice, with the appropriate addition of back edges and the forward edges, the density may not increase.*

Proof:
Right now, no proof is available for the statement 'the Density will never increase in the above case'. But many counter examples are available for the statement 'the Density will always increase in the above case'. One of the counter examples is:

```
1 2 ;
2 3 ;
3 4 ;
4 5 8 ;
5 6 ;
6 4 7 ;
7 5 8 ;
8 2 9 ;
9 10 13 ;
10 11 ;
11 9 12 ;
12 10 13 ;
13 3 .
```

□

Result 5.5 *If the structure of a flow graph is repeated four times, with the appropriate addition of back and forward edges, the density may increase.*

Proof:
Right now, no proof is available for the statement 'the Density will always increase in the above case'. But an examples is available for supporting the statement 'the Density may increase in the above case'. The example is:

```
0 1 ;
```

```
1 2 ;
2 3 ;
3 1 4 5 ;
4 2 5 ;
5 1 6 10 ;
6 7 ;
7 5 8 9 ;
8 6 9 ;
9 1 10 ;
10 1 11 ;
11 12 ;
12 13 ;
13 11 14 15 ;
14 12 15 ;
15 10 11 16 ;
16 17 ;
17 15 18 19 ;
18 16 19 ;
19 1 11 .
```

□

5.4 A Sureshot Method of Increasing the Density

5.4.1 Some definitions

Definition 5.3 (RNL) *A restricted node listing (RNL)* for a region R is a sequence of nodes in R (with possible repetitions) such that every acyclic path in R that ends at an exit node of R is a subsequence thereof. (More precise definition needed)

Definition 5.4 (ARNL) *An acyclic restricted node listing (ARNL) for a region R is an RNL of R such that the acyclic ordering of R is a subsequence thereof.*

Thus, for every region R, we associate with it:

- An Acyclic Ordering (AO)
- A Restricted Node Listing (RNL)
- An Acyclic Restricted Node Listing (ARNL)

Result 5.6 *Given a minimal node listing for a reducible flow graph G, for each node n :*

1. *There exists at least one acyclic path beginning with n such that while mapping that path in the node listing, we must map n to the first occurrence of n in the node listing.*

   Proof:
   If there does not exist any path in which the first occurrence of n is not mapped, then removing the first occurrence of n from the node listing yields a shorter node listing. This contradicts the fact that the given node listing is minimal. □

2. *When mapping all acyclic paths beginning with node n, we can always map the starting node n to the first occurrence of n in the node listing.*

5.4.2 A Sureshot Method of Increasing the Density

Result 5.7 *From a graph G of density $\delta(\delta > 0)$, we can obtain a graph of density $\delta + 1$, if we replace each node of G by a region having RNL in which at least one node appears twice.*

We can construct the node listing for the new graph from the minimal node listing of the old graph by:

1. *Replacing each node that occurs exactly once by its ARNL.*

2. *For a node that appears more than once, replace the very first occurrence of the node by its ARNL, unless the node is the first node in the node listing, in which case we replace it by its RNL, and all other occurrences of the node by its AO.*

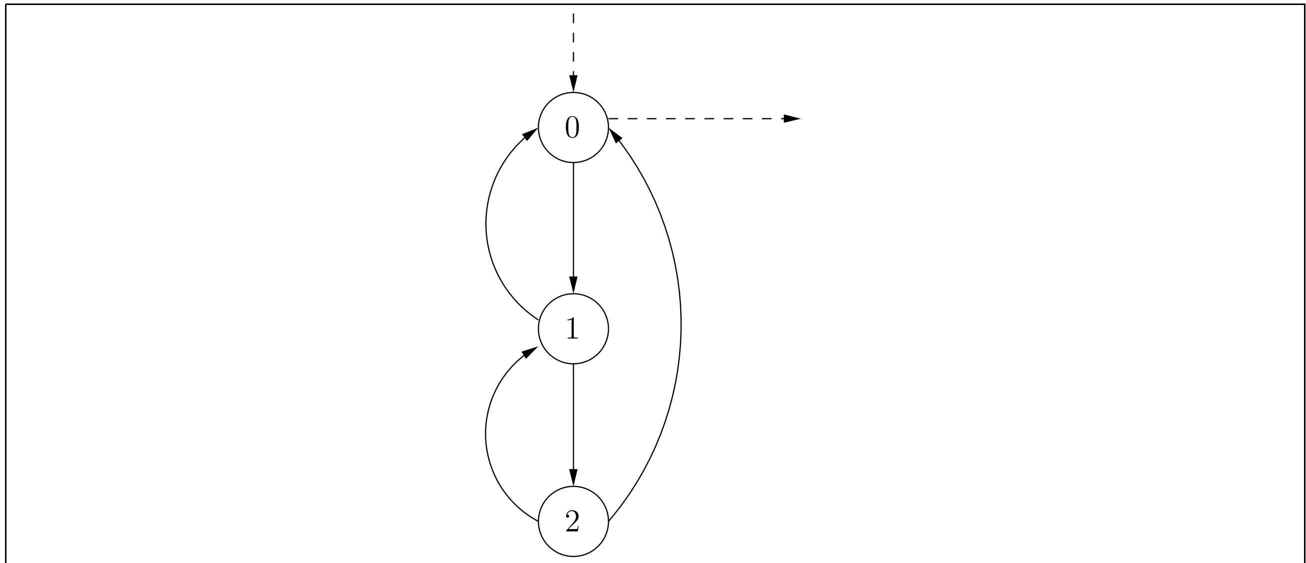


Figure 5.1: A "region" with RNL having two repetitions

Example 5.1 Figure 5.1 shows a "region" with a restricted node listing in which node 1 appears twice. This region has a single "exit" node viz. node 0 . The various "listings" associated with the region are

ARNL 0 1 2 1 0

RNL 1 2 1 0

AO 0 1 2

Thus, replacing each node is a graph of density δ with the above region will give us a graph of density $\delta + 1$.

5.5 Densities of Spiral Graphs

It is experimentally observed that for a spiral graph of n nodes in which all the nodes are added using rule (2b), the density is given by the expression $\delta = \lceil log_2 n \rceil$

Result 5.8 *For a $2b$-spiral graph having n nodes, the density is $\lceil \log n \rceil$, where n is the number of nodes.*

Proof:
We define a $2b$-Spiral Graph of n nodes as a Spiral Graph of n nodes in which all the nodes are added using the rule $2b$, as mentioned in [4]. We prove this statement by induction.

Basis of Induction
The above statement is true for $n = 1$ to 9. The densities of these graphs are as shown in following table.

n	Density	$\lceil \log n \rceil$
1	0	0
2	1	1
3	2	2
4	2	2
5	3	3
6	3	3
7	3	3
8	3	3
9	4	4

Induction Step
Assume that a $2b$-Spiral Graph having $n - 1$ nodes has the density $\lceil \log n - 1 \rceil$.

Hence
For a spiral graph having n nodes, say G, in which nodes are added in order $n - 1, \ldots, 0$, we can split the graph into 2 graphs, each having approximately $n/2$ nodes and each is a $2b$-spiral graph. The first graph G' has the nodes added in order $n - 1, \ldots, \lceil n/2 \rceil$ and the second G'' has the nodes added in order $\lfloor n/2 \rfloor, \ldots, 0$.

Let

$A' =$ acyclic ordering of nodes in G'

$B' =$ acyclic ordering of nodes in G''

$A =$ node listing of G'

$B =$ node listing of G''

Then $AB'A'B$ is always a node listing of G, as it can be easily seen that this covers all the acyclic paths. This is illustrated in the Figure 5.2 □

Thus, it can be seen that if a reducible flow graph has a subgraph which is a spiral graph of k nodes, the density (δ) of that reducible flow graph is $\delta \leq \lceil \log k \rceil$

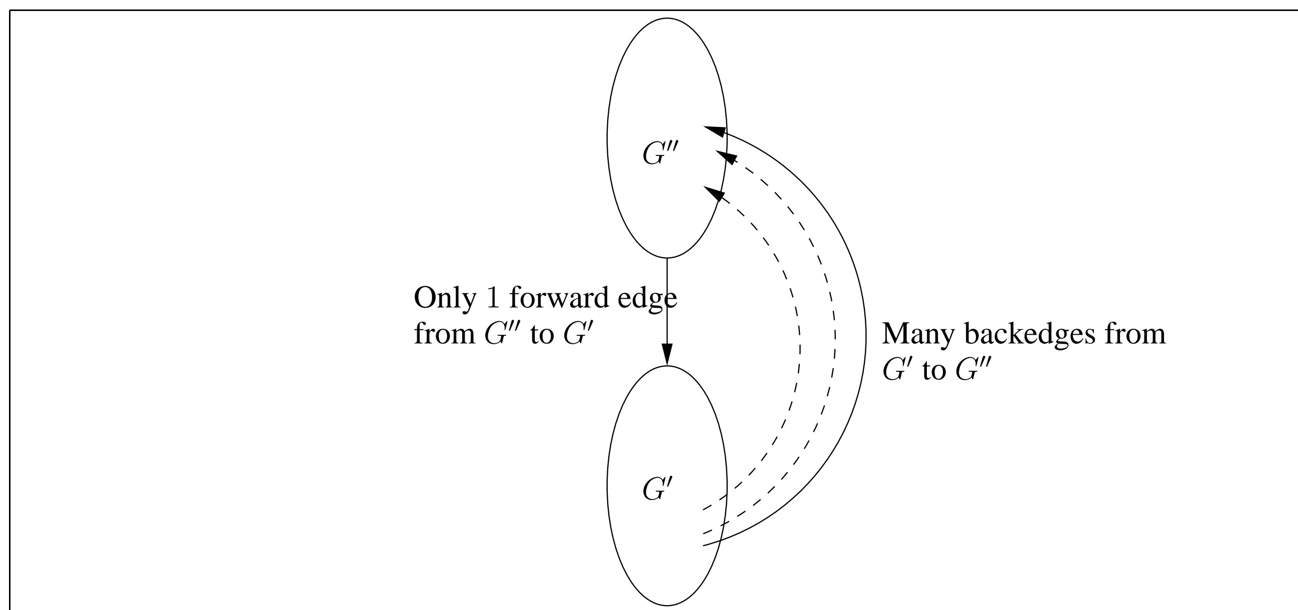


Figure 5.2: A $2b$-spiral graph split into two subgraphs: G' and G''

Result 5.9 *For any reducible flow graph, having n nodes, the upper bound of density is given by* $\lceil \log n \rceil$.

Proof:
Among all maximal rfgs of n nodes, the density of the $2b$-spiral graph will the largest, as it has the maximum number of acyclic paths, which cover all the nodes and also there are maximum number of back edges.

As stated in the previous result, the density of $2b$-spiral graph having n nodes is $\lceil \log n \rceil$. So, the upper bound of the density on any reducible flow graph is $\lceil \log n \rceil$. □

5.6 Two Methods of Finding Density

To find the density (δ) of any reducible flow graph (or for that matter *any* graph), we can proceed by the following two methods

- Assume an "initial" traversal in the acyclic order over the graph. This "initial" traversal is used to initialize the data flow variables. Now, since this "initial" traversal contains the part of every acyclic path in the flow graph up to but not including the first back edge in the path, we now need to consider only those acyclic paths in the flow graph that begin with a back edge. We find all the acyclic paths in the flow graph that begin with a back edge, ignore the first nodes in these paths (as these nodes will be covered in the initial traversal) and then try to "fit" the path so that we get minimum repetitions of nodes. We call the density of a flow graph obtained by this method as *higher density* of the flow graph and denote it by δ_h. It is observed that in case of spiral graphs with all nodes added using rule (2b), the higher density is given by

  $\delta_h = \lceil \log n \rceil$

  However, there seems to be a "catch" in this method. If we keep aside the "initial" traversal for initialization, then the information will not flow along the initial acyclic parts of the paths. This

may give *wrong data flow analysis*. The only remedy to this problem seems to be to keep the initialization of data flow variables a *separate* step from the initial traversal. Only then can we guarantee that correct data flow analysis will take place.

- As we saw earlier, it seems that initialization of data flow variables *cannot* be comfortable incorporated into the initial traversal of the graph. Initialization necessarily must be a separate step carried out prior to the data flow analysis. In that case, we can *get rid of the initial traversal*. That is, we find all the acyclic paths in the given flow graph (both that begin with a back edge and those that do not begin with a back edge). Then we try to "fit" the paths so as to get minimum repetitions of nodes. We call the density of the flow graph obtained by this method as the *lower density* of the flow graph and denote it as δ_l. It is observed that in case of spiral graphs in which all the nodes are added using rule (2b), the lower density of the graph is given by

  $\delta_l = \lfloor \log n \rfloor$
  $\Rightarrow \delta_l \leq \log n.$

  Also, it is observed that in case of a spiral graph in which all the nodes are added using rule (2b), if the number of nodes is 2^n, then

  $\delta_l = \delta_h = n$

❑❑❑

CHAPTER 6

Maximal Reducible Flow Graphs

In the paper "Node Listing for Reducible Flow Graphs," [4] Aho and Ullman define a special type of a reducible flow graph called the *spiral graph*. At first, it seems that spiral graph is the "largest" possible reducible flow graph for the given number of nodes since the addition of any additional edge to a spiral graph renders it irreducible. So, it may appear that every reducible flow graph of n nodes is a subgraph of some spiral graph of n nodes. In the next section, we examine the exact relationship between reducible flow graphs and spiral graphs. In particular, we show that there exists reducible flow graphs that are not subgraphs of any spiral graphs. Also, in further sections, we show that apart from spiral graphs there exists many other reducible flow graphs that are "largest" i.e. addition of any additional edge renders them irreducible. We then formalize this notion in terms of *maximal reducible flow graphs* and examine their properties in later sections. These graphs serve as a new method of synthesizing reducible flow graphs that is general and is based on the dominator tree of the flow graph. These graphs were discovered and their properties studied by Rahul U. Joshi.

6.1 Motivation For Maximal Reducible Flow Graphs

6.1.1 An Auxiliary Result

Result 6.1 *If a reducible flow graph $G = (N, E, n_0)$ is a subgraph of another reducible flow graph $G' = (N, E', n'_0)$ and $R' = (N'_R, E'_R, h_R)$ is a region in G', then the subgraph of G containing nodes N'_R and all the edges between them is also a region with header h_a, that is, $R = (N'_R, N'_R \times N'_R \cap E, h_a)$ is a region.*

Proof:
Let us assume that $R = (N'_R, N'_R \times N'_R \cap E, h_a)$ is not a region. Therefore, there exist at least two nodes, say $n_a, n_b \in N'_R$, such that there is an edge from a node outside R' to n_a and n_b, that is, there exists edges $p \to n_a$ and $q \to n_b$ such that $p, q \notin N'_R$.

Now, if $n_a = h_a$, then there cannot exist the edge $q \to n_b$ in G' as $R' = (N'_R, E'_R, h_R)$ is a region. In a similar fashion, when $n_b = h_a$ and when $n_b \neq h_a$ and $n_b \neq h_a$, we contradict the fact that R' is a region with header h_a. Thus, we can say that there exists one and only one node in R for which there

can exist an edge from a node outside R. Furthermore, thus node is exactly the header h_a of region R'. □

Thus, we have proved that if G is a subgraph of G' and R' is a region in G', then the subgraph of G containing the nodes in R' and all the edges between them is also a region with the same header as R'.

6.1.2 Reducible Flow Graphs and Spiral Graphs

Theorem 6.1 *Let $G = (N, E, n_0)$ be a reducible flow graph with $k > 1$ nodes. Then the necessary and sufficient condition for G to be a subgraph of some spiral graph on k nodes is that there exists a set of disjoint regions $R_1, R_2, ..., R_k$, whose union includes all the nodes in G, having the following properties:*

1. *$R_1, R_2, ..., R_k$ are all singleton.*
2. *There exists a sequence of regions $S_1, S_2, ..., S_k$ such that*
   (A) *$S_1 = R_1$,*
   (B) *for $i > 1$, S_i consists of S_{i-1} and R_i, with one the predecessor of the other,*
   (C) *S_k is G.*

Proof:
Let us first prove the sufficiency condition. Consider a reducible flow graph $G = (N, E, n_0)$ which satisfies the condition stated above. As each of the regions $R_1, R_2, ..., R_k$ is singleton, it consists of a single node. By a construction similar to Aho and Ullman, a spiral graph of which G is a subgraph can be obtained by adding the nodes $R_1, R_2, ..., R_k$ in the following sequence:

- Node R_1 is added using rule (1).
- If S_{i-1} is the predecessor of R_i, R_i is added using rule (2a).
- if R_i is the predecessor of S_{i-1}, R_i is added using rule (2b).

Let us now prove the necessity condition. Let us assume a reducible flow graph $G = (N, E, n_0)$ of $k > 1$ nodes which is a subgraph of some spiral graph formed by adding nodes of G in the order $R_1, R_2, ..., R_k$.

Let $S'_1 = R_1$. Now consider the spiral graph formed by adding the nodes $R_1, R_2, \ldots, R_{i-1}$. Let the header of this spiral graph be $R_j, 1 \leq j \leq i-1$. Therefore $S_{i-1} = (R_1, R_2, \ldots, R_{i-1})$, header = $R_j, 1 \leq j \leq i-1$. From Result 6.1, the subgraph of G having nodes $(R_1, R_2, \ldots, R_{i-1})$ is also a region with header R_j. Let us call this region S'_{i-1}.
$\therefore S'_{i-1} =$ Region with header R_j.

Now, if the node R_i is added to the spiral graph using rule (2b), we state that R_i is the predecessor of S'_{i-1} is G. For this, consider the spiral graph $S_{i-1} = (R_1, R_2, \ldots, R_{i-1}, R_i)$, header = R_i formed by adding to S_{i-1} the node R_i using rule (2b). Clearly the subgraph of g containing all nodes $(R_1, R_2, \ldots, R_{i-1}, R_i)$ and the edges between them is a region. Now, let this region be split into two regions (S'_{i-1}, R_i). Clearly, either S'_{i-1} is the predecessor of R_i or R_i is the predecessor of S'_{i-1}. If

S'_{i-1} is the predecessor of R_i then R_j is the header of S_i which contradicts the fact that R_i is the header of S_i. Thus R_i must be the predecessor of S'_{i-1}.

Similarly, if node R_i is added to S_{i-1} using rule (2a), we can prove that S_{i-1} is the predecessor of R_i. Now applying this argument inductively to S_i, S_{i-1} etc. we can see that G can indeed be partitioned into regions $S'_1, S'_2, \ldots, S'_k$ such that

- $S'_1 = R_1$,
- for $i > 1$, S'_i consists of S'_{i-1} and R'_i, with one the predecessor of the other
- S'_k is G.

Thus, we have proved both the necessity and sufficiency conditions for the theorem. □

Note Thus, we can see that a spiral graph is not a "maximal" reducible flow graph, in the sense that it does not contain all other reducible flow graph having the same number of nodes.

Example 6.1 Figure 6.1 shows a reducible flow graph of 4 nodes which is not a subgraph of any spiral graph of 4 nodes.

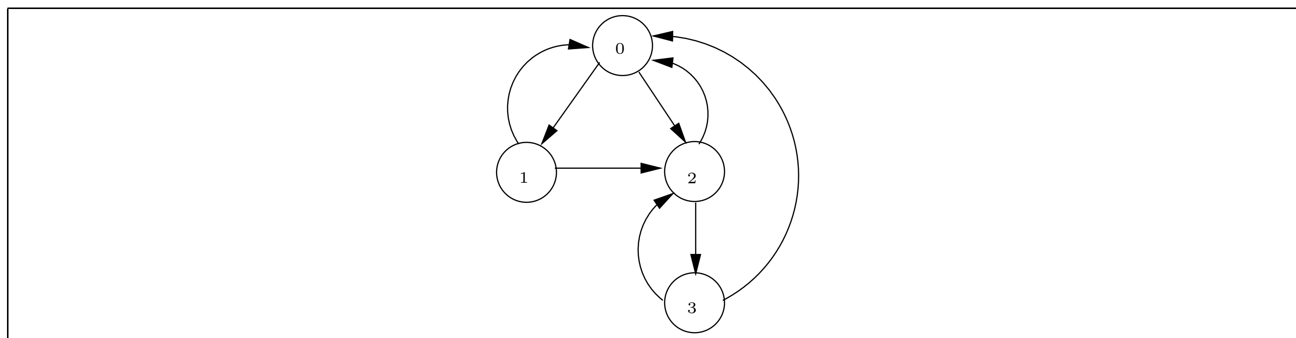


Figure 6.1: A reducible flow graph which is not a subgraph of any spiral graph.

6.1.3 More Motivations

In section 6.1.2 we have shown that spiral graphs are not the "largest" class of graphs. To gain further motivations for maximal reducible flow graphs, we now derive some more results.

Result 6.2 *In any reducible flow graph, if node m dominates node n, then addition of the edge $n \to m$ will keep the graph reducible.*

Proof:
Since node m dominates node n, the edge $n \to m$ added to the graph is a back edge. Now, by definition, a flow graph is reducible if and only if removal of all the back edges in the graph gives an acyclic graph. Since the original flow graph was reducible and we are adding a back edge, in the newly formed graph too, removal of all back edges will give an acyclic graph. Hence, the addition of the back edge $n \to m$ keeps the graph reducible. □

Result 6.3 *In any reducible flow graph, if two nodes m and n have the same immediate dominator (i.e. they are the children of the same node in the dominator tree) with m as the left sibling of n, then addition of the edge $m \to n$ keeps the graph reducible.*

Proof:
Let the common immediate dominator of nodes m and n be k. Let us consider that we add the edge $m \rightarrow n$ to the graph. Clearly, this edge is not a back edge as n does not dominate m. So, the edge is a forward edge. We will first show that the addition of this edge does not change the dominator tree of the flow graph. Addition of the edge $m \rightarrow n$ creates new path(s) from the initial node n_0 of the flow graph to n. These paths are all the paths from n_0 to m followed by the edge $m \rightarrow n$. Since k dominates m, each of these newly formed paths from n_0 to n contain k. Hence, addition of the edge $m \rightarrow n$ keeps the dominance relations same as before.

We will now show that the addition of the edge $m \rightarrow n$ will keep the graph reducible. We know that in the newly formed graph, the edge $m \rightarrow n$ is a forward edge. Let us assume that the newly formed graph is not reducible. Thus, there is a cycle consisting of only forward edges in the graph. Clearly, this cycle must contain the edge $m \rightarrow n$. Thus, there is a path in the original flow graph from n to m containing only forward edges. This means that n is before m in the acyclic ordering of the original flow graph. This contradicts the fact that m is a left sibling of n. Thus our original assumption is wrong. Thus, the flow graph formed by the addition of $m \rightarrow n$ is reducible. □

Result 6.4 *In any reducible flow graph, if there is an edge from a node m or any of its descendents to a node n where m and n have the same father in the dominator tree, then addition of the edge $n \rightarrow m$ will make the graph irreducible.*

Proof:
Let us assume that there is an edge from a node k to node n, where k can be m or any of its descendents and m and n are the children of the same node, say d in the dominator tree. Clearly, there is a

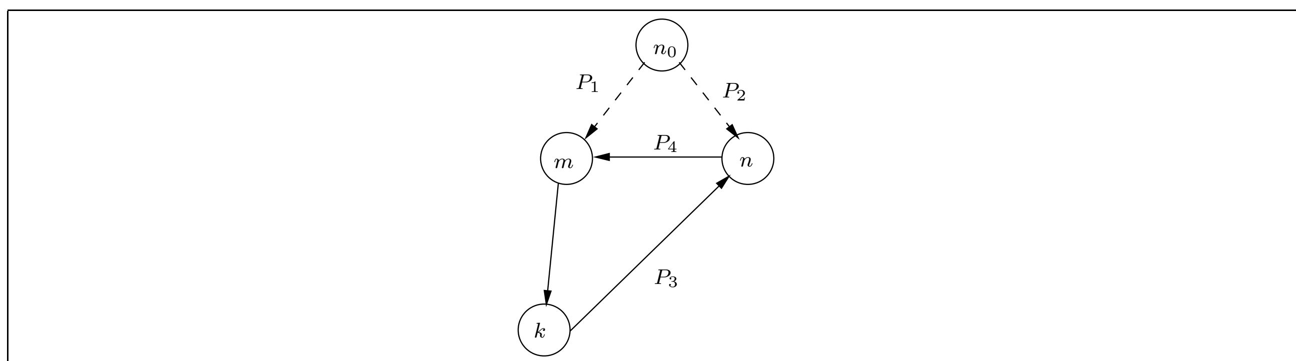


Figure 6.2: The (✶) Subgraph

path from d to m,since m is a child of d. Let this path be denoted as P_1. See Figure 6.2. Similarly, there is a path from d to n denoted as P_2. Now, path from m to k and the edge $k \rightarrow n$ constitute a path from m to n, say P_3. Also, the edge $n \rightarrow m$ constitute a path, say P_4 from n to m. It can now be easily seen that the subgraph of the original graph containing the node d, m and n form a (✶) subgraph, as defined in [32]. Now, according to the (✶) Characterization Theorem in [32], a flow graph is non reducible if and only if it contains a (✶) subgraph. Thus, the newly formed flow graph is irreducible. □

Result 6.5 *In any reducible flow graph, if m is any ancestor of n other that its father , then the addition of the edge $m \rightarrow n$ will either make the graph irreducible or change the dominator tree of the graph.*

Proof:
Let d be the father of m in the dominator tree. Thus, every path from the initial node n_0 to n contains d (and also m as m dominates d). Now if we add the edge $m \to n$, then we get additional path(s) in the flow graph from n_0 to n containing paths from n_0 to m followed by the edge $m \to n$. Since d does not dominate m, there exists at least one path from n_0 to m not containing d. This path along with the edge $m \to n$ forms a path from n_0 to n not containing d. Thus, in the newly formed graph d is no longer the dominator of n. Thus addition of the edge $m \to n$ changes the dominance relationship. See Figure 6.3.

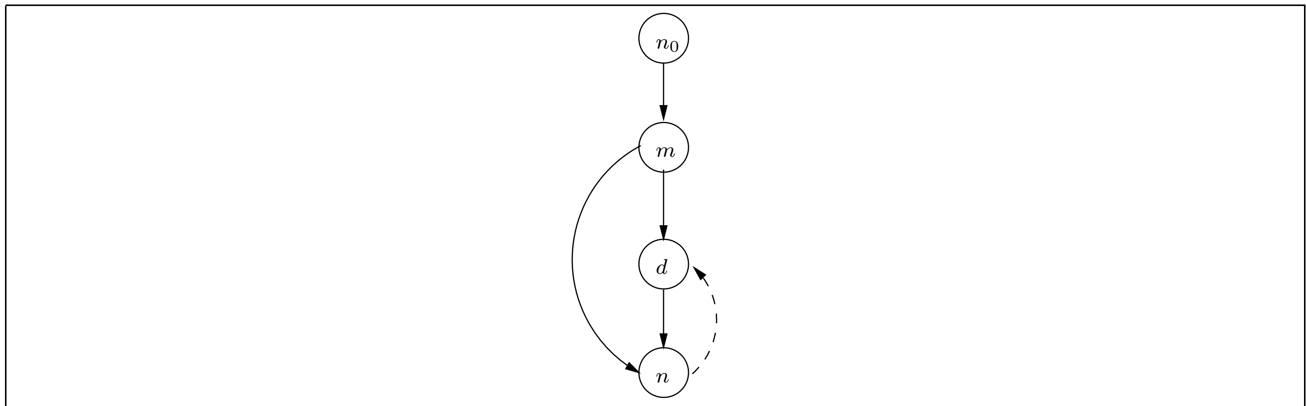


Figure 6.3: Addition of an edge $m \to n$

Furthermore, if there was a back edge $n \to d$ in the original flow graph, that back edge no longer remains a back edge in the new flow graph. This back edge along with the edge $d \to n$ forms a cycle in the flow graph containing only forward edge, making the flow graph irreducible. □

6.2 Maximal Reducible Flow Graphs

Definition 6.1 (Maximal Reducible Flow Graph) A *maximal rfg* of n nodes is an rfg of n nodes such that addition of any additional edge in that rfg renders it irreducible.

It can be seen that a spiral graph is a maximal rfg (Result 7.1). However, spiral graphs are not the only types of maximal rfg's.

Let us first consider the method of finding a maximal rfg having a given dominator tree. We find the maximal rfg having the given dominator tree by constructing the maximal rfg edge by edge adding edges using the following rules:

1. From a node say m, add an edge to all the ancestors of m in the dominator tree.
2. From a node say m, add an edge to all the children of m.
3. For all the node pairs (m, n) where both m and n have the same father, either add the edge $m \to n$ or add the edge $n \to m$ but not both. In any case, if the edge $m \to n$ is added, also add an edge from all the descendents of m to n i.e. add edge from all the nodes in the subtree rooted at m to n.

4. In other words if $n_1, n_2, \ldots, n_k$ all have the same father, then there exists an ordering among these nodes say $(n_1, n_2, \ldots, n_k)$ such that there is an edge $n_i \rightarrow n_j, \forall j > i$. Further, for each edge $n_i \rightarrow n_j$ there is an edge from all the descendents of n_i to n_j. Thus, we can say that there is no edge of the form $n_i \rightarrow n_j, j < i$. Thus, it can be seen that among the immediate children of a node there exists a "natural" ordering. We will define the "natural" order formally a little later.

5. Since self loops of the form $m \rightarrow m$ do not contribute to the reducibility or non reducibility, for all nodes, say m, add the self loop $m \rightarrow m$.

In the above rules, the edges added using rule (1) and (5) are *back edges*, the edges added using rule (2) are the *forward edges* and all the other edges are *cross edges*. Addition of any more edge to the graph will render it irreducible. So our construction indeed constructs a maximal rfg. The definition of a maximal reducible flow graph suggests that the dominator tree of the maximal reducible flow graph and hence of any reducible flow graph is an ordered tree. If $n_1, n_2, \ldots, n_k$ are the sons of any node, then the ordering of the nodes is given by $(n_1, n_2, \ldots, n_k)$ such that condition (4) is satisfied. We call n_1 the leftmost child and n_k as the rightmost child.

To find the maximal rfg of which the given rfg is a subgraph, we first find its dominator tree and go on adding edges using the above rules. The only ambiguity about the addition of an edge is rule (3), where we may have to decide which of the edges $m \rightarrow n$ or $n \rightarrow m$ is to be added. However, this ambiguity can also be resolved easily by applying rule (4). We find the acyclic ordering of the original graph and for each node, find the ordering of its immediate children such that the children are ordered in the same order as they appear in the acyclic ordering of the original graph. Once this ordering is formed, we add edges according to rule (4).

Example 6.2 As an example, Figure 6.4 shows a maximal reducible flow graph. The dark edges corresponding to the dominator tree from which the maximal reducible flow graph was constructed. The dotted edges are the edges added using rule (1) and the dash–dot edges are the edges added using rule (3). For simplicity, we have not shown the self loops.

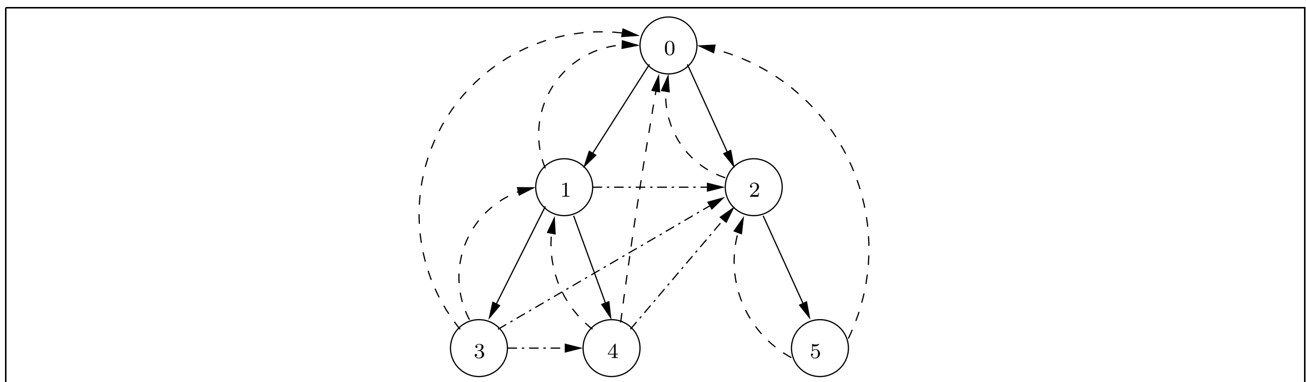


Figure 6.4: A maximal reducible flow graph

Definition 6.2 A "natural" ordering among the immediate children of a node in the dominator tree of any reducible flow graph is defined as an ordering among the nodes such that rule (4) in the definition of a maximal reducible flow graph is satisfied.

All the edges that are considered in rule (4) are forward edges. So to satisfy the rule (4), the children of any node must be ordered in the same order in which they appear in the acyclic ordering of the original graph. This "natural"ordering turns out to be simply the acyclic ordering of the children of the node.

Result 6.6 *The first node visited during the postorder traversal of the dominator tree has a single predecessor in the flow graph obtained from the maximal reducible flow graph for that dominator tree by the application of a T_1 transformation.*

Proof:
We first define the *postorder* traversal of the dominator tree, as follows,

1. Visit the children of the node in their "natural" order, and

2. Visit the node

Now consider the first node visited during the postorder traversal of the dominator tree. From the definition of postorder traversal, it is clear that this node is a leaf node of the dominator tree. Also, we can say that it is the leftmost child of some node, or else, its left-sibling will be visited first and will appear first in the postorder traversal. Since each time, the siblings of a node are visited in left to right fashion, we can then say that the first node visited during the postorder traversal is that descendent of the root of the dominator tree that is reached after traversing the leftmost edges until we can go no further. Figure 6.5.

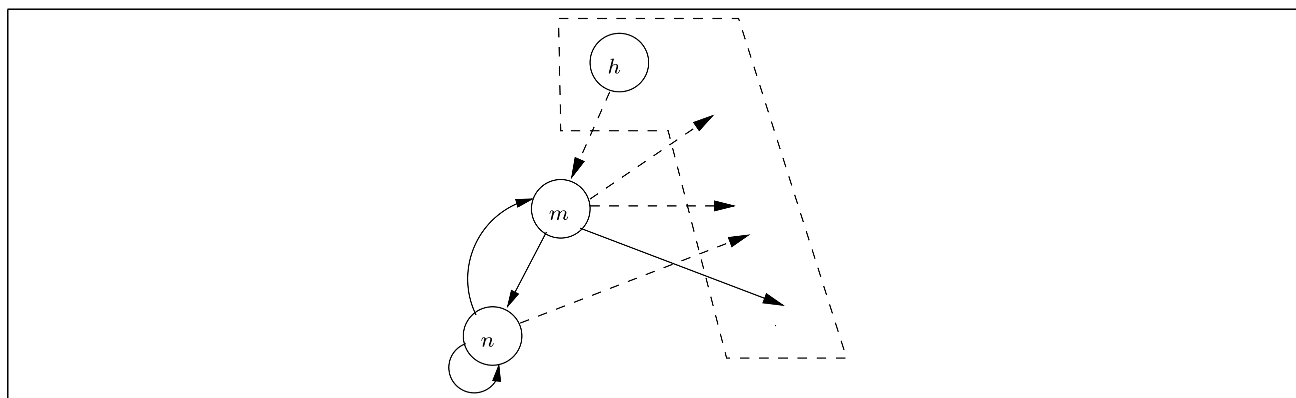


Figure 6.5: First node visited during postorder traversal

Now, if n is this node, then the only edges to node n added during the construction of a maximal rfg is $m \rightarrow n$, where m is the father of n, added by rule (2) and the self loop $n \rightarrow n$ added by rule (5). For a leaf node, we cannot add any edge to it by rule (1) and also for n we cannot add any edge to it by rules (3) and (4) since n or any of its ancestors do not have left-siblings.

Now, when we apply a T_1 transformation to the maximal rfg for the dominator tree the edge $n \rightarrow n$ will be deleted. Thus, the only edge in the resulting flow graph to the node n will be the edge $m \rightarrow n$. Thus, in the resulting flow graph, n has a single predecessor m. □

Result 6.7 *Given a dominator tree D, the flow graph obtained by the application of the above rules is indeed reducible.*

Proof:
We show that the flow graph obtained from above construction is reducible by showing that it can be reduced into a single node by repeated application of T_1 and T_2 transformations.

Consider that we construct the maximal rfg for the given dominator tree D by the application of the above rules. Now let n be the first node in the postorder traversal of D, as defined in Result 6.6. Now from result 6.6, we know that after the application of a T_1 transformation, n has a single predecessor. So, we can now apply a T_2 transformation to the flow graph in which m consumes n. After the consumption of n by m, in the resulting flow graph (which will also be a maximal rfg), if m has any more children, the next left sibling of n in the old dominator tree will now be the first node to be visited during the postorder traversal. If m does not have any more children, then m will now be the first node visited during the postorder traversal. In any case, in the resulting maximal rfg, we can again apply the above argument and eliminate the first node. In this way, we go on applying alternate T_1 and T_2 transformations, each time eliminating the first node visited during the postorder traversal. Thus, we can always reduce any maximal rfg into a single node by repeated alternate applications of T_1 and T_2 transformations. Hence, maximal rfg's are indeed reducible. □

Result 6.8 *Given a dominator tree D, the flow graph obtained from the above rules is indeed maximal.*

Proof:
We know that the addition of any additional edge in a maximal rfg renders it irreducible. Let us suppose that G is the flow graph obtained from D by the above rules. The only edges that were not added during the construction of G from D are as follows:

1. An edge from a node say m to its descendents other than its children.
2. An edge from a node say n or its descendents to its left-sibling or any descendent of its left sibling.
3. An edge from a node to any of the descendents of its right sibling.

We show that the addition of any of these edge makes the flow graph irreducible. Consider the first class of edges, i.e. from a node to its non-immediate descendent. We have shown in Result 6.5 that addition of such an edge makes the flow graph irreducible.

Now consider the second class of edge. Figure 6.6.

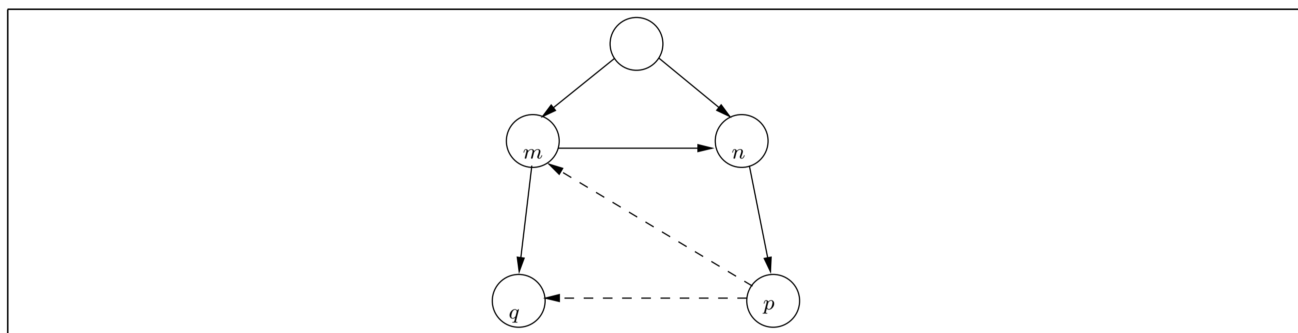


Figure 6.6: The edge $p \rightarrow q$

Let us add an edge from some descendent p of n or n to m, where m is a left sibling of n. Clearly, the edge $p \rightarrow m$ is not a back edge as m does not dominate p. So the edge $p \rightarrow q$ is a forward edge.

Now the edge $m \rightarrow n$ (added to the max. rfg using rule (4)), the path from n to p (added to the max. rfg using rule (2)), and the edge $p \rightarrow m$ form a cycle consisting of only forward edges. Thus the flow graph obtained by the addition of the edge $p \rightarrow m$ is irreducible. If we add the edge $p \rightarrow q$, then the edge $p \rightarrow q$ forms a path from a node outside the region consisting of m and its descendents to a node q inside that region that does not contain the header m. This violates the fact that m dominates q.

Now, in case of the third type of edges, an edge $m \rightarrow p$ will again contradict the fact that n dominates p.

Thus, we have shown that the addition of all possible additional edges to a maximal rfg renders it irreducible. Thus a maximal rfg is indeed a "maximal" rfg. □

Observation In a maximal reducible flow graph, if we traverse the edge $m \rightarrow n$ then

1. if $m \rightarrow n$ is a back edge, we reach a dominator of m.
2. if $m \rightarrow n$ is a forward edge, we reach an immediate ancestor of m.
3. if $m \rightarrow n$ is a cross edge, we reach a node which is neither the dominator nor the immediate child of m.

Now, given a rfg of n nodes, suppose we want to add one more node to it to give a rfg of $n+1$ nodes. Let us suppose that we want the dominator tree of the old graph to be the subtree of that of the new graph. Thus, we want to add a new node without disturbing the dominator tree. There and $n+1$ ways to do so:

1. Add it as the header to the old dominator tree
2. Add its as the son of one of the n nodes in the tree.

In any case, after the node is added, we can find the maximal rfg of the new graph using the above rules.

The significance of a maximal reducible flow graph is that every reducible flow graph of n nodes is a subgraph of some maximal reducible flow graph of n nodes. Thus, if we want to prove any property of reducible flow graph, then *proving that property for maximal reducible flow graphs is sufficient*. Of course, the property must be such that if a certain graph has that property then every subgraph of that graph has that property. Thus maximal rfg's can be used to prove the upper and lower bounds on the properties of reducible flow graphs. In our case, if the density of a graph is δ, then every subgraph of it also has a density δ. Thus, proving that the density δ of every maximal reducible flow graph is
$\delta \leq \lfloor \log n \rfloor$
will be sufficient to prove that the density of every reducible flow graph is $\leq \lfloor \log n \rfloor$.

Result 6.9 *Given a dominator tree D, the number of different maximal reducible flow graphs corresponding to that tree are,*

$\prod_{i=1}^{k}(S_{n_i}!)$

where $(n_1, n_2, \ldots, n_k)$ is the set of interior nodes in the dominator tree and S_{n_j} = number of sons of n_j in the dominator tree.

Proof:
Given an "unordered" dominator tree D, the number of "ordered" dominator trees corresponding to D is

$\prod_{i=1}^{k}(S_{n_i}!)$

which can be obtained by simple combinatorics. Since each ordered dominator tree corresponds to one maximal rfg, the above quantity is also the number of maximal rfg's corresponding to the dominator tree D. □

Thus, it can be seen that corresponding to a dominator tree, there exists a large number of maximal reducible flow graphs.

Definition 6.3 (Reachability) A node n is said to be *reachable* from a node m if $n = m$ or there is an edge $m \rightarrow n$ or there exists a node k in the graph such that n is reachable from k and k is reachable from m.

Result 6.10 *Each node in a maximal reducible flow graph is reachable from every other node.*

Proof:
By definition of a reducible flow graph, in any flow graph each node is reachable from the initial node n_0. Now, by construction of a maximal reducible flow graph, for each node n in the flow graph, there exists a back edge $n \rightarrow n_0$. Thus, the initial node n_0 is reachable from every other node. Therefore, by definition of reachability, every node in a maximal reducible flow graph is reachable from every other node. □

Result 6.11 *A depth first traversal of the dominator tree of a maximal reducible flow graph (and hence of any graph) gives its acyclic ordering.*

Proof:
Obvious □

Observation The *dag* of a maximal reducible flow graph corresponding to the dominator tree D is the flow graph obtained by adding edges using only the rules (2), (3) and (4). This is obvious because the dag of a flow graph is a subgraph of that flow graph containing all the nodes and all the edges except the back edges, as stated in Result 2.3.

Observation We know that in general, a flow graph can have more than one acyclic orderings. However, in case of a maximal rfg, we anticipate that each maximal rfg will have a single unique acyclic ordering. This can be easily seen by considering the following fact:

1. Find the *dag* of the flow graph. Its topological sort will give the acyclic ordering.

2. Initially, only the root node has no predecessor, so it is listed and then removed from the graph. After that the only node that has no predecessor is the leftmost child of the root.

3. Once this leftmost child is removed, the only node that now has no predecessor is the leftmost child of this node. If there is no leftmost child of that node, then the next right sibling is the only node that has a single predecessor and so on.

Thus, during the topological sort of the dag of a maximal rfg, at each stage there is just one node that can be listed next, giving a unique acyclic ordering. It is nothing but the depth first traversal of the dag.

6.3 Paths in Maximal Reducible Flow Graphs

Definition 6.4 (Redundant Paths) Given a list L of acyclic paths in a flow graph, the path n_1, $n_2, \ldots, n_k$ is said to be *redundant* if there exist some other path $m_1, m_2, \ldots, m_j$ such that $n_2, \ldots, n_k$ is a subsequence of $m_2, \ldots, m_j$.

Thus, it can be seen that when considering the redundant paths, the first node is ignored. This is because we will be considering all paths that begin with a back edge and also we assume an "initial" traversal of the graph.

Definition 6.5 (Non Redundant List) A list L of acyclic path in a flow graph is said to be *non-redundant* if no path in the list L is redundant.

Result 6.12 *All non-redundant path in a maximal reducible flow graph that begin with a back edge start at a leaf node of the dominator tree of the maximal reducible flow graph.*

Proof:
We first prove that an acyclic path in any reducible flow graph that begins with a back edge cannot contain any of the descendents of the first node of the path in the dominator tree. Let us assume that there exists an acyclic path in the reducible flow graph that begins with the back edge $m \rightarrow n$ and contains a descendant k of the first node m in the path. First of all, $n \neq m$ or else the path will not be acyclic. So clearly n is an ancestor of m in the dominator tree as $m \rightarrow n$ is a back edge. Let the portion of the acyclic path from n to k be A. So we now have an acyclic path $mnAk$. Now consider a path from the initial node n_0 of the graph (i.e. the root of the dominator tree) to n that does not contain m. Clearly, such a path does exist, or else m dominates n and since $m \neq n$, $m \rightarrow n$ will not be a back edge. Let the path from n_0 to n be X. So, we can say that $m \notin X$. Now consider the path n_0XnAk as shown in Figure 6.7.

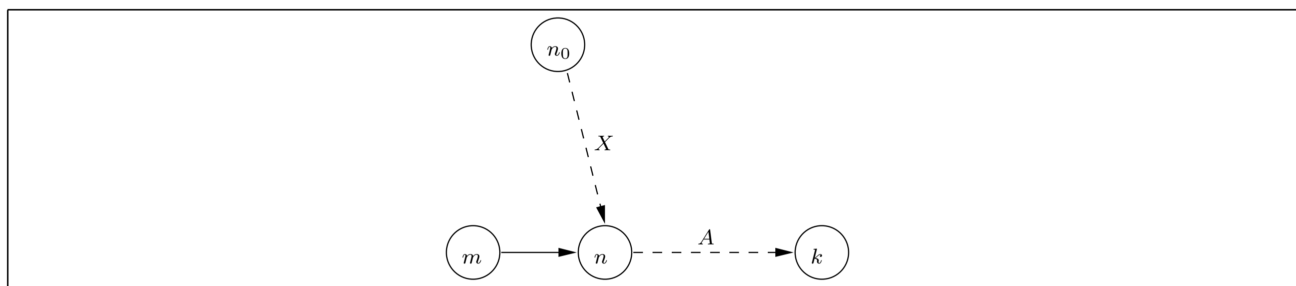


Figure 6.7: Acyclic Path in a Flow Graph

Clearly, $m \notin X$, $m \neq n$, $m \notin A$ and $m \neq k$. Also $m \neq n_0$ as there cannot be a back edge in a reducible flow graph that begins with the initial node n_0. Thus the path n_0XnAk is a path from the initial node n_0 to node k that does not contain the node m. This contradicts the fact that k is a descendant of m in the dominator tree i.e. the fact that m dominates k. Thus our original assumption about the existence of such a path is wrong. Therefore, we can say that any acyclic path in a reducible flow graph that begins with a back edge cannot contain any of the descendents of the first node in the path.

Now, consider an acyclic path L in a maximal reducible flow graph that begins with a back edge and whose first node, say m is not a leaf node. Since m is not a leaf node, it has at least one descendant. Let k be some descendant of m such that k is a leaf node. Now by the above result, we

can say that L does not contain k. Secondly, by definition of a maximal reducible flow graph there exist a back edge $k \to m$ in the maximal reducible flow graph. Now consider the path kL, as shown in Figure 6.8.

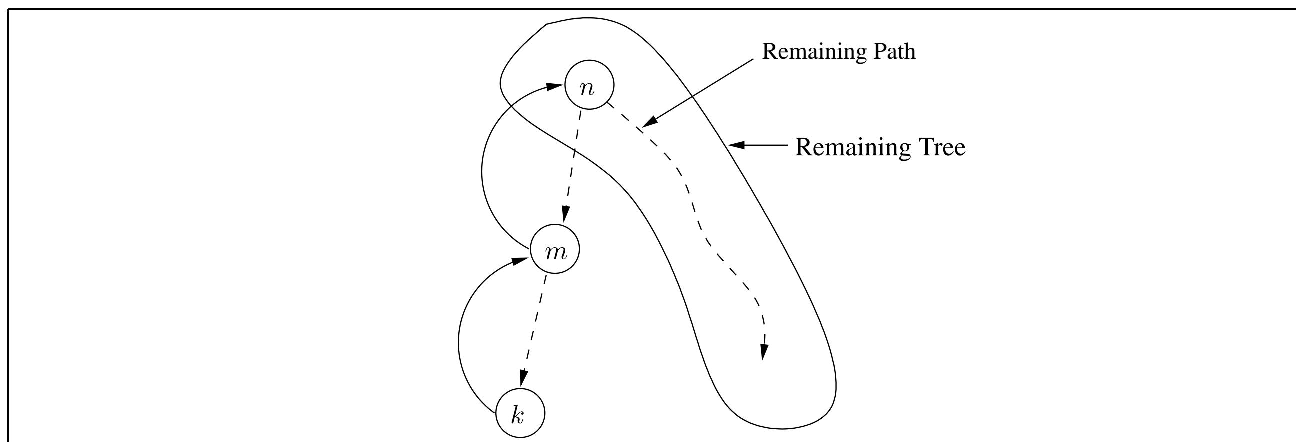


Figure 6.8: Acyclic Path Beginning With a Leaf Node

Clearly this is an acyclic path that begins with a leaf node of the dominator tree. Also, it is obvious that the path L becomes redundant in the presence of path kL. Since the node m and the path L was chosen arbitrarily, we can say that for every acyclic path in a maximal reducible flow graph that begins with a back edge and whose first node is not a leaf node, there exist some acyclic path that begins the with a leaf node of the dominator tree in whose presence, the original path becomes redundant. Therefore, we can conclude that an acyclic path that begins with a back edge and whose first node is not a leaf node of the dominator tree cannot be present in the list of no-redundant paths. As a result, all the non-redundant paths in a maximal reducible flow graph that begin with a back edge start at a leaf node of the dominator tree. □

Result 6.13 *A subgraph of a maximal reducible flow graph consisting of a node all its descendents and the edges between them is a region.*

Proof:

We know that a region is a subgraph of a flow graph such that every path from the initial node to a node in the subgraph contains the header of the region. Thus, in case the the subgraph is the subgraph consisting of a node h and all its descendents, then it follows by definition of dominance that every path from the initial node to a node in the subgraph will be through h. As a result, we can say that a subgraph of a maximal reducible flow graph consisting of a node and all its descendents in the dominator tree form a region. □

Result 6.14 *In a maximal reducible flow graph, if we traverse the cross edge $m \to n$, then the path from that point onwards, the path will not contain any of the descendents of m.*

Proof:

Consider an acyclic path containing the cross edge $m \to n$. Clearly, n is not a child of m. Also, n is not any other descendant of m by definition of a cross edge. Thus n is a certain node outside the region formed by m and its descendants. So, we know that any path from outside a region to any node of a region must contain the header of the region. Thus if the path contains some descendant of m, it must also contain m somewhere after n. This contradicts the fact that the path is acyclic. Thus our assumption is wrong. □

Now, we have already proved that an acyclic path beginning with a back edge cannot contain any descendents of the first node of the path. We can unify these results as follows.

Result 6.15 *An acyclic path in a maximal reducible flow graph that begins with either a back edge $m \rightarrow n$ or a cross edge $m \rightarrow n$ cannot contain any descendents of m.*

6.4 Regions in Maximal Reducible Flow Graphs

In this section we examine some properties of regions with respect to the maximal reducible flow graphs.

Result 6.16 *In any reducible flow graph, if we replace a node k by a region R, then the dominator tree of the new flow graph can be obtained by replacing the node k in the dominator tree of the old graph by the dominator tree of the region R.*

Proof:
Obvious □

Result 6.17 *Let h be some node in the dominator tree of a maximal reducible flow graph and suppose we are interested in finding a region R with header h. Let k be some child of h in the dominator tree. Then, if we include node k in R, the entire subtree of the dominator tree rooted at k must be included in R.*

Proof:
Let us suppose that we add a node k to the region R with header h. k is the child of h in the dominator tree. Let T be the subtree of the dominator tree of R rooted at k. See Figure 6.9.

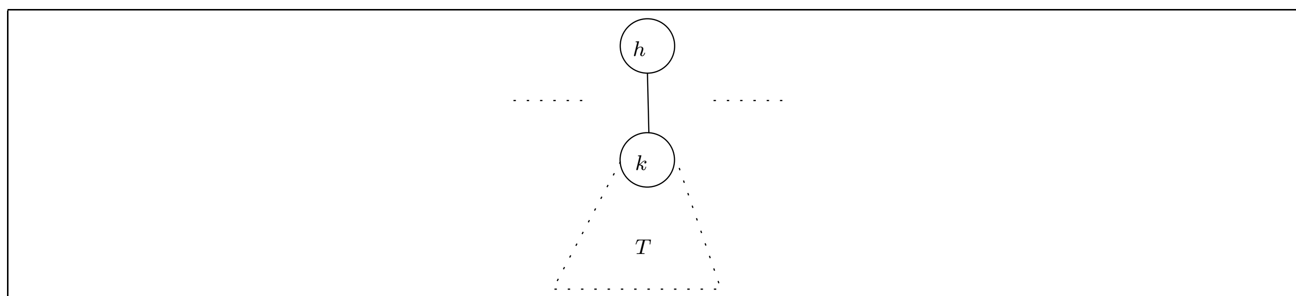


Figure 6.9: Region consisting the child of the header

Now, from the paper of Aho and Ullman [4], if there is an edge $m \rightarrow n$ in the flow graph and n is in a region R, $n \neq h$, h being the header of the region R, them m is in R.

Now, in a maximal reducible flow graph, there is a back edge from all the descendents of a node to that node by construction. Thus, in this case, there exists edges $m \rightarrow k, \forall m \in T$. Also, $k \in R, k \neq h$. So applying the above result, $m \in R \forall m \in T$. Thus the subtree of the dominator tree rooted at k has to be included in R. □

Result 6.18 *Let h be some node in the dominator tree of a maximal reducible flow graph and suppose we are interested in finding a region R with header h. Let k be some child of h other than the leftmost child. Then, if we include k in R, we have to include all the left-siblings of k in R.*

Proof:
Let us suppose that we include k in region R. Now let $(k_0, k_1, \ldots, k_{n-1})$ be the left siblings of k in the dominator tree. Now by construction in a maximal reducible flow graph, there exists edges $k_i \to k, 0 \le i \le n-1$. So, by the argument in [4] all the left siblings of k have to be included in R.

Now, by virtue of Result 6.17, when we include a node k in the region, $k \neq h$, then all the descendents of k also have to be included in the region. Thus, when we include a node k in the region, we have to include all the left siblings as well as the subtrees rooted at them in the region. □

Definition 6.6 (Feasible Region) Let $R = (N, E, h)$ be a region in a reducible flow graph and let $R_1 = (N_1, N_1 \times N_1 \cap E, h_1)$ be a subregion of R. Let $R_2 = (N_2, N_2 \times N_2 \cap E)$ be the subgraph of R not containing the nodes in R_1 i.e. $N_2 = N - N_1$. Then the region R_1 is a *feasible* region if and only if R_2 is a region.

It can be easily seen that if R_1 is feasible, then R_2 is also feasible and that a pair of feasible subregions forms a parse of the given maximal region.

Result 6.19 *Let h be some node in the dominator tree of a maximal reducible flow graph. Let R be the region consisting h and all its descendents in the dominator tree. Let $(k_1, k_2, \ldots, k_n)$ be all the children of h in their "natural" order. Let R_1 be the region consisting of k_n and all its descendents. Then R_1 is a feasible region.*

Proof:
See Figure 6.10. We know that k_n and all its descendents form a region. Thus R_1 is a region. Now

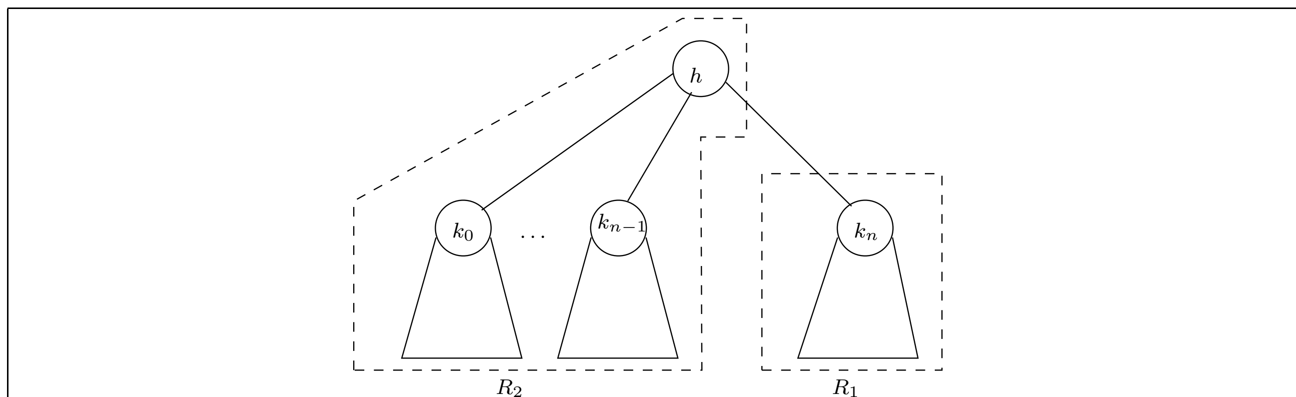


Figure 6.10: Region with rightmost child as header

to show that R_1 is feasible, it is sufficient to show that R_2 consisting of all the remaining nodes is a region. To show that, we first note that in a maximal rfg there exists an edge $m \to h$ in $R \forall m \in R_1$. Apart from that, there does not exist any other edge of the form $m \to n, m \in R_1, n \in R_2$. Thus all the edge from R_1 to R_2 are to its header h. Thus R_2 in indeed a region. Hence R_1 and therefore R_2 is a feasible region. □

Result 6.20 *Let h be some node in the dominator tree of a maximal reducible flow graph. Let R be the region consisting of h and all its descendents in the dominator tree. Let $(k_1, k_2, \ldots, k_n)$ be all the children of h in their "natural" order. Let R_1 be the subgraph consisting of any two children k_i and k_j and all its descendents. Then R_1 is not region.*

Proof:

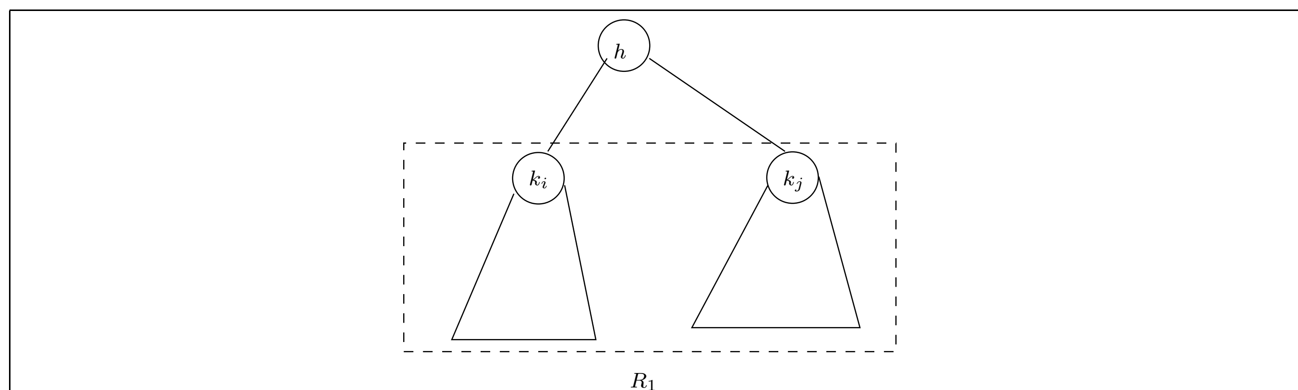


Figure 6.11: Two children forming a region

See Figure 6.11. Let R_1 be the subgraph of R containing nodes k_i, k_j and their descendents.Since k_i and k_j are the children of h, by definition, there exists the edges $h \rightarrow k_i$ and $h \rightarrow k_j$. Thus the subgraph R_1 has two entry points from outside it. So, R_1 cannot be a region and hence it cannot also be a feasible region. □

Result 6.21 *Let a maximal region R be divided into two regions R_1 and R_2 such that R_2 dominates R_1. Then the header of R_1 is the rightmost child of the header of R_2 in the dominator tree of R.*

Proof:

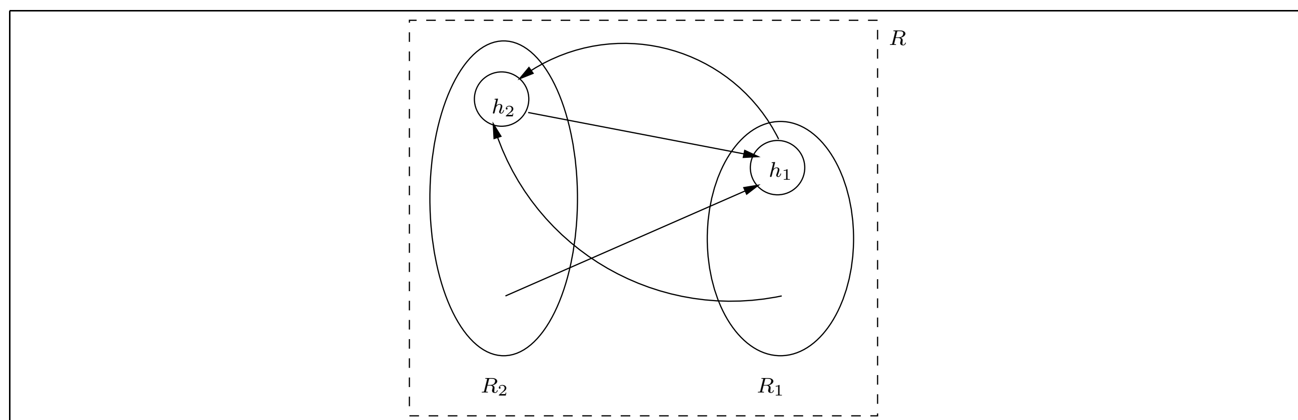


Figure 6.12: Division of a region into two regions

See Figure 6.12. R is a region divided into two subregions R_1 and R_2 with headers h_1 and h_2 respectively. Also, R_2 dominates R_1. Because of this, the header h_2 of R_2 is nothing but the header h of the original region R. Since h dominates all the nodes in R, h_2 dominates all the nodes in R_2 as well as all the nodes in R_1. So, clearly h_2 dominates h_1. Also, h_2 is the immediate dominator of h_1. To see this, let us suppose that there is an immediate dominator, say $d \in R$, of h_1 and $d \neq h_2$. Then every path from h_2 to h_1 must pass through d. But there exists an edge $h_2 \rightarrow h_1$, giving a path from h_2 to h_1 that does not contain d. Thus our assumption about the existence of node d is wrong. Hence h_2 is the immediate dominator of h_1 in the dominator tree of R.

Now in a maximal region, there exist edges from all the nodes in R_1 to h_2. There edges are the back edges and can be accounted for as the edge from a node to its dominator in a maximal rfg. Next, there is an edge $h_2 \rightarrow h_1$ which can be accounted for as an edge from a node to its child in a maximal

rfg. Also, there are edges from nodes in R_2 to the header h_1 of the region R_1. Clearly, these edges are not back edges as h_1 does not dominate any node in R_2. So these edges are forward edges. These edges can be accounted for as edges from the left siblings of a node and their descendents to that node in a maximal rfg. Thus, all the nodes in R_2 except h_2 are either the left siblings of h_1 or the descendents of some left sibling of h_1. So these nodes must be on the left of h_1 in the dominator tree. So, h_1 must be the rightmost child of h_2 in the dominator tree of R. □

These results will be used further to show that the two characterizations of spiral graphs that we have developed viz. Result 6.1.2 and section 7.2 are identical.

Result 6.22 *Every maximal reducible flow graph has a unique parse.*

Proof:
It can be observed that a given maximal reducible flow graph can be parsed in a single way as given in Result 6.21. The parses themselves are maximal regions. Applying result 6.21 inductively to them, the result immediately follows. □

6.4.1 Synthesis of Reducible Flow Graphs

In Chapter 2 we have defined reducible flow graphs and given a flow graph, we can find whether or not it is reducible. This is the analysis aspect of the problem. But what about the synthesis or construction of reducible flow graphs? Researchers have proposed methods of constructing reducible flow graphs, but these methods are either not general (they do not generate all possible reducible flow graphs, e.g. spiral graphs) or are based on the inverse of T_1 and T_2 transformations as we will be defining in Chapter 9. The concept of *maximal reducible flow graphs can be thought as a new method of synthesizing reducible flow graphs that is general and is based on the dominator tree of the flow graphs*.

Any reducible flow graph can be generate by first constructing the maximal reducible flow graphs and then removing selected edges from the graph, taking care that in the resulting flow graphs, removing all the back edges, the flow graph is such that all nodes are reachable from the initial node. This method allows us to construct arbitrary reducible flow graphs that may seldom occur practically.

6.5 Partial Ordering in Reducible Flow Graphs

6.5.1 The Dominance Relation

We know that a node d of a flow graph *dominates* node n, written as *d dom m*, if every path from the initial node of the flow graph to n goes through d.

Now it can be easily seen that this definition implies a dominance relationship among the nodes of the flow graph. We represent this relationship symbolically by the $\leq$ sign. Thus, in a flow graph $d \leq m$ if d *dom* m. This dominance relation is

1. *reflexive*, since all nodes n, $n \leq n$ i.e. every node dominates itself.

2. *transitive*, since if $a \leq b$ and $b \leq c \Rightarrow a \leq c$ i.e. if a dominates b and b dominates c, then a dominates c.

3. *antisymmetric*, since $a \leq b \Rightarrow b \not\leq a$, unless $a = b$ i.e. if a and b are distinct, then either a dominates b or b dominates a, but not both.

Thus, the dominance relation on the nodes of a flow graph is reflexive, transitive and antisymmetric i.e. it is a *partial ordering relation*. Now each partial ordering relation can be represented by a *Hasse diagram* in which there is a sequence of arrows from a to b if and only if $a \leq b$ and the diagram is drawn is such a way that all the arrow heads point upwards. In case of flow graph, the *Hasse diagram is nothing but the dominator tree of the flow graph drawn inverted with the root at the bottom.*

6.5.2 Some Results Based On Partial Ordering

We can now immediately make some observations about the reducible flow graphs by considering the dominance relationship $\leq$

1. The set of nodes N in the reducible flow graph and the dominance relation ($\leq$) from a partially ordered set (*poset*), represented as $(N, \leq)$.

2. If H is the height of the dominator tree, then the length of the longest *chain* in the poset $(N, \leq)$ is nothing but H.

3. The *minimal element* of N is the root (n_0) of the dominator tree, since for no $b \in N, b \neq n_0, b \leq n_0$ i.e. there is no node $b \neq n_0$ such that b dominates n_0.

4. The *maximal elements* of N are nothing but the leaf nodes of the dominator tree since if k is a leaf node of the dominator tree then for no $b \in N, b \neq k, k \leq b$ i.e. there is no node $b \neq k$ such that k dominates b.

5. The poset $(N, \leq)$ *cannot* possibly from a *lattice*. This is because given two elements $a, b \in N$, such that neither $a \leq b$ nor $b \leq a$ we can find their *greatest lower bound* but these two element *cannot* have any common upper bound. If the nodes a and b have any common upper bound i.e a node k such that $a \leq k$ and $b \leq k$ then a dominates k and b dominates k. Then either a dominates b (or else there is a path in the flow graph from n_0 to k that does not contain a, contradicting the fact that $a \leq k$) or b dominates a (or else there is a path in the flow graph from n_0 to k that does not contain b, contradicting the fact that $b \leq a$). This contradicts the fact that neither $a \leq b$ nor $b \leq a$. Thus we cannot find the least upper bound of two unrelated elements in $(N, \leq)$. Therefore, $(N, \leq)$ cannot form a poset.

6. We know that in a partially ordered set $(N, \leq)$, if the length of the longest chain in n, then the elements of N can be partitioned into n disjoint *antichains*. In case of reducible flow graphs the length of the longest chain is the height H of the dominator tree. Thus, the nodes in a reducible flow graph can be partitioned into H disjoint subsets such that for any two distinct elements a and b belonging to the same subset, neither $a \leq b$ nor $b \leq a$.

6.5.3 Antichain Method for Matrix of Levels

Let us first define the matrix of levels for the given flow graph.

Definition 6.7 (Level of a node) If $m \rightarrow n$ is a back edge in a non-redundant acyclic path in a flow graph, then the *level* of node n is the number of back edges covered in reaching n in the path beginning at the first node.

Definition 6.8 (Matrix of Levels) The *matrix of levels* for a flow graph is a matrix indicating the levels of the nodes in the flow graph. It consists of d rows, d being the depth of the flow graph. A node n is in row (i.e. level) i iff its level is i in some non-redundant acyclic path in the flow graph.

Observation It has been observed that there is a certain relationship between the matrix of levels of a maximal rfg and its dominator tree. We know that the dominator relation induces a partial ordering among the nodes of the flow graph. If the height of the dominator tree is h, then the length of the longest chain in the poset $(N, \leq)$ is h. Thus, there exists h antichains in $(N, \leq)$. We can form these antichains as shown in the Figure 6.13.

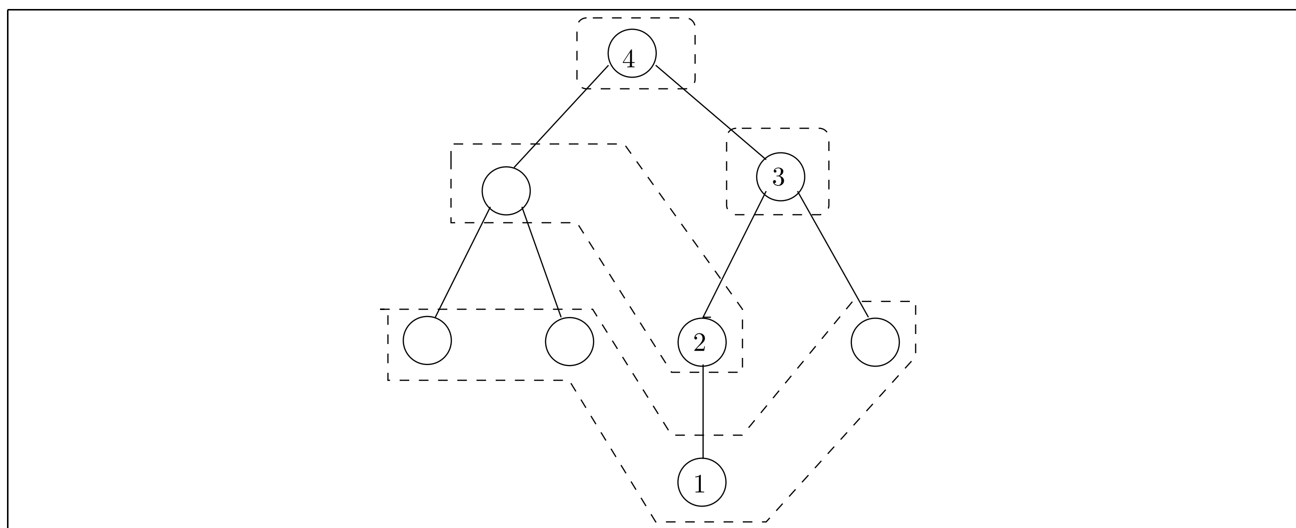


Figure 6.13: Antichains in the dominator tree

Now let us number the nodes in the longest chain starting with the leaf and working towards the root. The leaf is assigned the number 1 and continuing the root is assigned the number h. Now each antichain formed will contain exactly one element of the longest chain. Let the antichain A_i contain element i of the longest chain. Then it has been observed the the level i of the matrix contains all the nodes $N - A_i \cup A_{i-1} \cup \ldots \cup A_1$.

Also, we can observe that at each level, the antichain consists of all the leaf nodes of the dominator tree.

From the above observation, we can immediately give an algorithm to find the matrix of levels for a maximal rfg, given the dominator tree of the max rfg. This algorithm can be used to find the matrix of levels for a maximal rfg by directly looking at its dominator tree instead of first finding the list of non-redundant paths and then finding the matrix.

6.5.4 Antichain Method for Heuristic Node Listings of Maximal rfgs

In this section, we give the relationship between the antichains in the dominator tree and the node listing produced by the simplified heuristic node listing program (`sheuristic`).

```
1: /* D is the dominator tree of the maximal rfg */
   /* h is the root of D */
2: level := 1 ; /* initial level */
3: while D is not empty do
4:    print level;
5:    print all non leaf nodes in D ;
6:    remove all leaf nodes from D ;
7:    level := level + 1 ;
8: end while
```

Algorithm 6: Finding the matrix of levels for max rfg

Let the dominator tree be partitioned into h disjoint antichains as before. Then we give two equivalent methods of finding the heuristic node listing for the maximal rfg corresponding to that dominator tree. These methods always give $\delta \leq d$, where d is the depth of the maximal rfg (i.e. $h - 1$).

Method 1

1. Find the antichains in the dominator tree and label them as $A_1, A_2, \ldots, A_h$.
2. For all the antichains $A_1, A_2, \ldots A_{h-1}$ do step 3.
3. Given an antichain A_i, for all nodes n_i in A_i do the following: at level i, list all the nodes in D except n_i and its descendents.

Method 2

This method is a refinement of Method 1. If n_1 and n_2 are two nodes in a certain level, then by definition of an antichain, none of the one dominates the other. So, n_1 and n_2 do not have common descendents. So when applying the rules of Method 1, n_1 and its descendents will be listed when applying the rule to n_2 and n_2 and its descendents will be listed when applying the rule to n_1. In other words, *if antichain A_i has two or more elements, then that level has all the nodes in the graph.*

If the antichain A_i is singleton consisting of only n, then level i does not contain n and its descendents. This leads to the following method.

1. Find all the antichains in the dominator tree and label then as $A_1, A_2, \ldots, A_h$.
2. For antichains $A_1, A_2, \ldots, A_{h-1}$, do step 3.
3. If A_i is singleton consisting of only n_i, then list at level i all nodes except n_i and its descendents, else list all the nodes at level i.

This method of generating heuristic node listings is computationally much more efficient than the method of first finding all the acyclic paths in the flow graph that begin with a back edge, eliminating the redundant paths and then finsing the node listing.

As an example of this method, consider the maximal rfg corresponding to the dominator tree shown in Figure 6.14.

The node listing for this maximal rfg is as follows:

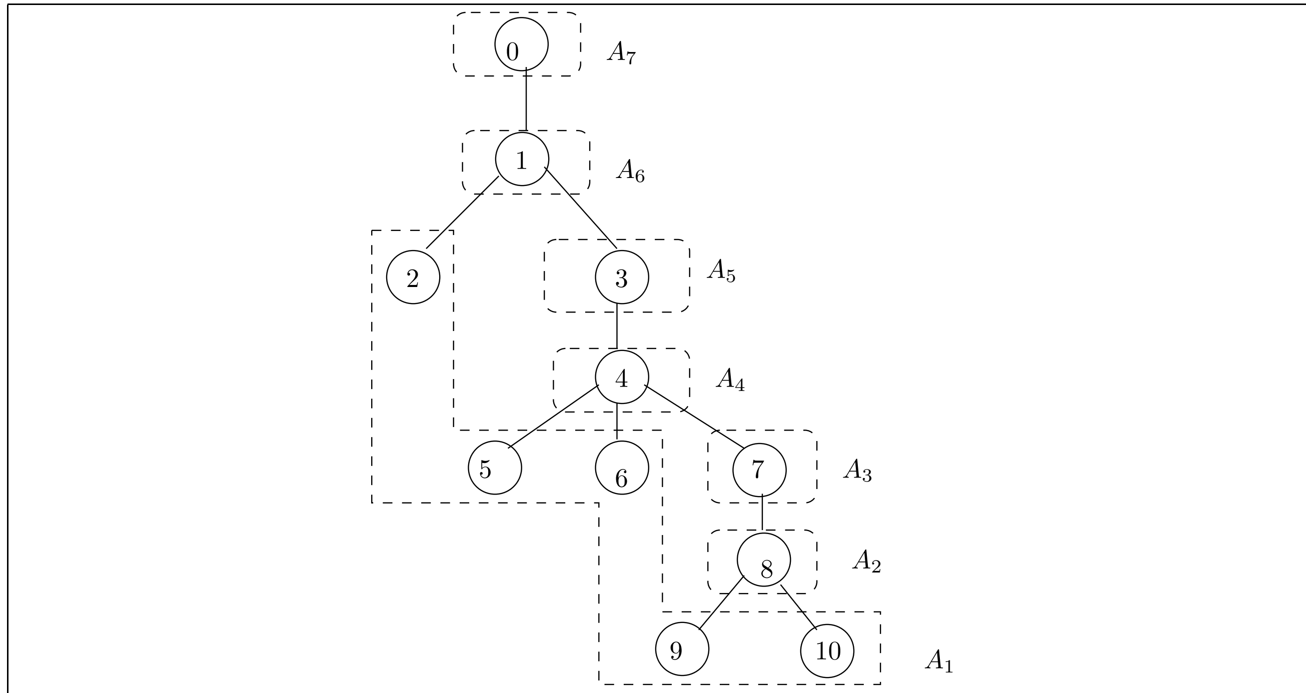


Figure 6.14: Heuristic node listing of maximal rfg

Level	Nodes
1	0 1 2 3 4 5 6 7 8 9 10
2	0 1 2 3 4 5 6 7
3	0 1 2 3 4 5 6
4	0 1 2 3
5	0 1 2
6	0

- A_1 is non-singleton, so level 1 has all the nodes.
- Level 2 does not contain 8 and its descendents $9, 10$.
- Level 3 does not contain 7 and its descendents $8, 9, 10$ and so on.

6.6 Binary Parsable Reducible Flow Graphs

As seen in the previous chapters, the only reducible flow graphs for which we can say that $\delta \leq \log n$ are the spiral graphs with all the nodes added by rule (2b). In this section, we introduce a more general class of reducible flow graphs for which the density is provably bounded by $\log n$.

Definition 6.9 (Binary Parsable rfgs) A *binary parsable reducible flow graph* of 2^n nodes is a reducible flow graph of 2^n nodes and is defined as follows.

1. A single node with no edge is a binary parsable rfg.
2. A spiral (2b) graph of 2^n nodes is a binary parsable rfg.
3. If a reducible flow graph G of 2^n nodes can be parsed into two regions R_1 and R_2 each with 2^{n-1} nodes and R_1 and R_2 themselves are binary parsable rfgs, then G is also a binary parsable rfg.

Thus, it can be seen that a binary parsable rfg is either a (2b) spiral graph or can be parsed into two regions each having half the number of nodes. It can be easily seen that there exist reducible flow graph that are not binary parsable viz. a spiral graph with the last node added using rule (2a). Nevertheless, the class of binary parsable flow graphs is more general than the class of spiral (2b) graphs.

Result 6.23 *For every binary parsable rfg of 2^n nodes, $\delta \leq n$.*

Proof:
We will prove this result by induction on n. As the basis of induction, a single node has density 0, which trivially satisfies the result. Now consider a binary parsable rfg with 2^n nodes, $n > 0$. If it is a (2b) spiral graph, we have already proved that for spiral (2b) graphs $\delta \leq \log n$. In case it is not a spiral (2b) graph, then it can be parsed into two regions R_1 and R_2 each with 2^{n-1} nodes. Let us assume without loss of generality that R_2 dominates R_1. Then this parse is as shown in figure 6.12.

Now any path that begins in R_1, enters R_2 and again enters R_1 is contained in $R_1\hat{R}_2\hat{R}_1$. Any path that begins in R_2, enters R_1 and again enters R_2 is contained in $R_2\hat{R}_1\hat{R}_2$. Thus, $R_2R_1\hat{R}_2\hat{R}_1$. By the inductive hypothesis, each of R_1 and R_2 has a density $\delta' \leq n-1$. Therefore, the density of the original graph is given by $\delta \leq n-1+1 \Rightarrow \delta \leq n$. □

6.6.1 Constructing Binary Parsable Flow Graphs

In this section, we describe one method of constructing binary parsable reducible flow graphs. This method is based on the fact that a maximal rfg can be partitioned into regions in a unique was, as stated before. The method of construction can be stated in a single line as,

To construct a binary parsable rfg of 2^n nodes, take two binary parsable rfgs of 2^{n-1} nodes, say R_1 and R_2. Attach the root of the dominator tree of R_1 as the rightmost child of the root of the dominator tree of R_2 and construct the maximal rfg for the dominator tree so formed. This maximal rfg is binary parsable.

As an example, the figure 6.15 shows the construction of binary parsable flow graph of 2, 4 and 8 nodes.

6.7 Weak Node Listings for Maximal Rfg's

In this section, we present some results regarding weak node listings for maximal reducible flow graphs.

Result 6.24 *In a maximal rfg, any basic path can have at most one back edge.*

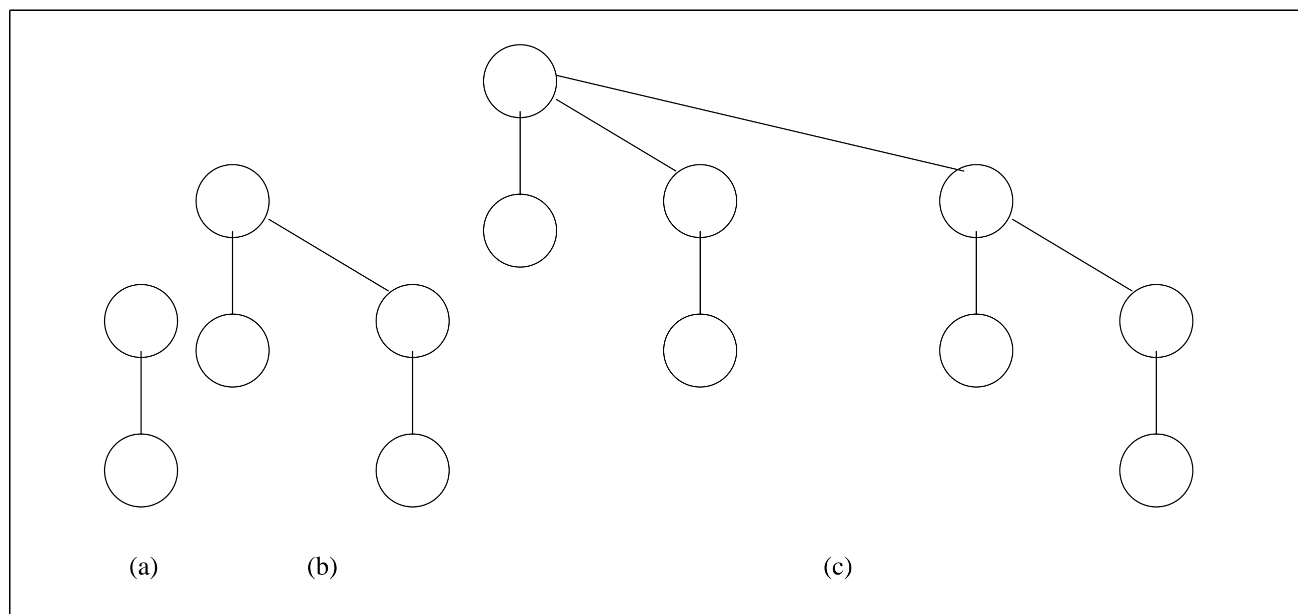


Figure 6.15: Construction of Binary Parsable Reducible Flow Graphs

Proof:
A basic path in a flow graph is an acyclic path $(x_1, x_2, \ldots, x_k), k \geq 1$ such that there is no shorter acyclic path from x_1 to x_k which is a subsequence of $(x_1, x_2, \ldots, x_k)$. Now consider that a basic path in a maximal reducible flow graph G contains 2 or more back edges. Let the first of these back edges be $a \rightarrow b$ and the second be $c \rightarrow d$. Thus, the path can be represented as $P_1abP_2cdP_3$. Therefore, by the generalization of Lemma 2 in [4] (Result 8.7), d dominates a. Now, by construction, in a maximal rfg, there exists the back edge $a \rightarrow d$. Now the path P_1adP_3 is a path contained in $P_1abP_2cdP_3$ and having the same source and destination as before. This means that $P_1abP_2cdP_3$ cannot be a basic path. □

Result 6.25 *For every maximal rfg of n nodes, there exists a weak node listing of length $2n$.*

Proof:
From Result 6.24, any basic path in a maximal rfg can have at most 1 back edge. If $\hat{A}$ is the acyclic ordering of the maximal rfg, then it can be easily seen that all the basic paths in the maximal rfg are a subsequence of $\hat{A}\hat{A}$. Thus, $\hat{A}\hat{A}$ is a *weak node listing* for the maximal rfg and its length is $2n$. □

The significance of the above result is limited since if G' is a subgraph of G, it does not necessarily imply that the weak node listing for G is also a weak node listing for G'.

6.8 Applications of Maximal Reducible Flow Graphs

In the previous sections, we have derived many interesting results and properties of maximal reducible flow graphs. In the next chapter, we are going to examine one particular type of maximal rfg viz. spiral graphs and their properties. It can be seen that a maximal rfg has a very well defined and fixed structure. This introduces regularity in the properties of maximal rfg. We can derive many properties of maximal rfg by exploiting this regularity in the structure of a maximal rfg. Some possible applications of maximal reducible flow graphs can be,

- **To derive or prove the upper or lower bounds on the properties of reducible flow graphs**
  As we have seen, every reducible flow graph is a subgraph of one or more maximal reducible flow graph. Thus, any arbitrary rfg is contained in some maximal rfg. Thus, if we want to derive or prove any property for rfg, *is is sufficient to prove that property for maximal rfg's*. Of course, the property must be such that if a flow graph G has that property, every reducible sub-flow graph G' of G also has that property.

- **To analyze various data flow analysis algorithms**
  Since maximal rfgs are the "largest" of all the reducible flow graphs, we conjecture that various data flow analysis algorithms may show worst/best case behavior when they perform analysis on maximal rfgs. A study of the behavior of these algorithms in relation to maximal rfgs may enable us to improve the worst case time complexity of these algorithms.

- **For comparison study of various data flow analysis algorithms**
  A large number of data flow analysis algorithms are available today. For worst case comparison study, we feel that a comparison of these algorithms on the basis of their behavior on maximal rfgs may be sufficient.

- **As a measure of program complexity**
  The number of nodes in the flow graph along with the degree of "closeness" of the given reducible flow graph to maximal rfgs can be used as a measure of the complexity of the program. The "closeness" may be measured in terms of the number of edges that are needed to be added to the given flow to convert into a maximal rfg.

In addition, during the course of the discussions, we have also found that the maximal rfgs are useful for verifying our conjectures or to provide contradictions to the things we propose.

As an example of the use of maximal rfgs in proving the upper and lower bounds, we prove that the number of acyclic orderings of a given reducible flow graph is a lower bound on the number of parses it can have. We know that the parse of a reducible flow graph is nothing but the its division into regions. This is done so as to impose a hierarchical structure on the flow graph. Different parses may lead to different interpretations of the same program, from the control flow analysis point of view. Intuitively, a simple program has many different interpretations and a complex on has fewer interpretations. Thus, more the number of parses, simpler the program and vice versa.

Now, we know that every reducible flow graph is a subgraph of some maximal reducible flow graph. Also, a reducible flow graph can be a subgraph of more than one maximal rfg. We have said that the number of maximal rfgs of which the given rfg is a subgraph is equal to the number of acyclic orderings of the given rfg. However, it has been practically observed that apart from these maximal rfgs, a given rfg can be a subgraph of some more rfg's as well. The dominator trees of these maximal rfg's is different from any "ordered" dominator tree of the given rfg. Thus, if N_A is the number of acyclic orderings of the given rfg and N_M is the number of maximal reducible flow graphs of which the given rfg is a subgraph, then $N_M \geq N_A$. Now it can be easily seen that each maximal rfg of which the given rfg is a subgraph corresponds to a different parse of the given rfg. Thus, the number of parses of the given rfg is given by $N_P = N_M$. Hence $N_P \geq N_A$. Thus, *the number of acyclic orderings of the given rfg is a lower bound on the number of parses it can have*.

One important point that come out is regarding the complexity. We can say that the *complexity of the program is inversely proportional to* N_M. Consider a maximal rfg itself. Since it has a unique parse, $N_M = 1$ and hence N_M has the smallest value for maximal rfg and hence the maximal rfg

corresponds to the most complex programs. As N_M for the given rfg increases, its complexity drops down.

6.9 Density, Node Listings and Maximal Rfg's

As we have seen previously, every reducible flow graph of n nodes is a subgraph of some maximal rfg of n nodes. As a result, every *strong* node listing for a maximal rfg is also a strong node listing for every subgraph of it. Thus, the *densities of maximal reducible flow graphs are an upper bound on the densities of arbitrary reducible flow graphs*. Thus, to prove that $\delta \leq \log n$ for any reducible flow graph, it is sufficient to show that the result is true for all maximal rfgs. We hope that dealing with maximal rfgs will make the proof easier to tackle. We first state the following simple result.

Result 6.26 *The δ is the density of a maximal rfg with a dominator tree of height h then $\delta \geq \lfloor \log h \rfloor$.*

Proof:
If h is the height of the dominator tree of a maximal rfg G, then the subgraph S of G formed from the h nodes in the longest chain is a spiral graph of h nodes, all added using rule (2b). Now, for (2b) spiral graphs, we have already proved that $\delta_S = \lfloor \log h \rfloor$. Since S is a subgraph of G, it follows that for G, $\delta \geq \delta_S \Rightarrow \delta \geq \lfloor \log h \rfloor$. □

Also, from the construction of a maximal rfg, it can be seen that the dominator tree of a flow graph is one of its important properties. Also, the depth of the flow graph is bounded from above by the height of the dominator tree, since $d \leq h - 1$, h being the height of the dominator tree. Some experimentation along with the observations regarding maximal rfgs suggest that the density of a reducible flow graph is bounded by $\delta \leq \log{(d+1)}$, which is a more stronger bound than $\log n$. Thus, the concept of maximal rfgs have not only motivated us to shift our concentration on the dominator tree and the dominator relationship in the flow graphs but also have enabled us to arrive at a stronger bound on the densities of flow graphs than the previous one.

❑❑❑

CHAPTER 7

Properties of Spiral Graphs

In this chapter, we examine the properties of spiral graphs in relation to our newly formed concept of maximal reducible flow graphs. In particular, we show that spiral graphs are special type of maximal reducible flow graph and then examine some properties of spiral graphs.

7.1 Spiral Graphs are Maximal Reducible Flow Graphs

Result 7.1 *A spiral graph is a maximal reducible flow graph.*

Proof:
We will prove this by induction on n, the number of nodes in the spiral graph. $n = 1$ and $n = 2$ are trivial cases as no additional edge can be added to these spiral graphs. Now consider a spiral graph of $n = 3$ nodes. There are $3! \times 2^{3-1}$ spiral graphs of 3 nodes. Now, it is obvious that we need not consider the isomorphic spiral graphs. As a result, what is needed is just to check the 2^{n-1} spiral graphs when nodes are added in the sequence say $0, 1, \ldots, n-1$. Thus, for 3 nodes, there are $2^{3-1} = 4$ spiral graphs that need to be checked. It can be shown that for all these graphs, addition of the only possible additional edge renders it irreducible. Thus, we have proved the above statement for $n = 3$.

Induction Hypothesis Let us assume that the statement holds true for any spiral graph having $n - 1$ nodes.

Induction Step Now consider a spiral graph on n nodes for $n > 3$. It can be constructed by adding the nth node to a spiral graph of $n - 1$ nodes, say G. In that case any additional edge belonging to G itself will make the original spiral graph (G) and hence the new graph irreducible.

Now, Consider that the nth node is added using rule (2a). (Figure 7.1–a) Any additional edge in the graph must be of the form $n \rightarrow x$, where $x \in G, x \neq n_0$. Let us consider that the addition of such an edge does not cause the graph to become irreducible. Now consider the two edges $x \rightarrow n, n \rightarrow x$. They form a cycle. Now the edge $x \rightarrow n$ is a forward edge by definition of a spiral graph. So, $n \rightarrow x$ must be a back edge. Therefore, x dominates n. So every path from the initial node n_0 to n must contain x. However, there is a direct edge from n_0 to n. Therefore, our original assumption is false.

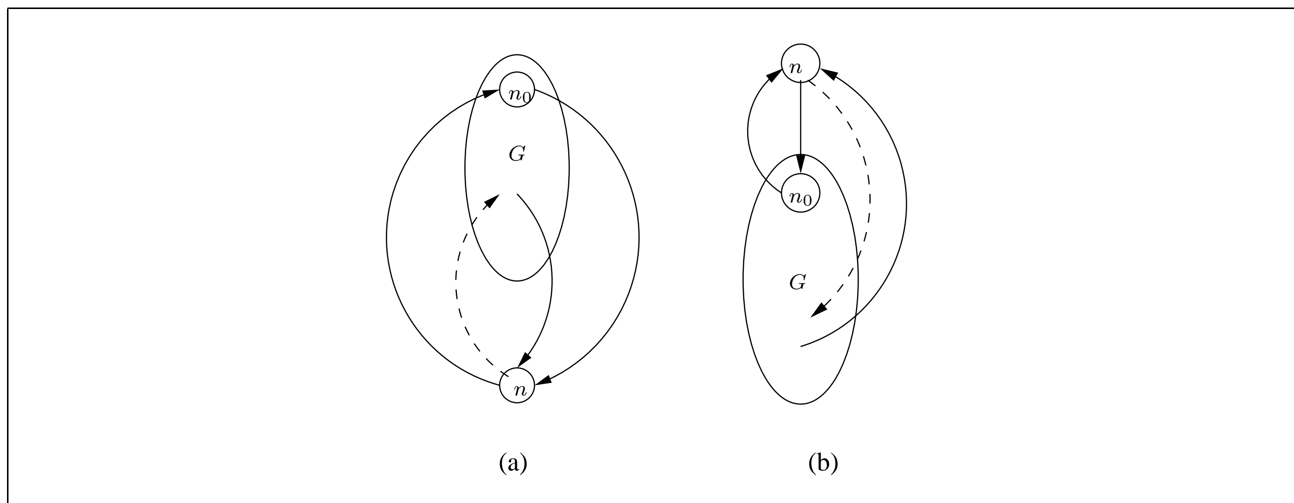


Figure 7.1: Adding a node to a spiral graph

Thus, addition of any additional edge will make the graph irreducible.

Now, consider that the nth node is added using rule (2b). (Figure 7.1–b) Any additional edge in the graph is of the form $n \rightarrow x$, where $x \in G, x \neq n_0$. Let us consider that the addition of such an edge does not cause the graph to become irreducible. Now, we know that G itself is a spiral graph, so that n_0 dominates all the nodes $x, x \in G, x \neq n_0$. However, addition of the edge of the form $n \rightarrow x$ creates a path from the header n to x which does not include n_0. This contradicts the fact that n_0 dominates all node x, $x \in G, x \neq n_0$. Thus our original assumption is wrong. Thus, the addition of any additional edge in this case too renders the graph irreducible. □

7.1.1 How are spiral graphs a special case of maximal rfg's ?

From the above theorem, we can see that spiral graphs are really special type of maximal reducible flow graphs. Consider adding a node using rule (2a). See Figure 7.2. This can be thought of adding the new node as the *rightmost* son of the header, and for each son m of the header, $m \neq (newnode)$ add the edge $m \rightarrow (newnode)$. At the same time, add the edges from the nodes in the trees rooted at all the sons of header to (new node). In Figure 7.2 the original graph is shown in dark edges. It is a spiral graph with nodes added in the order $0, 1, 2, 3$ all using rule (2a). The Figure shows how the addition of node 4 by rule (2a) can now be visualized as adding the node 4 as the son of the header 0. The dotted edges are the ones that will be added in this case.

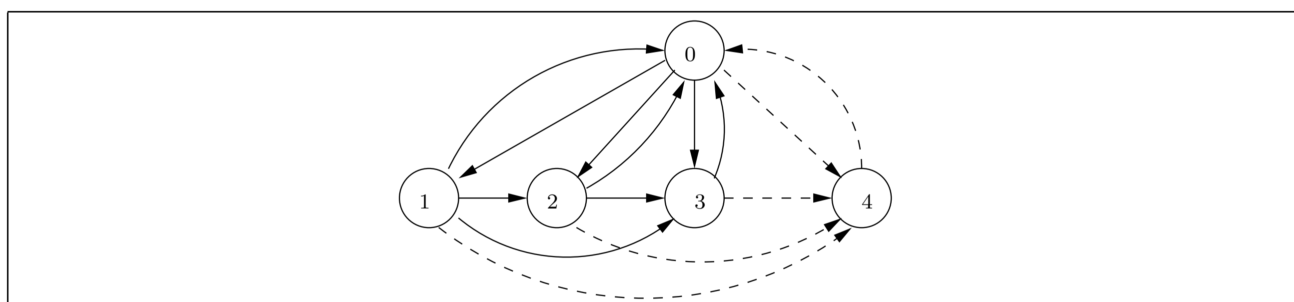


Figure 7.2: Adding a node using rule (2a)

Consider adding a node using rule (2b). See Figure 7.3. This can be thought of as adding the new node as the header of the dominator tree. Then the edge $n \rightarrow n_0$ is added using rule (b), edge

from previous nodes to n are added using rule (a). In Figure 7.3, the original graph is a spiral graph with nodes added in the order $0, 1, 2$ all using rule (2b). The figure shows how the addition of node 3 by rule (2b) can now be visualized as adding node 3 as the header of the original graph. The dotted edges are the one that will be added in this case.

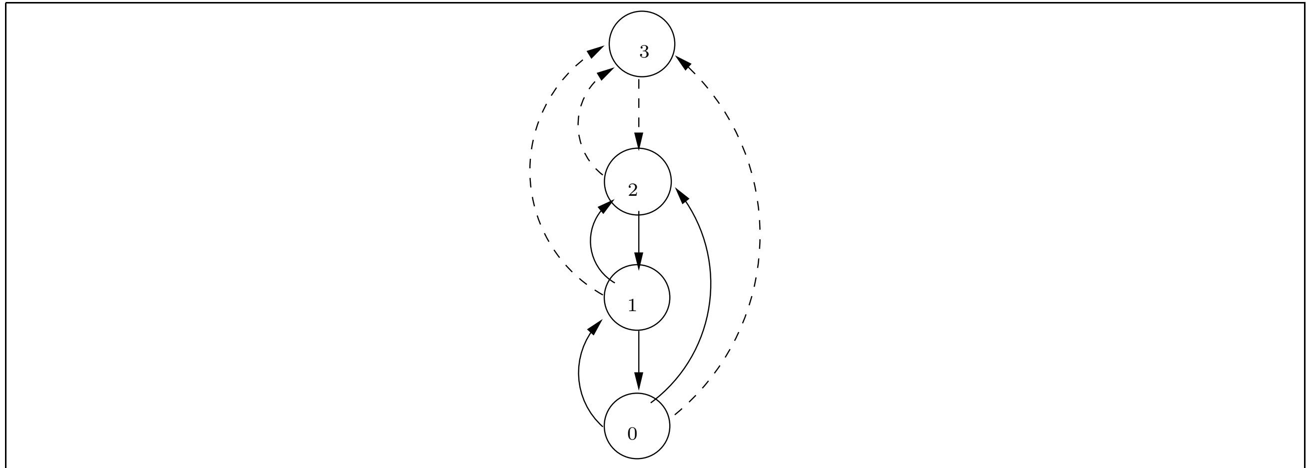


Figure 7.3: Adding a node using rule (2b)

Thus, out of $n + 1$ possibilities, spiral graphs consider only two possibilities of adding a node and when adding using rule (2a) for spiral graphs, the rule (c) is being applied in a still restricted manner. Thus a spiral graph is more restricted that a maximal rfg. In other words, a spiral graph is a special kind of maximal rfg.

7.2 Properties of Spiral Graphs

Result 7.2 *A node added to a spiral graph using rule (2a) is a leaf node of the dominator tree of the spiral graph.*

Proof:

We will prove this result by examining the effect that the addition of a node to a spiral graph has on its dominator tree. If we add a node, say m, to the spiral graph using rule (2a), the node m becomes the rightmost son of the header h of the graph, as shown in Figure 7.4. On the other hand, if the node m is added to the spiral graph using rule (2b), it becomes the father of the header of the graph, as shown in Figure 7.5.

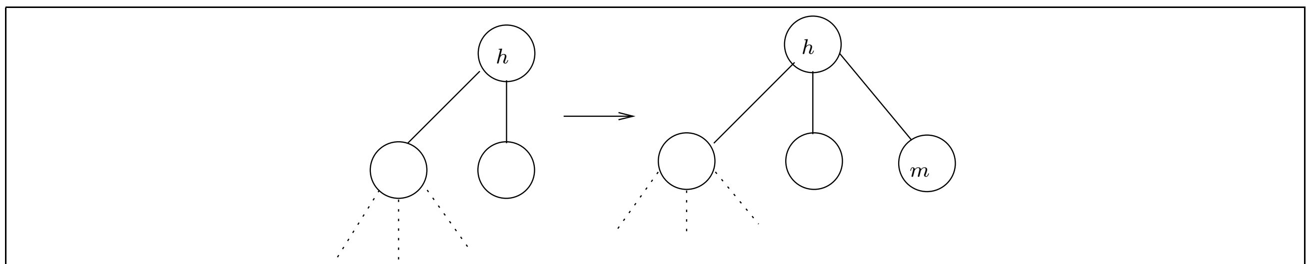


Figure 7.4: A node added using rule (2a)

Now consider that a node m is added to the spiral graph using rule (2a). It will then become the son of the current header of the graph. Now any additional nodes added to the spiral graph (after

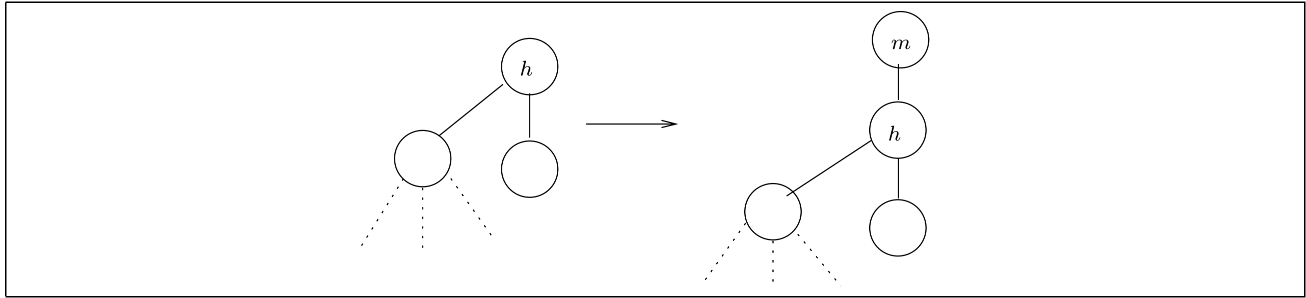


Figure 7.5: A node added using rule (2b)

node m) using rule (2a), will clearly become the sons of the current header h and not of m. The first node , say k, added to the graph by rule (2b) after the addition of node m will become the father of the header h of the graph. Now if any nodes are added using rule (2a), they will become the children of k and so on. Now since a node can be the child of at most one node, we can say that none of the nodes added to the spiral graph after the addition of a node by rule (2a) can become the children of that node. Since node of the nodes added to the spiral graph before the addition of a node can be the children of that node, we can say that if we add a node to a spiral graph using rule (2a), that node will not have any child in the dominator tree i.e. it will be the leaf node of the dominator tree. □

Result 7.3 *A node added to a spiral graph using rule (2b) is always the leftmost child of its father and is never a leaf node of the dominator tree.*

Proof:

The second part of the result is obvious. When we add a node using rule (2b), we add it as the father of the current header of the graph. Thus clearly that node has a son and hence it is not a leaf node of the dominator tree. To prove the first part, let us assume that a node say m was added to a spiral graph using rule (2b). Now if we add any additional nodes using rule (2a), these will become the children of m and hence we need not consider them. In case we do not add any node after m using rule (2b) m remains the header of the graph for which the above statement is trivially true. Now consider a node k added to the spiral graph using rule (2b). k will now become the header of the graph and m will be the child of k. Now, only those nodes that are added using rule (2a) will be in the same level as m. By the previous observation, these nodes will be added as the rightmost sons of k. Thus we can see that m is the leftmost child of k. □

From the above results, we can immediately say that the dominator trees of spiral graphs are characterized by the following properties (Figure 7.6)

- All the nodes at the same level have the same father.
- At each level in the dominator tree, there is *at most one node* which is not a leaf node and that node is the leftmost son of its father and it is added using rule (2b).
- All the other nodes in that level are leaf nodes and are added using rule (2a).

Result 7.4 *The above characterization of spiral graphs and the one stated in Result 6.1 are identical.*

Proof:

We recall from Result 6.21 that if a maximal region R is divided into two subregions R_1 and R_2

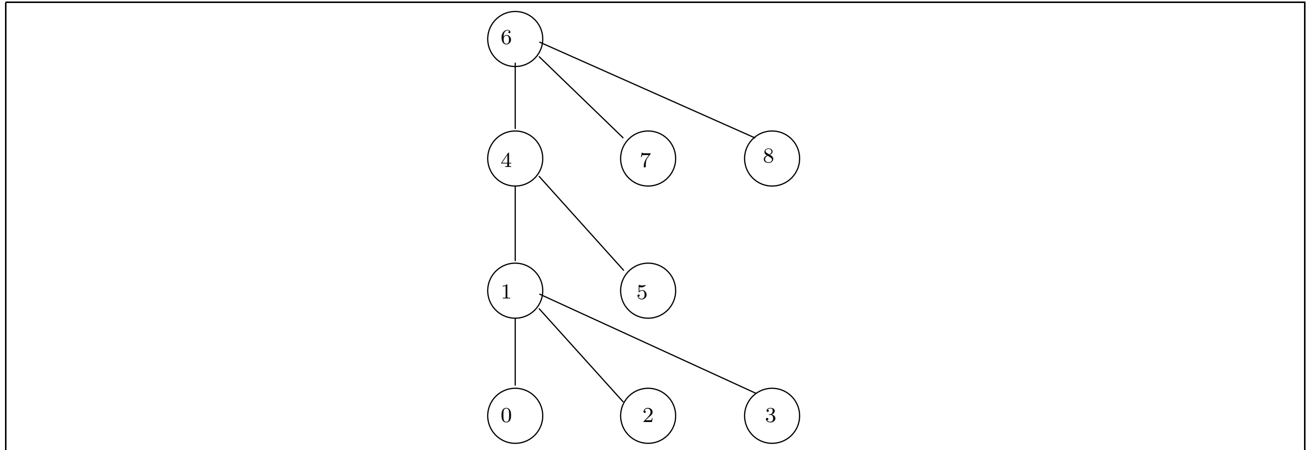


Figure 7.6: The dominator tree of a spiral graph

with headers h_1 and h_2 respectively and R_2 dominates R_1, then h_1 is the rightmost child of h_2 in the dominator tree of R. Conversely, we can say that given a maximal region R, we can divide it into two regions in only one way, such that the rightmost child of the header and all its descendents for one region, say R_1 and the remaining nodes form another region, say R_2, such that R_2 dominates R_1.

Now, in case of spiral graphs, let us first show that if maximal rfg has a dominator tree as characterized above, then it can be divided into a series of singleton feasible regions. Let us consider the header h of the dominator tree . Let $k_1, k_2, \ldots, k_n$ be the children of h in their natural order. To begin with k_n alone is a feasible region, as the remaining nodes i.e. $h, k_1, \ldots k_{n-1}$ and their descendents form a region. Thus the original spiral graph can be divided into two regions of which one is singleton. We apply the same argument to $k_{n-1}, k_{n-1}, \ldots k_2$ and divide each of the intermediate regions into two regions, one of which is singleton. See Figure 7.7. We will finally reach a stage when h

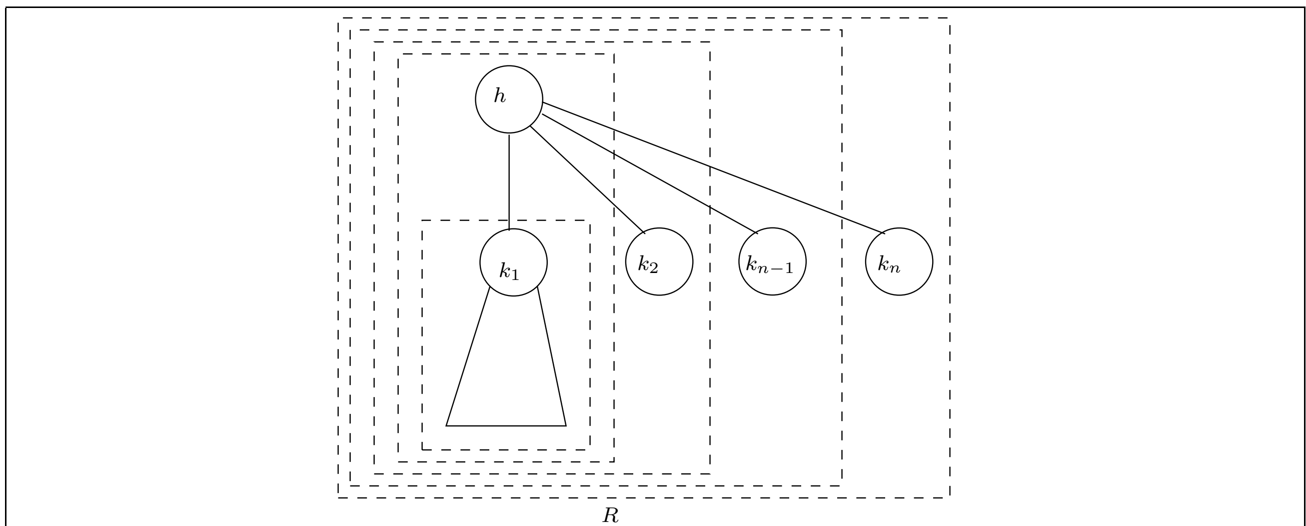


Figure 7.7: Division non-leftmost children

has a single child k_1. In this case, k_1 and its descendents form one region and the header h forms other singleton region. Now we can again apply the same argument to the region consisting of k_1 and all its descendents and show that it can be divided into a series of singleton regions. Thus the characterization of the spiral graphs as stated above is equivalent to the one stated in Result 6.1. This characterization is only one way, i.e. if the given rfg has a dominator tree having the above properties, it surely is a subgraph of some spiral graphs. Else, it may or may not be a subgraph of some spiral

graph. More accurately, a rfg is a subgraph of some spiral graph iff it is a subgraph of some maximal rfg having a dominator tree having the above properties. □

Result 7.5 *The height(h) of the dominator tree of a spiral graph is one more than the number of nodes added to it suing rule (2b).*

Proof:
When we add the first node using rule (1), the height of the dominator tree is 1. After that, every time we add a node using rule (2a), the height of the dominator tree remains unchanged and every time we add a node using rule (2b) the height of the dominator tree increases by 1. Thus, we can say that

$h = 1+$ number of nodes added to the spiral graph using rule (2b). □

Result 7.6 *The order in which the nodes are added to a spiral graph is given by the "inorder" traversal of its dominator tree.*

Proof:
We first define the "inorder" traversal of the dominator tree of a spiral graph as follows

1. Visit the leftmost son
2. Visit the node
3. Visit the remaining children in their "natural" order

Thus, for the dominator tree in Figure 7.6, the inorder traversal of the dominator tree gives the sequence of nodes as 0 1 2 3 4 5 6 7 8. In this case, we can say that node 0 was added using rule 1, node 1 was added using rule (2b), nodes 2,3 by rule (2a), 4 by rule (2b), 5 by (2a), 6 by (2b) and finally 7 and 8 by (2a). □

7.3 Reduction of a Flow Graphs into a Spiral Graph

In the previous section, we examined the characteristics of the dominator trees of spiral graphs. In the paper "Node Listing for reducible flow graphs" Aho and Ullman have shown that when proper regions are replaced by single nodes, any flow graph can be reduced into a spiral graph. In this section we examine exactly how this reduction takes place and which are the regions that are to be replaced for such a reduction.

As seen in the previous section, in any spiral graph, all except the leftmost node in the dominator tree are the leaf nodes. We have also seen that in any reducible flow graph, the subgraph formed by a node and all its descendents is a region. Now, if we are given any reducible flow graph which has a dominator tree in which these exists node(s) which is not the leftmost node and is not a leaf node, then if we replace that node and all its descendents (which for a region) by another node, then we will get a flow graph in which all the nodes that are not the leftmost nodes will be leaf nodes i.e. the resulting flow graph will be a spiral graph. This is shown in Figure 7.8.

Thus, if we define a transformation of replacing the non-leftmost node and all its descendents in the dominator tree by a single node, then repeated application of such a transformation until the transformation cannot be applied will finally give a graph which is a spiral graph with some edges

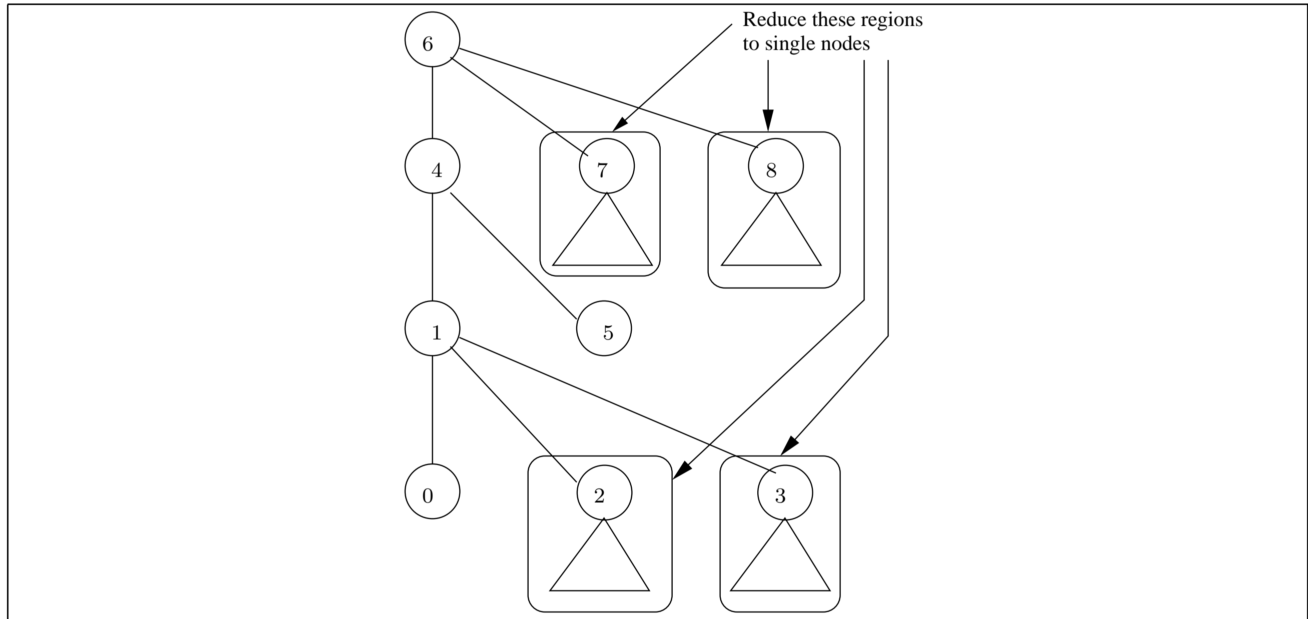


Figure 7.8: Reduction of a flow graph to a spiral graph

removed. Therefore, looking at the dominator tree we can easily figure out which are the regions that are to be replaced by single nodes in order to reduce the flow graph to a spiral graph. These regions are exactly those node and their descendents in the dominator tree which are not the leaf nodes and are not the leftmost children in the dominator tree.

7.4 Algorithm For Computing Sequences of Regions

In the paper "Node Listings for Reducible Flow Graphs" [4], Aho and Ullman have given an algorithm for converting a flow graph into a subgraph of a spiral graph by replacing appropriate regions by single nodes. In the previous section, we have seen how this conversion takes place. They have also given an algorithm for that purpose. We now give an equivalent algorithm.

First, let us see the algorithm given in [4]. Given a reducible flow graph $G = (N, E, n_0)$, we are interested in finding a set of disjoint regions $R_1, R_2, \ldots, R_m$, whose union includes all nodes of G, having the following properties:

1. none of $R_1, R_2, \ldots, R_m$ has more that $\frac{2}{3}k$ nodes ;
2. there is a sequence of regions $S_1, S_2, \ldots, S_m$ such that:
   (a) $S_1 = R_1$,
   (b) for $i > 1$, S_i consists of S_{i-1} and R_i with one as the predecessor of the other,
   (c) S_m is G.
3. The graph formed from G by reducing each of $R_1, R_2, \ldots, R_m$ to a single node with no loops is a spiral graph with zero or more edges removed.

As stated in [4], if T is any region of more that $\frac{2}{3}k$ nodes, then either

1. it is composed of two nonempty regions, one of which has more that $\frac{2}{3}k$ nodes, or
2. it is composed of two regions the larger of which has between $\frac{1}{3}k$ and $\frac{2}{3}k$ nodes.

The algorithm for generating the sequence of pairs $(S_m, R_m), (S_{m-1}, R_{m-1}), \ldots, (S_1, R_1)$ as given in [4] is shown in Algorithm 7.

```
1: T := G ;
2: while T has more than (2/3)k nodes do
3:    let T be composed of regions T1 and T2, with T1 having no fewer nodes that T2 ;
4:    print (T, T2) ;
5:    T := T1 ;
6: end while
7: print (T, T) ;
```

Algorithm 7: Computing the sequences of regions

We now give an equivalent algorithm. The algorithm is based on the fact that a maximal region can be divided into two region in only one way, one region consisting of the rightmost child of the header and its descendents and the other consisting of the remaining nodes. This algorithm expects as input the weighted dominator tree of the flow graph. We first define a weighted dominator tree.

Definition 7.1 (Weighted Dominator Tree) The *weighted dominator tree* of a flow graph is the dominator tree of the flow graph with each node assigned a certain weight as follows:

1. The weight of a leaf node is 1, and
2. The weight of a non leaf node is given by, $w = w_1 + w_2 + \ldots + w_n + 1$, where $w_1, w_2, \ldots, w_n$ are the weights of the children of that node.

We now give an algorithm for finding the sequences of regions that satisfy the condition stated above (Algorithm 8). The algorithm is motivated by the one given by Aho and Ullman. It makes use of the fact that a non-singleton maximal region can be divided into two regions in a single way, as given in Result 6.21.

Since this algorithm is not concerned at all about the actual edges that are present in the flow graph, the above algorithm works for all reducible flow graphs including maximal rfg's. Furthermore, the output of this algorithm is the same for all the flow graph having the same dominator tree even if they have different edges.

As an example, consider the flow graph shown in figure 7.9–a, which is taken from [4]. Its weighted dominator tree is shown in figure 7.9–b.

```
1:  /* h is the root of the weighted dominator tree of the given flow graph */
    /* R_node denotes the region formed by node and all its descendents */
2:  k := w_h ;
3:  while w_h > (2/3)k do
4:     let k_n be the rightmost child of h and w_n be its weight ;
       /* So, R_h can be divided into two regions, one with weight w_n and the other with weight
       w_h - w_n */
5:     if w_n > w_h - w_n then
6:        print (R_h, R_h - R_{k_n}) ;
7:        h := k_n ;
8:     else
9:        print (R_h, R_{k_n}) ;
10:       Remove from the tree rooted at h k_n and its descendents ;
11:       w_h := w_h - w_n ;
12:    end if
13: end while
14: print (R_h, R_h) ;
```

Algorithm 8: Finding the sequences of regions

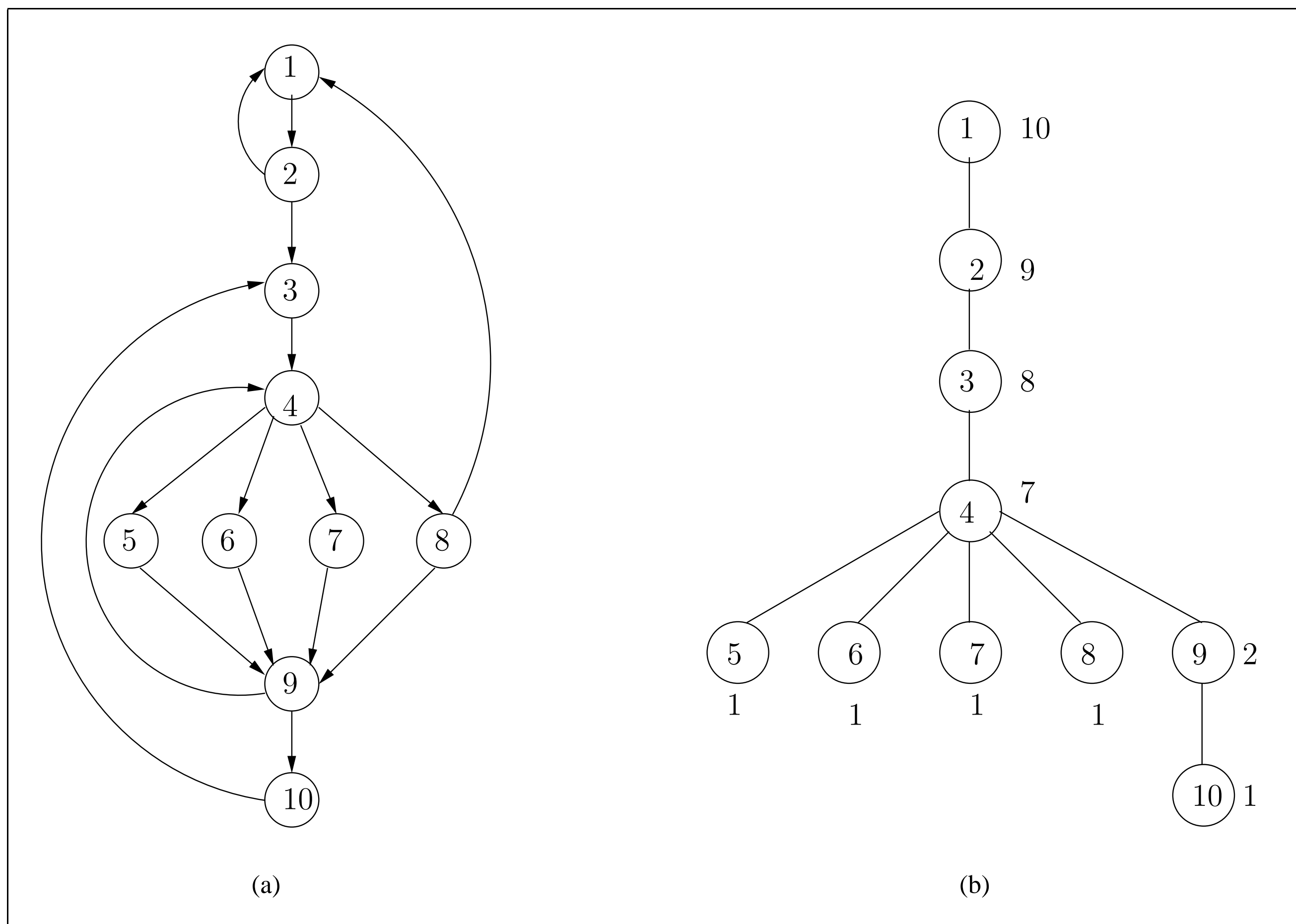


Figure 7.9: Reducing a flow graph into a spiral graph

At the first stage, $w_n = 9$ so the **if** condition satisfies and the pair of regions $(\{1, 2, \ldots, 10\}, \{1\})$ is printed. In a similar fashion, the algorithm proceeds until all the regions have less that $\frac{2}{3}k$ nodes. The calculations done by the algorithm are summarized in the following table. It can be seen that this algorithm produces 5 sequences of regions in the reduction, whereas the one in [4] produces 4 sequences of regions. This is because we have assumed the graph to be a maximal rfg, so at the first stage the regions consists simply the node 1 and not the nodes 1 and 2.

Stage	h	w_h	k_n	w_n	$w_h - w_n$	S_i	R_i
1	1	10	2	9	1	$\{1, 2 \ldots, 10\}$	$\{1\}$
2	2	9	3	8	1	$\{2, 3, \ldots, 10\}$	$\{2\}$
3	3	8	4	7	1	$\{3, 4, \ldots, 10\}$	$\{3\}$
4	4	7	9	2	5	$\{4, 5, \ldots, 10\}$	$\{9, 10\}$
5	4	5	8	1	4	$\{4, 5, \ldots, 8\}$	$\{4, 5, \ldots, 8\}$

7.4.1 Analysis of the Algorithm

As seen in chapter 4, the algorithm by Aho and Ullman enables us to find a node listing of length at most $n + 2.01n \log n$ for any arbitrary reducible flow graph. If the number of edges in the flow graph (e) is bounded by $2n$ i.e. $e \leq 2n$, then they give an algorithm for finding the node listing in time $O(n \log n)$. This bound is required for "efficient" parsing of the given reducible flow graph to produce a sequence of regions.

Here, we analyze the above algorithm (Algorithm 8). It can be easily seen that if the "ordered" dominator tree of the given rfg is given, then parsing that rfg is $O(1)$ i.e. constant time as we already know what the division into the regions will be, as given in Result 6.21. Thus, the only problem is to find the "ordered" dominator tree, which also bounds the time complexity of the above algorithm. In a general setting, we have $e \leq n^2$. Then, the "ordered" dominator tree can be found by using the following steps:

1. Perform the depth first traversal and find the depth first numbers – $O(n)$.
2. Find the acyclic ordering which is the same as the depth first ordering – $O(1)$.
3. Find the immediate dominators for each node – $O(e \log e) = O(n^2 \log n)$ since $e = O(n^2)$.
4. Find the "ordered" dominator tree – $O(n)$, if nodes are visited in the depth first order during the construction of the dominator tree.
5. At each stage, find a parse of the given regions – $O(1)$.

Thus, it can be seen that the time complexity of the above algorithm for finding the sequences of regions and hence the node listing is $O(n^2 \log n)$.

❑❑❑

CHAPTER 8

Miscellaneous Results

This chapter is a cornucopia of smaller but important results that were found during the project but do not fit into any of the chapters.

8.1 About Spiral Graphs

Result 8.1 *Are any two spiral graphs with the same number of nodes isomorphic ?*

The answer is **no**, they are isomorphic if the nodes are added using the same sequence of rules. Can they be isomorphic if the nodes are added using different sequence of rules ? **Investigate**

Result 8.2 *The number of different spiral graphs having n nodes are $n! \times 2^{n-1}$.*

Proof:
Given n nodes, the number of different orders in which these nodes can be added to the spiral graph is $n!$. Of these n nodes, the first has to be added using rule (1), and each of the next nodes can be added using either rule (2a) or (2b), thus giving 2^{n-1} possibilities. Thus, the total number of spiral graphs of n nodes is $n! \times 2^{n-1}$. □

Result 8.3 *The number of edges is a spiral graph having n nodes is $\frac{n(n+1)}{2} - 1$.*

Proof:
We prove the result by induction on n. The result is trivially true for $n = 1$. Now let us assume that the result holds for a spiral graph of $n-1$ nodes. Therefore, a spiral graph of $n-1$ nodes has $\frac{n(n-1)}{2} - 1$ edges. Now, when the nth node is added to the spiral graph, by definitions exactly n edges are added to the graph. Thus, the number of edges in the spiral graph so formed in $\frac{n(n-1)}{2} - 1 + n = \frac{n(n+1)}{2} - 1$.
□

Result 8.4 1. *If a spiral graph is formed only by adding the nodes using either rule 1 or rule (2a), its dominator tree will be of height 2, with the header at level 0 and all the other nodes at level 1.*

2. *If a spiral graph is formed only by adding the nodes using either rule 1 or rule (2b), its dominator tree will be skew and of height $h = n$, where n is the number of nodes in the graph. The node added using rule 1, i.e. the first node will be the leaf of the dominator tree and the last node will be the root of the dominator tree.*

8.1.1 Upper Bound on the Number of Paths

Note In what follows,

1. A 'path' means 'a non-redundant acyclic path beginning with a back edge'
2. A '$2b$-spiral graph' means 'a spiral graph in which all nodes are added by rule $2b$'. The nodes are added in the order $n-1, n-2, \ldots, 0$.

Result 8.5 *In a reducible flow graph of n nodes, the maximum number of paths having b back edges is given by b th entry in $n-1$th row of the Pascal's triangle.*

Example 8.1 The Pascal's Triangle is shown in Figure 8.1.

row = 1, n = 2	1
row = 2, n = 3	1 1
row = 3, n = 4	1 2 1
row = 4, n = 5	1 3 3 1
row = 5, n = 6	1 4 6 4 1
row = 6, n = 7	1 5 10 10 5 1
......	......
......	......
......	....

Figure 8.1: Pascal's Triangle

E.g., for any reducible flow graph of 7 nodes, the max. no of paths having 3 back edges is 10, since the 3rd entry in $(7-1) = 6^{th}$ row of Pascal's triangle is 10. Similarly, the maximum no of paths having 5 back edges is 5.

Proof:

1. Among all reducible flow graphs of n nodes, the $2b$-spiral graph of n nodes will have the maximum number of back edges in any path.
2. So, it is sufficient to show that in a $2b$-spiral graph the number of paths having b back edges is equal to the b th entry in $n-1$th row of Pascal's triangle.
3. We prove stmt.[2] by induction.
   - Statement [2] can be verified for $n = 2$ to 7
   - Let us assume that stmt.[2] is true for some $n = m$.

- We now prove that stmt[2] is true for $n = m + 1$

Case [i] :
For $b = 1$ and $b = n - 1$, there is only one path. So, stmt[2] is proved for $b = 1$ and $n - 1$

Case[ii] :
For $1 < b < n - 1$.
We describe a procedure to get a path of b back edges for $2b$-spiral graph of $(n = m + 1)$ nodes from $2b$-spiral graph of $(n = m)$ nodes. The rules of constructing the paths of b-back edges are:-

(1) Get the paths having $b - 1$ back edges in $2b$-spiral graph of $(n = m)$ nodes. Insert the node m (i.e. $n - 1$) at the beginning of that path. e.g. The path 5 4 3 2 0 1 can be obtained from the path 4 3 2 0 1. In this case, $b = 4$ and $m = 5$.
By using this rule, no. of paths having b back edges created for $2b$-spiral graph of $(n = m + 1)$ nodes = no. of paths having 'b-1' back edges in $2b$-spiral graph of $(n = m)$ nodes.

(2) Get the path of b back edges in $2b$-spiral graph of $(n = m)$ nodes. The node $m - 1$ will be at the beginning of that path. Convert this path into a new path (which begins with a forward edge) as follows. Remove that node from the beginning and shift that node as far as possible towards right, such that in that path the nodes up to $m - 1$ are in ascending order.
e.g. The path 4 2 3 0 1 will be converted to 2 3 4 0 1.
Now, at the beginning of this converted path, insert the node m. e.g. At the beginning of the converted path above, node 5 will be inserted. So, the resultant path will be 5 2 3 4 0 1, which has 2 back edges. Here $b = 2$ and $m = 5$.
By using this rule, no. of paths having b back edges created for $2b$-spiral graph of $(n = m + 1)$ nodes = no. of paths having b back edges in $2b$-spiral graph of $(n = m)$ nodes.

(3) The path having b back edges can be constructed using only the above rules.
Now, the total number of paths having b back edges for $2b$-spiral graph of $(n = m + 1)$ nodes
= (the no. of such paths which can be created by rule 1) + (the no. of such paths which can be created by rule 2)
= (the no. of paths having $b - 1$ back edges when $(n = m)$) + (the no. of paths having b back edges when $(n = m)$)
= ($b - 1$ th entry on mth row of the Pascal's triangle) + (bth entry on mth row of Pascal's triangle)
= (bth entry on $m + 1$th row of Pascal's triangle) ..by definition of Pascal's triangle

Hence proved. □

Result 8.6 *The number of paths in a $2b$-Spiral graph having n nodes is given by 2^{n-2}.*

Proof:
Here, as usual, a path means a non-redundant acyclic path beginning with a back edge.

1. The number of paths in a $2b$-Spiral graph having n nodes is given by the summation of all the entries in the $n - 1$th row of the Pascal's triangle.

2. We prove the required result by induction. The result can be verified for $n = 1$ to 7.

3. Let us assume the result for $n = m$. So, the number of paths in the $2b$-spiral graph having m nodes is 2^{m-2}. So, the summation of all the entries in the $m-1$th row of the Pascal's triangle is 2^{m-2}.

4. Now,
   Number of paths in (2b)-spiral graph having $m+1$ nodes
   = Summation of all the entries in the mth row of the Pascal's Triangle
   = 2×(Summation of all the entries in the $m-1$th row of the Pascal's Triangle
   = $2 \times 2^{m-2}$
   = 2^{m-1}

Hence Proved. □

8.1.2 Method of Finding Subgraphs of Spiral Graphs

Before the development of the theory of maximal rfg's and the properties of spiral graphs, we did not have any theoretical basis for finding whether a given rfg is a subgraph of some spiral graph. This was essential as we had proposed the following proof for $\delta \leq \log n$ for any reducible flow graph,

1. Every rfg of n nodes is a subgraph of some spiral graph of n nodes.
2. The density of a (2b) spiral graph is $\delta = \lceil \log n \rceil$.
3. Of all the spiral graphs of n nodes, (2b) spiral graph has the highest density intuitively.
4. Therefore, $\delta \leq \log n$ for any rfg.

The first point is the crux of the entire proof. Although the first point was later proved to be false, until that, the following method of finding the subgraphs of spiral graphs was used. It works for all the subgraphs of (2b) spiral graphs as well as some (2a)–(2b) spiral graphs. The method is as follows

1. Write down the description of the graph as give on page 123.
2. Start with any arbitrary node of the graph and strike out the entire row for this node in the description of the graph.
3. If the starting node is the successor of a single node, strike out the entire row for that node also.
4. Repeat (4) as long as possible, each time striking out a row and finding out a new node, only of which, the current node is a successor i.e. the current striked out node appears only once in the remaining graph description.
5. If we reach a stage in which the currently striked out node is a successor of 2 or more remaining nodes, verify whether it is a successor of *all* the remaining nodes. If not, the method fails. If yes, order the remaining nodes such that the successors of a node are also the successors of the previous node in the ordering.
6. If all the nodes have been exhausted, the graph is a subgraph of a spiral graph with nodes processed by rule (4) added by rule (2b) and those processed by rule (6) added by rule (2a).

7. If the method fails, try again starting with a different start node. If the method fails for all possible choices of starting nodes, then the graph may or may not be a subgraph of a spiral graph. A more reliable and always correct method is based on the dominator tree characterization and is implemented as the program *isspiral*.

As an example of this method, consider the graph in figure 8.2. Let us start applying the rules

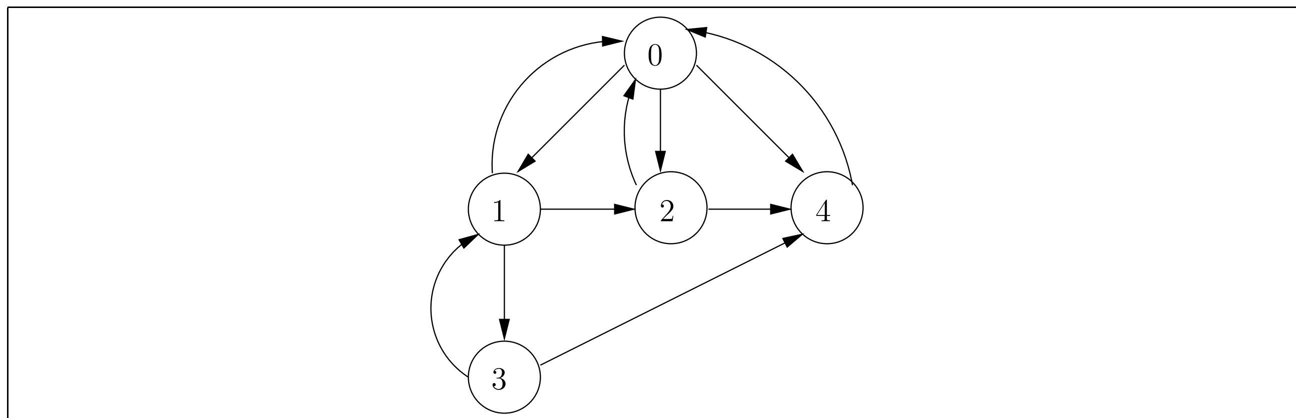


Figure 8.2: Example of the method

beginning with node 3. Initially the graph description is,

0 1 2 4
1 0 2 3
2 0 4
3 1 4
4 0

Now, after striking out the row for node 3, we strike out the row for node 1 as 3 appears only once in the row for node 1, then we strike out the row for node 0 as 1 appears only once in the row for 0. This brings us to the following stage,

~~*0 1 2 4*~~
~~*1 0 2 3*~~
2 0 4
~~*3 1 4*~~
4 0

At this stage, the current node 0 is the successor of 2 nodes viz. nodes 2 and 4. As it is the successor of all the remaining nodes, we order the remaining nodes as $(2, 4)$. Thus, the given graph is a subgraph of a spiral graph with nodes added in the order: 3–(1), 1–(2b), 0–(2b), 2–(2a) and finally 4–2(a). This can also be seen easily from the dominator tree of the graph.

8.2 Generalization of Lemma 2 in Aho and Ullman

We now generalize the lemma 2 given in the paper "Node Listings for Reducible Flow Graphs" by Aho and Ullman [4]. The lemma 2 states

Lemma 2 *Let $P = n_1, n_2, \ldots, n_k$ be an acyclic path in a reducible flow graph and let $n_{i_1-1} \rightarrow n_{i_1}, n_{i_2-1} \rightarrow n_{i_2}, \ldots, n_{i_r-1} \rightarrow n_{i_r}$ be the sequence of back edges along P in that order. Then n_{i_j} dominates $n_{i_{j-1}}$ for all $j, 1 < j \leq k$.*

The above lemma states that the head of a back edge in an acyclic path dominates the heads of all the previous back edges along that path. We now propose a more generalized lemma as follows.

Result 8.7 *The head of a back edge in an acyclic path in a reducible flow graph dominates all the nodes before it in that path. In other words, if $P = (n_0, n_1, ..., n_k)$ is an acyclic path in a reducible flow graph and $n_i \rightarrow n_{i+1}$ is a back edge in P, then n_{i+1} dominates n_j $\forall$ $1 \leq j \leq k$.*

Proof:

Let P be an acyclic path in a reducible flow graph and let P include the back edge $c \rightarrow d$. Then we have to prove that d dominates all the previous nodes along P. Since $c \rightarrow d$ is a back edge, d dominates c by definition of a back edge.

Now consider a node b other than c that occurs before d in P. Clearly, it occurs before c too. Let A be the portion of P from b to c.(Figure 8.3) Let us assume that neither b, c nor d is the initial

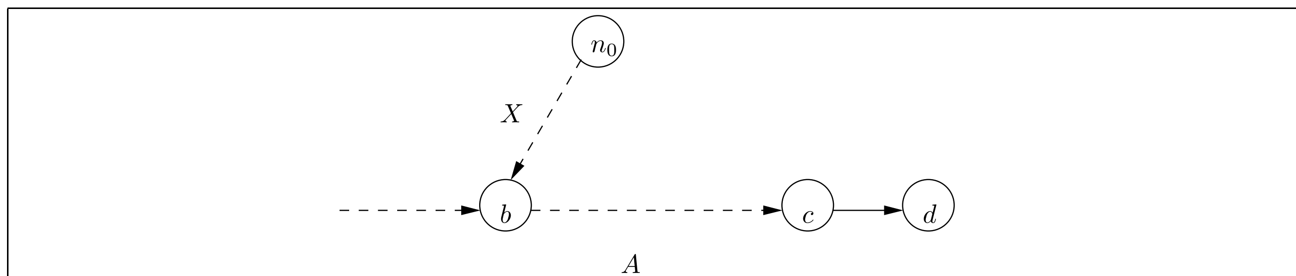


Figure 8.3: An acyclic path in a reducible flow graph

node (n_0) of the flow graph for simplicity. Now let us assume that d does not dominate b i.e. there exists a path from the initial node n_0 to b that does not include d. Let this path be X. Now consider the path $XbAc$. Clearly, X does not include d, nor does A and b,c are not equal to d or else P will not be acyclic. Thus $XbAc$ is a path from the initial node n_0 to c that does not include d. This is a contradiction to the fact that d dominates c. So our original assumption that d does not dominate b is wrong. So d does dominate b. In fact, along the path $XbAc$, d cannot be in A and d is not b and c. So d must be in X. Since X is any arbitrary path from n_0 to b, it follows that d is in every path from n_0 to b. This implies that d dominates b.

Since in the proof, node b was arbitrarily selected, we can conclude that d dominates all the nodes that occur before it along path P. In the proof, we have assumed that neither of b,c and d is n_0. Let us sort out these cases:

1. $b = n_0$ In this case, the path is as shown in Figure 8.4

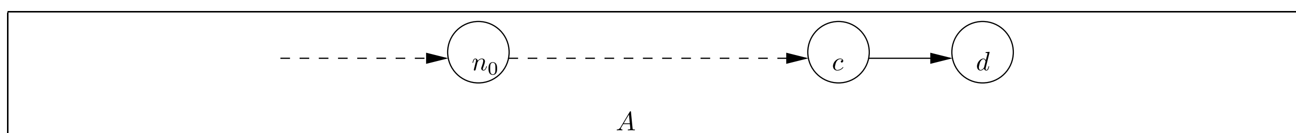


Figure 8.4: Case $b = n_0$

In this case the portion of the path from n_0 to c forms a path from the initial node to c that does not include d. This contradicts the fact that d dominates c. So, b cannot be n_0 in this situation.

In other words, any acyclic path that begins at the initial node cannot have a back edge. Or, in any acyclic path that contains the initial node n_0, the portion of the path after the occurrence of n_0 cannot have a back edge.

2. $d = n_0$ In this case, the path is as shown in the Figure 8.5

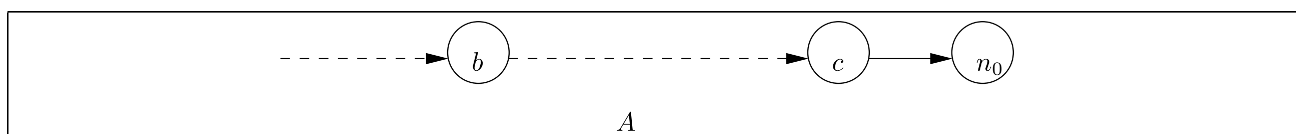


Figure 8.5: Case $d = n_0$

In this case, the initial node n_0 dominates c as well as b since the initial node dominates all the nodes of a flow graph. Thus the case $d = n_0$ is consistent with the given generalization.

3. $c = n_0$ In this case, the path is as shown in the Figure 8.6

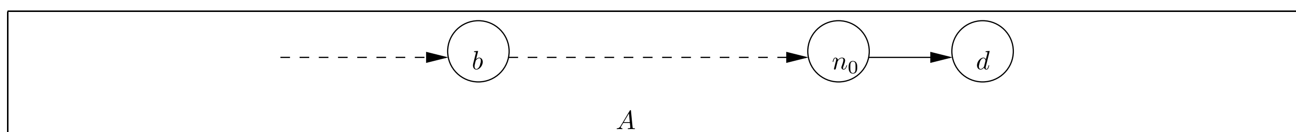


Figure 8.6: Case $c = n_0$

In this case n_0 dom d and d dom n_0. In this case, consider the dominator tree of the flow graph. In this dominator tree, d is an ancestor of n_0 (since n_0 dom d) and n_0 is an ancestor of d (since d dom n_0). Clearly, this is possible only if $d = n_0$. In that case, we have the path as shown in the Figure 8.7 which is cyclic and contradicts the original assumption that P is acyclic. So we

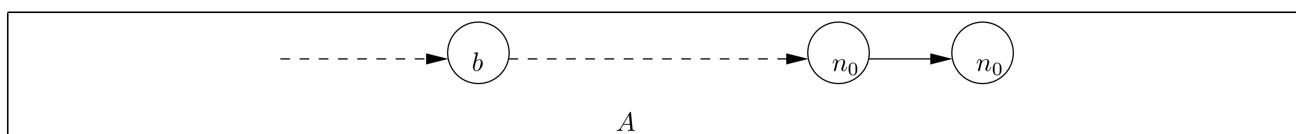


Figure 8.7: Case $c = n_0$

conclude that the case $d = n_0$ is not possible. In other words, the initial node n_0 cannot be the tail of a back edge.

□

Thus, we can see that in an acyclic path, the head of a back edge dominates all the previous nodes in the path. This is a more generalized result than that due to Aho and Ullman. It can now be easily observed that if, along an acyclic path in a reducible flow graph, we traverse the back edge $n \rightarrow d$, then d must be a common dominator of all the nodes that precede d in that path.

8.3 Some More Results

Result 8.8 *In a minimal node listing for any reducible flow graph, now two adjacent nodes are identical.*

Proof:
Obvious. □

Result 8.9 *If a reducible flow graph* (R) *has a subgraph* (R') *of density* $\delta_{R'}$*, then the density of* R *is* $\delta_R \geq \delta_{R'}$.

Proof:
Obvious. □

Result 8.10 *Removal of any back edge from a spiral graph of* 5 *nodes decreases its density by* 1 *from* 3 *to* 2.

Observation There is a relationship between the matrix of levels as produced by the program `matrix` and the "heuristic" node listing produced by the program `sheuristics`. In particular, the matrix of levels is contained in the "heuristic" node listing. Also, for a spiral graph in which all the nodes are added using rule (2b), the node listing and the heuristics are the same.

Observation During the discussion with Prof. Diwan of IIT, Powai, he had suggested to "blow" any reducible flow graph into a spiral graph by the addition of nodes in such a way that all the paths in the original flow graph are covered in the resulting spiral graph. However, it seems that an arbitrary reducible flow graph cannot always be blown into a spiral graph. The intuitive reason behind this statement is as follows:

- Consider a flow graph that is not a subgraph of any spiral graph having the same number of nodes. Its dominator tree has at least one non-leaf non-leftmost node.

- Now, if we add any number of nodes to the graph, it is not going to convert the non-leftmost non-leaf node into a leaf node. Thus, the resulting flow graph will not have a dominator tree that satisfies the characteristics of the dominator trees of spiral graphs.

Result 8.11 *For the maximal reducible flow graph with the dominator tree as shown in figure 8.8, the density is always 1.*

Proof:
Consider the maximal rfg with the dominator tree as shown in figure 8.8.

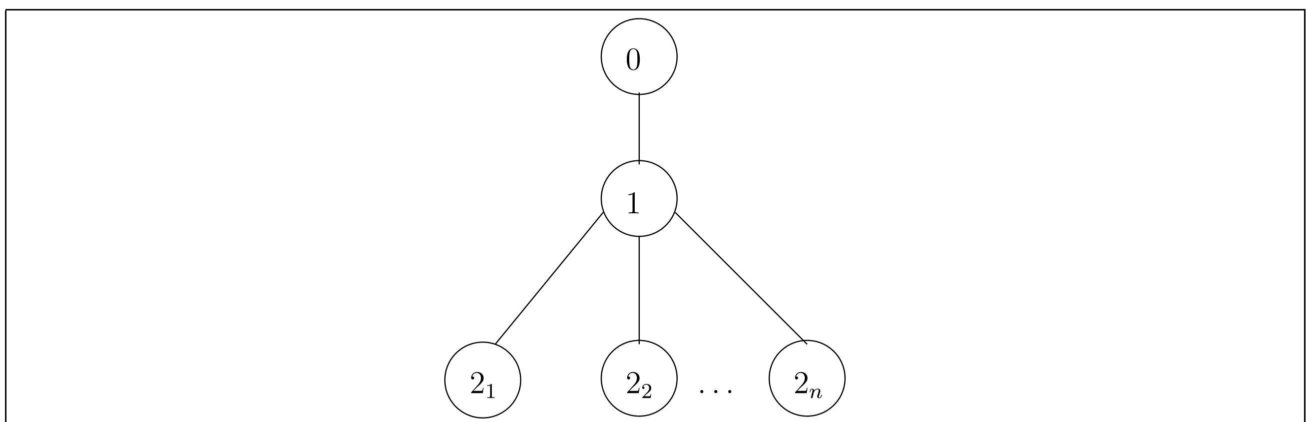


Figure 8.8: A maximal rfg with depth 2

This maximal rfg is similar to a spiral graph with 3 nodes, all added using rule (2b). What we have done is “spilt” the node 2 into a number of children of node 1 in the dominator tree, denoted as $2_1, 2_2, \ldots, 2_n$. It can be now seen that $2_1, 2_2, \ldots, 2_n, 0, 1, 2_1, 2_2, \ldots, 2_n, 0$ is a node listing for this maximal rfg. As each node appears at most twice, this type of maximal rfg has a density 1. □

❑ ❑ ❑

CHAPTER 9

Minimal Reducible Flow Graphs

In this chapter, we define various "transformations"and observe their effect on the densities of the graphs. We see that some of these transformations do not affect the density whereas some of them may either increase or decrease the density depending upon whether a node was added or removed from the graph during the transformation. In each case, we first define the transformation and then state its effect on the density. Based on the observations, we then formulate the concept of a *minimal reducible flow graph*.

9.1 Transformations and Their Effect on Density

9.1.1 T_1 Transformations

Definition 9.1 (T_1 Transformation) We define a T_1 transformation as the removal of zero or more self loop of the form $n \rightarrow n$ from the graph.

Since self loops do not contribute towards the formation of acyclic paths in the flow graphs, we can easily say that a T_1 transformation does not affect the density of a graph. In other words, if G is a graph with density δ and we obtain a new graph G' from G by applying a T_1 transformation, then the density δ' of the new graph will be $\delta' = \delta$.

9.1.2 Inverse T_1 Transformation

Definition 9.2 (Inverse T_1 Transformation) We define an inverse T_1 transformation as the addition of zero or more self loops of the form $n \rightarrow n$ to the graph.

Again, since self loops do not contribute towards acyclic paths in the flow graph, inverse T_1 transformation does not affect the density of the graph.

9.1.3 T_2 Transformation

Definition 9.3 (T_2 Transformation) If there is a node n, not the initial node, that has a unique predecessor, m, then a T_2 transformation is defined as the deletion of n and making all the successors of n (including m, possibly) the successors of m.

We can now intuitively say that a T_2 transformation may result in a decrease in the density of the graph, but it will never result in an increase in the density of the graph.

We now extend the concept of a T_2 transformation by classifying a T_2 transformation into two classes. If a T_2 transformation is applied so that the node m consumes node n, then the T_2 transformation is classified as

- **Type 1**, if the node m is *not* reachable from n in the original graph.
- **Type 2**, if the node m is reachable from n in the original graph.

Observation In a maximal reducible flow graph, every node is reachable from every other node. Hence, a T_2 transformation Type 1 *cannot* be applied to a maximal reducible flow graph.

9.1.4 Inverse T_2 Transformation Type 1

Definition 9.4 (Inverse T_2 Type 1 Transformation) We define an inverse T_2 transformation type 1 as the addition of a node say k to a graph and the addition of the following edges to the graph

1. The edge $m \to k$, where m is some node in the original graph.
2. For all the edges of the form $m \to n$ in the original graph, add the edge $k \to n$ in the new graph, only if the addition of the edge does not make m reachable from k.

Thus, it can be seen that an inverse T_2 transformation type 1 is the expansion of a node m in the original graph by adding a node k, making m as the only predecessor of k and replicating *some* the edges "going out" from m on the new node k. The inverse T_2 transformation type 1 is illustrated in Figure 9.1

We can now say that an inverse T_2 transformation type 1 does not increase the density of the graph. Let G be the original graph with density δ and L as the minimal node listing. Let G' be obtained from G by adding a node k as the successor of m using an inverse T_2 transformation type 1. Then the node listing L' of the new graph G' can be obtained from L by replacing in L every occurrence of m by m, k. This is because any acyclic path that begins with k does not contain m as m is not reachable from k. Thus no node in L' is repeated more than δ times, so that the density of the new graph G' is also δ.

Thus, we can also argue that a T_2 type 1 transformation also does not decrease the density of the graph.

We can also view the T_2 type 1 transformations as follows. If the graph has an *articulation* point that divides the flow graph into two parts, then the density of the graph is the maximum of the density of the two parts. This is illustrated in Figure 9.2.

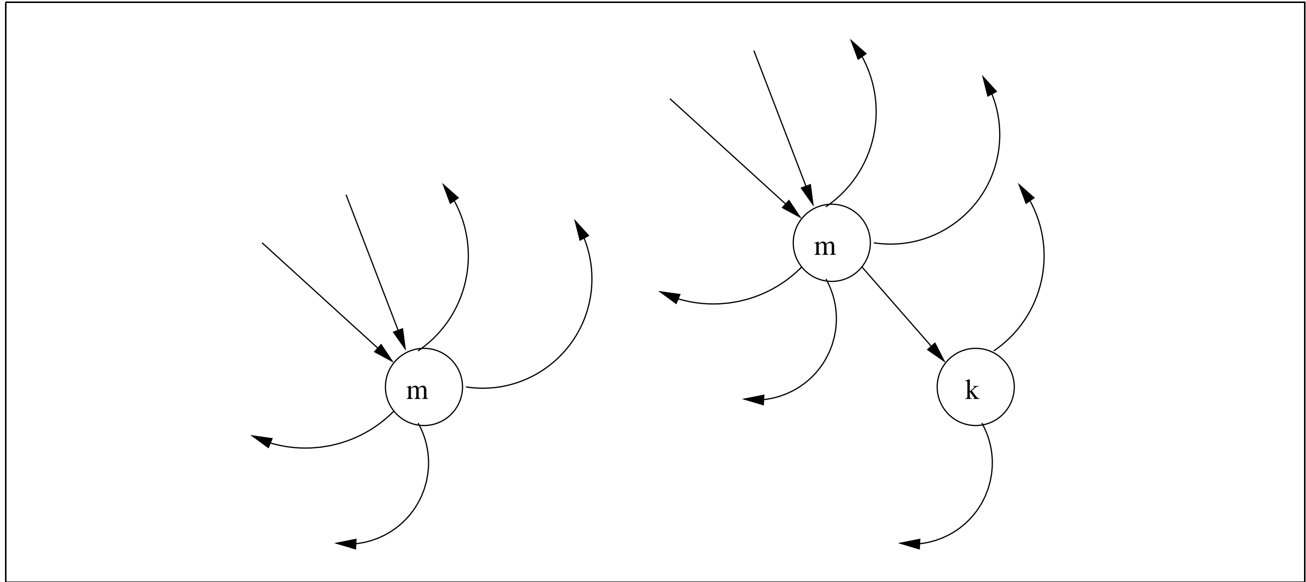


Figure 9.1: Inverse T_2 Transformation Type 1

9.1.5 Inverse T_2 Transformation Type 2

Definition 9.5 (Inverse T_2 Type 2 Transformation) We define an inverse T_2 transformation type 2 as the addition of a node say k to a graph and the addition of the following edges to the graph

1. The edge $m \to k$, where m is some node in the original graph.
2. For all the edges of the form $m \to n$ in the original graph, add the edge $k \to n$.
3. The edge $k \to m$.

An inverse T_2 transformation type 2 is illustrated in Figure 9.3.

We can now say that an inverse T_2 transformation type 2 may result in an increase in the density of the graph. If G is a graph with density δ and we apply an inverse T_2 transformation type 2 by adding a node k as the successor of m, then the node listing L' of the new graph can be obtained from L by replacing the first occurrence of m in L by k, m, k and all the other occurrences of m in L by m, k. This the density δ' of the new graph can be at most $\delta + 1$ i.e. $\delta' \leq \delta + 1$.

Thus, we can also argue that a T_2 type 2 transformation may or may not decrease the density of the graph.

9.1.6 Significance of These Transformations

We know that any reducible flow graph can be reduced to a single node by repeated application of T_1 and T_2 transformations. Thus, inversely, we can say that any reducible flow graph can be constructed from a single node by repeated application of inverse T_1 and inverse T_2 type 1 and 2 transformations. Of these inverse T_1 and inverse T_2 type 1 transformations do not affect the density of the graph. Only an inverse T_2 type 2 transformation may result in an increase in the density of the graph by 1. Now, a single node has a trivial density $\delta = 0$. Thus, the density of a graph is limited by the number of inverse T_2 type 2 transformations required to construct it from a single node. In other words,

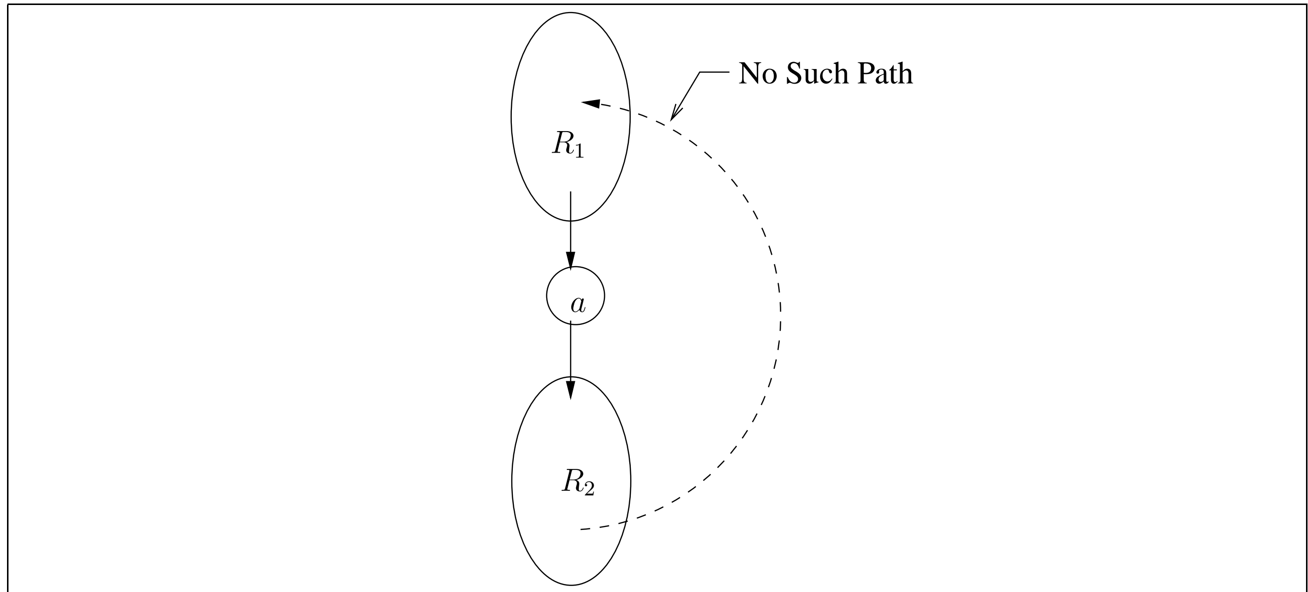


Figure 9.2: An articulation point in a flow graph. $\delta = max(\delta_1, \delta_2)$

$\delta \leq N_{T_{2_2}}$

where $N_{T_{2_2}}$ is the number of inverse T_2 type 2 transformations required to construct the graph from a single node.

9.2 Minimal Reducible Flow Graphs

Definition 9.6 (Minimal Reducible Flow Graph) A *minimal reducible flow graph* of n nodes and density δ is a reducible flow graph of n nodes and density δ such that

1. *The removal of any edge from the graph will either render it disconnected or will reduce its density.*
2. *It cannot be obtained from any minimal reducible flow graph of density δ and less than n nodes by repeated application of any of these transformations* [1]
   (a) T_1 *transformation*
   (b) *inverse* T_1 *transformation*
   (c) *inverse* T_2 *transformation type 1*

A minimal reducible flow graph of n nodes and density δ will be denoted as $min_{(n,\delta)}$.

In essence, minimal reducible flow graphs of density δ depict the "essential" graph structure(s) required to have density δ.

[1]Right Now, we are unaware of any other transformations that will not change the density of a flow graph. As more of such transformations are discovered, they must be added to this list.

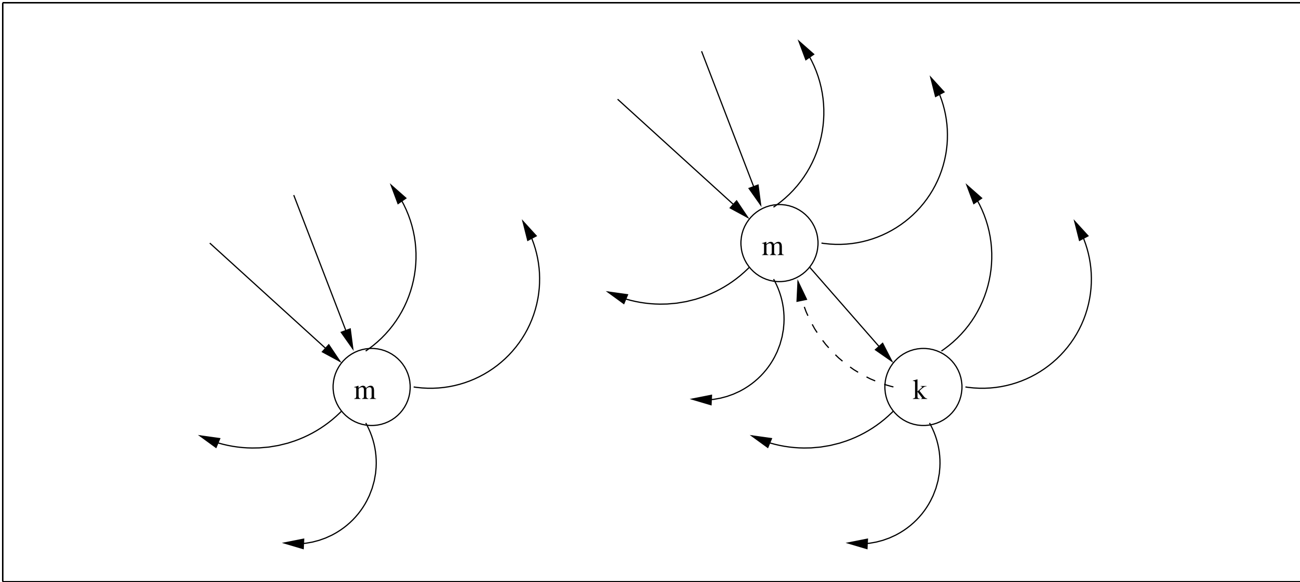


Figure 9.3: Inverse T_2 Transformation Type 2

CHAPTER 10

Conclusion and Future Work

10.1 Conclusion

As a part of this research project, we were able to derive many interesting results regarding *node listings* and their applications to data flow analysis. *Density* is a new upper bound on the length of the node listings proposed by our guide Dr. Khedker. We worked on this concept and were able to prove that $\delta \leq \log n$ for some specific but sufficiently general reducible flow graphs. We also have an "intuitive" proof for $\delta \leq \log n$ for all reducible flow graphs. During this work, we have also formulated a new concept of *maximal reducible flow graphs* and derived some interesting properties. Since any reducible flow graph is a subgraph of some maximal rfg, maximal rfg's can be used to derive or prove upper or lower bounds on properties of reducible flow graphs.

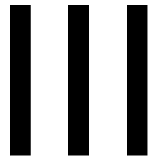
For the experimentation and verification of the theoretical results, we have also developed a library of tools for working with flow graphs and node listings. This library was developed under Linux using Lex and Yacc. For user friendliness, we have also developed a GUI frontend using both GTK+/GNOME and Qt libraries. Apart from this, we have also implemented parallel brute force programs using PVM and MPI on the PARAM 10000 Supercomputer at CDAC, Pune.

10.2 Future Work

Research is a continuous activity. Although we have contributed a handful of results to the ongoing research on density by our guide *Dr. Khedker*, there is still much scope for further work. A theoretical proof for $\delta \leq \log n$ for all reducible flow graphs is still to be derived. For effective use of the node listing based global data flow analysis method in compilers, an efficient algorithm to find minimal node listing is required. Although the concept of maximal rfg is theoretically well developed, its practical applications are yet to be discovered.

One possible way to prove $\delta \leq \log n$ would be to prove it for all maximal rfg as density of maximal rfgs is an upper bound on the density of the contained reducible flow graphs.

❑ ❑ ❑



SOFTWARE ENGINEERING

CHAPTER 11

Organization of The Programs

In this chapter, we describe the general organization of the programs and tools developed as a part of this project. A research project like this one involves a high technical risk with the exact requirements of the software not known before hand. The lack of this knowledge of the exact requirements of the software reduces the scope of the application of the principles of software engineering such as requirement analysis, design etc. Most of the software development was "demand driven," a new idea was thought or the need of verifying some results arose, and then the programs were developed. However, we have tried to maintain the compatibility of all the programs, so for example, a program expecting as input a graph can have its input redirected to the output of the program that produces a graph. Extensive application of the doctrines of software engineering was done during the development of the GUI front end as described in chapter 14.

File Formats

Following the standard UNIX conventions, we decided to keep all the data generated by the program in text files as opposed to binary files. This has the following advantages:

- The input/output files can be easily viewed using a standard text editor. Thus, small changes can be made readily and no separate programs are required to render the data in a human understandable form.
- Manipulation of the text files is often required during the experimentations. In case the data is kept as a text file, UNIX offers text manipulation tools like *Awk* programming language. Thus, the mapping from one format to another can be handled by writing small *Awk* programs rather that coding that in C. This saves time and increases reliability due to the use of existing tools.

Data Input and Output

Each of the programs developed expects its input to be in a certain format. The input was read and parsed using the standard tools *lex* and *yacc*. Each input file can have a comment that begins with a number sign (#) and continues till the end of the line. The programs were designed so that each program reads input from the standard input and writes the results to the standard output. This enables establishing a pipeline of commands using the pipe operator |. However, in some programs, input is

taken through files that are to be specified at the command line. All errors are written to standard error.

Source Compilation

Each of our programs is generally a multifile C program involving lex and yacc specification files, common header files and program specific files. For efficient compilation, each program has an associated makefile in its directory. The make file makes the compilation of the programs efficient, since we don't have to retype the commands and only those parts that are required to be recompiled are recompiled. In each directory, simply typing "make" invokes a program that reads the makefile and issues appropriate commands if some files are out of date.

Source Code Maintainance

In order to maintain the previous versions of the source files, the Revision Control System (RCS) is used. RCS allows us to store all the previous versions of the source file efficiently, since it does not replicate the entire file but only stores the changes. Also, it allows us to retrieve any of the earlier versions of the files and also automatically numbers each version for easy reference. To modify a certain source file say "parse,"we first need to "check out" that file from the RCS system and lock it. This can be done using the following commands:

```
$ co -l parse
```

Now the file "parse" can be modified. Once the changes are made, the file must be 'checked in" the RCS system. This can be done using the command:

```
$ ci parse
```

This command adds the modified file to the RCS system and deleted the file "parse." All the previous and this new version of the file will reside in a file "parse,v" in the same directory.

To simply view the latest version of the file without modifying it, use the command:

```
$ co parse
```

Apart from these basic RCS commands, there are many other command in the RCS system. One of the benefits of the RCS system is that the "make" utility for building the programs knows about RCS. Thus, if it expects to see a file "parse" and does not find one, then it will automatically extract the latest version of the file "parse" from "parse,v" and use it. After the compiling has been done, all these extracted files and other intermediate files are deleted by the command:

```
$ make clean
```

Help and Documentation

All the programs are organized in a directory structure with a separate directory for each program. Each directory has an optional file "algorithm" that explains in brief the algorithm for that program. The program source code itself is also sufficiently documented. Apart from that, online documentation is available in the format of the *UNIX Manual Pages*. The manual pages for all the programs are written using the *groff* text formatting language and are available in the *manpages* directory. They can be accesses by the command like:

```
$ man ./sheuristic.1 ....................... man page for sheuristic
```

❑❑❑

IV

IMPLEMENTATION DETAILS

CHAPTER 12

Brute Force Implementation on PARAM 10000 Supercomputer

The most straightforward method of finding the densities is to generate a list of all the acyclic paths in the graph and then try to "fit" them together manually. However, this method does not always give the minimum density. To find the exact densities, we need to check all possible node listings i.e. we should go by the brute force method. For this purpose, we have implemented parallel programs that find density by an exhaustive search of the solution space. These programs were implemented using both the *Parallel Virtual Machine* and the *Message Passing Interface*. They were executed on the *PARAM 10000 Supercomputer* at the *National PARAM Supercomputing Facility* of the *Center for Development of Advanced Computing*.

12.1 Sequential Brute Force Algorithm

The basic method of finding the node listings by the brute force method is to first find some known density by the heuristic methods and then to try out all the possible combinations of the node listings of that length. Thus, if we have a known density d of the graph then, we consider all the permutations of an array of length $d \times n$. For each permutation, we find the density and then calculate the minimum. Thus, the basic program is given in Algorithm 9

In this algorithm, there are $k = d \times n$ nested for loops, one for each position in the array of the nodes. Since in a minimal node listing, two consecutive nodes will never be the same, all such permutations are eliminated by the if statements following the for statements. It is clear that, because of too many nested for loops, this program will take a long time if run on a sequential machine. Also, the nested loops are essentially independent. This independence motivates us to run the program on a parallel machine so as to speed up the execution. So, we decided to run this program using PVM on PARAM 10000 Supercomputer.

```
/* k is the length of the array for node listing */
/* i.e. k = d * N, N being the number of nodes */
/* The nodes in the graph are numbered as 0, 1, ..., N -1 */
int L[k];

for(L[0] = 0; L[0] < N; L[0]++)
 for(L[1] = 0; L[1] < N; L[1]++)
  if(L[1] != L[0])
  for(L[2] = 0; L[2] < N; L[2]++)
   .
   .
   .
    for(L[k-1] = 0; L[k-1] < N; L[k-1]++)
     if(L[k-1] != L[k-2])
     {
      check whether L has a node listing.
      if( yes )
      {
       find its density in 'this_density'.
       if(this_density < current_density)
       {
        current_density = this_density;
        current_nl = this node listing;
       }
      } /* end if( yes ) */
     } /* end of the body of the for loops */
```

Algorithm 9: Sequential brute force algorithm

12.2 Implementation with Parallel Virtual Machine

PVM (Parallel Virtual Machine) is a software environment for heterogeneous computing (i.e. the underlying processors on which the processes execute may not be identical). It allows a user to create and access a parallel computing system made from a collection of distributed processors, and treat the resulting system as a single virtual machine (hence the name, parallel virtual machine).

The hardware in a user's virtual machine may be single processor workstations, vector machines or parallel supercomputers or any combination of those. The individual elements may all be of a single type (homogeneous) or all different (heterogeneous) or any mixture, as long as all machines used are connected through one or more networks. These networks may be as small as a LAN connecting machines in the same room, as large as the Internet connecting machines across the world or any combination. This ability to bring together diverse resources under a central control allows the PVM user to divide a problem into subtasks and assign each one to be executed on the processor architecture that is best suited for that subtask.

PVM is based on the message-passing model of parallel programming. Messages are passed between tasks over the connecting networks. User's tasks are able to initiate and terminate other

tasks, send and receive data, and synchronize with one another using a library of message passing routines. Tasks are dynamic (i.e. can be started or killed during the execution of a program), even the configuration of the virtual machine (i.e. the actual machine that are part of your PVM) can be dynamically configured.

In order to parallelize the program, we decided to use the master–slave paradigm. out of the $d \times n$ for loops, the master will process some outer for loops and the remaining for loops will be processed by the slaves. Thus, in effect, the master will calculate prefixes of the permutations and send them to the slaves. The slaves will then find out permutations having that prefix and find densities for them. This is shown in figure 12.1.

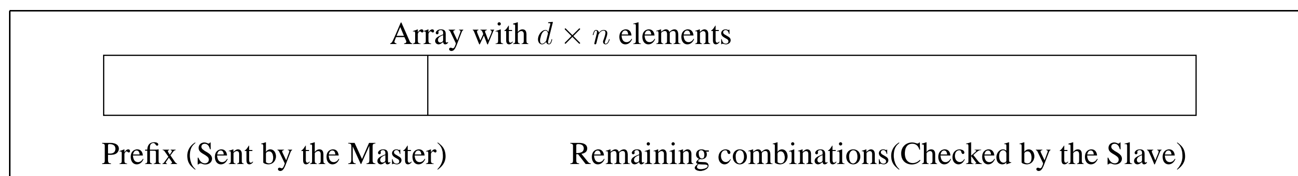


Figure 12.1: Parallelization of the brute force program

If n is the number of nodes and p is the length of the prefix that the master will send to each slave, then the number of different prefixes possible is given by the formula:

Number of prefixes = $n \times (n-1)^{(p-1)}$

In the PVM program, we assign a seperate slave corresponding to each prefix, and so the number of slaves will be the same as the number of prefixes.

The different files which are used in the implementation are:

1. common.h This file contains declarations of some constants that are to be set according to the graph. These constants are:
   - noOfNodesToSendReceive – It specifies the length of the prefix of the nodelisting to be sent from the master to slave
   - noOfNodes – It specifies the number of nodes in the graph
   - fileName – It specifies the name of the file to which the output of the program and the timing information is to be stored
   - maxDensity – It specifies the expected maximum density of the graph
2. constants.h This file specifies the maximum allowed values of some parameters.
   - MAX_NODES – Maximum number of nodes allowed in the flow graph
   - MAX_PATHS – Maximum number of paths allowed in the file
   - MAX_NO_OF_TASKS – Maximum number of tasks which can be spawned
3. gen_density.c This file generates a file density.c, which contains a user defined function density(), which calculates the density of a node listing. The file “density.c” is “# include”ed in worker.c
4. gen_nodestosend.c It generates a file nodestosend.c, which has the function nodestosend(). This function determines what sequence of prefix nodes should be sent to each child task. The file “nodestosend.c” is “# include”ed in master.y

5. master.y Calculates the density of the graph whose non-redundant acyclic paths are stored in temp.paths.

6. worker.c It receives the number of paths and the actual paths from the master. It also receives the prefix of the nodelisting from the master. It then constructs all possible node listings having that prefix and finds their density. Returns the minimum density found to the master.

12.3 Implementation with Message Passing Interface

After the implementation of the brute force program using a Parallel Virtual Machine, it was decided to re-implement the program using the Message Passing Interface library as more speed was required. The MPI Library has much less overhead than the PVM like the absence of a daemon for messaging, use of shared memory for exchanging the messages between processes on the same nodes etc. Secondly, the implementation of the MPI library has been further optimized specially for PARAM at CDAC by removing extra layers. Even with MPI, for large graphs, the program may take a long time to run. For such large experiments, it is necessary to have the following features.

1. The program must periodically save its state in some data file so that in case of reboots, it can begin from the point it left, so that the computations are not lost.

2. For compute intensive algorithms like ours, it would inefficient to spawn as many processes as required. The number of processes must be limited and the work distributed among them.

3. Do as much processing as possible statically, i.e. before the program runs.

In the MPI implementation, we have provided all these features and also have applied some optimizations that may decrease the amount of work to be done by as large as 99.75% This implementation was done by Rahul U. Joshi.

A parallel program consists of a number of processes running on different processors (or nodes) and working simultaneously to solve a problem. For synchronization between these processes, they need to communicate with each other. This *message – passing* paradigm of parallel computing is the most widely used parallel programming paradigm. The Message Passing Interface (MPI) is an application programming interface (API) that defines a standard interface with which message passing programs can be written and run on a variety of distributed systems. In this application, we use the *master – slave* paradigm, with the master distributing the work among the slaves and collecting results from them. Also, since there are two different programs, the program is essentially a *multiple instruction multiple data (MIMD)* style of program, so that we need to use an MPI Application Schema. Information about some representative MPI functions used in the program in given in Appendix C. More information about MPI can be found in [25].

The MPI implementation for finding exact densities is actually a program generator that will generate the programs to find the node listings and the densities by brute force method using the Message Passing Interface. The program generator needs to be supplied with some parameters by setting the values of some constants in the file constants.h. The parameters are as follows:

1. The maximum number of acyclic paths in the graph (MAX_PATHS).

2. The number of nodes in the graph (NUM_NODES).

3. The known density of the graph (DENSITY).
4. The length of the prefix of the permutation that will be sent to the slave (PREFIX_LEN).
5. The number of slaves (NUM_SLAVES).
6. The name of the data file in which the intermediate results will be saved (SAVE_FILE).
7. The name of the source file containing the paths in textual form (PATH_SOURCE).
8. The name of the data file in which the paths will be saved (PATH_DATA).
9. The time for which the update thread of the slave will sleep (UPDATE_SLEEP).
10. The time for which the save thread of the master will sleep (SAVE_SLEEP).
11. The number of nodes of the prefix to be examined for validity (OPTIMIZATION_FACTOR).
12. Flag for whether to favor time or speed when generating the initial results (FAVOR_TIME).
13. Flag to enable/disable debugging messages (NL_DEBUG).
14. Whether *mutexex* are to be used for synchronization among the threads or not (USE_MUTEX).

Once you know these constants, change the #define statements in the constants.h file and then run the shell script gen.sh. This shell script will run the needed program generators and will generate the executable files for the master (master) and the slave (slave). It will then ask for the directory in which to save the generated executables and other files. In that directory, it will save the following files:

1. constants.h – So that you know what the settings were for the programs in the directory.
2. master, slave – The master and the slave executables.
3. init_results – Program to generate the initial permutations and save them in the data file.
4. data_anal – Program to read the saved data and display it.
5. density.schema – The MPI application schema description file.
6. show_paths – Program to display the paths in the paths data file.
7. gen_paths – Program to read the path source and generate path data file.
8. numpaths.h, prefixes.h – header files for these parameters
9. path source, path data and the results data files.

To run the MPI Program, just type mpirun density.schema in the corresponding directory. The master will continuously save the status of each slave in the data file. So, in case the program is interrupted, it can begin at the point it was left in the next run, so that the intermediate results are not lost.

12.3.1 Logic and Algorithm for MPI Implementation

We have seen that, in general we have more prefixes and less slaves, so we have to distribute the prefix to the slaves dynamically and a slave may have to process more than one prefix. Secondly, the other consideration is that the program may run for a long time and in case the system reboots, we will lose our intermediate results and it is not feasible to restart the calculations again. For that purpose, the slaves must periodically convey to the masters the stage of calculation that they have reached and the master must also periodically save this data in some file so that in case of a reboot/crash, the calculations can begin from the point that was last saved, so that not much of the computations are lost.

Here is a description of how the MPI program operates.

1. First, whenever the master starts, we don't want to have the overhead of reading and parsing the text file containing the paths. So we have a separate program for that purpose. It reads the text file (path source file) containing textual description of the acyclic paths in the graph and saves that description into a raw or binary format in the path data file. It also creates a header file numpaths.h that define the actual number of paths in the graph. Thus, now the master, at each startup, need simply read the raw path data into an array and need not parse the input. Secondly, since the number of paths are now known, we need allocate only that much memory and the master need not convey the slaves the number of paths. This makes the program memory efficient.

2. We want that in case of reboot/crash, the program must start from the point it was before the last save. Secondly, the prefixes to be sent to the slaves need to be generated only once, not each time the master starts. For that purpose, the process of generating the initial prefixes has been separated out into another program init_results. This program generated the prefixes to be sent and saves the results into the data file for results. Thus, the master need only read this file into an array and need not calculate the prefixes each time it starts. The init_results program initializes the nodes other that the prefix nodes to 0. When the master runs, it continuously updates these other nodes to whatever permutation the slave last reported. The program is designed such that the next time it runs, it will start from this permutation.

3. Since these results are stored in a binary form, a data analyzer data_anal is provided that can display the saved results file in human readable format.

4. The master and the slave program operate as follows.

    (a) First, the master reads all the paths in the graph and broadcasts them to all the slaves.

    (b) Then, the master send the slaves the permutations to operate on. While sending the permutations, the master may come across a situation that all the slaves have been assigned some permutation to work on and no slave is available to process the next permutation. In that case, it waits for one of the slaves to finish its work and then send it the next permutation. This continues until all the permutations are assigned to some slaves.

    (c) In between, the master may receive update messages from the slaves. It updates the corresponding data in its array. These update messages are sent periodically by an "update" thread in the slave and also when the slave find a node listing of a density lower than what it has found till that time

(d) Finally, when all the permutations are assigned, the master waits for all the slaves to finish and send their results. It then calculates the density and displays it. Also, it saves the result data so that you can later look at it.

(e) A thread in the slave, the “update” thread, continuously send update messages to the master about the status of the current computations.

(f) A thread in the master, the “save” thread, continuously saved the intermediate results of the computations in the save data file so that they are not lost.

The algorithms for the master and the slave are given in the following.

1. Read the paths data from the paths data file and broadcast it to all the slaves using MPI_Bcast().
2. Initialize the state of each slaves (slave[i]) as IDLE.
3. Read the initial permutations from the results data file.
4. **while** There is an unprocessed and unassigned prefix p **do**
   Find an IDLE slave;
   if IDLE slave available **then**
   Send p to the IDLE slave with tag TAG_RESULT using MPI_Send();
   Mark the slave as WORKING and the prefix as assigned;
   else
   Probe for any messages from slaves using MPI_Probe();
   if a TAG_UPDATE message is received, update the result;
   if TAG_DONE message received **then**
   Receive the result using MPI_Recv() and mark it as final;
   Send that slave the prefix p with a TAG_RESULT message;
   Mark prefix p as assigned;
   end if
   end if
   end while
5. /* At this stage, all prefixes are distributed among the slaves */
   still_working = the number of working slaves;
   while still_working $\neq$ 0 **do**
   Probe for a message from any of the slaves;
   if TAG_UPDATE message is received, update the result;
   if TAG_DONE message is received **then**
   Receive the result and mark it as final;
   Send a TAG_END message to that slave;
   Mark the slave as KILLED and reduce still_working by 1;
   end if
   end while
6. Save the final results in the data file. Find the density from the results received from the slaves and display it.

Algorithm 10: Master Program for MPI Implementation

1. Receive the paths and the initial permutation from the master;
2. **while** true **do**
    Probe for any message from the master;
    if TAG_RESULT is received **then**
        Find the density and send a TAG_DONE message to master;
        While finding the density, whenever a new low is attained, send a TAG_UPDATE message to the master;
    else
        /* TAG_DONE message was received */
        Clean up the update thread ;
        exit;
    end if
   end while

Algorithm 11: Slave program for MPI Implementation

12.4 Some Optimization Heuristics

12.4.1 Skipping Redundant Permutations

This is a dynamic optimization in the sense that its efficacy depends upon the actual runtime conditions and cannot be predetermined easily. The basic idea behind this optimization is as follows. Let $L_0, L_1, \ldots, L_{k-1}$ be some permutation that the program checks. Let us assume that the program finds a node listing in this permutation. If m is the length of the node listing found, then let $L_{i_1}, L_{i_2}, \ldots, L_{i_m}$ be the node listing found. Now if $i_m < k-1$, then for all possible combinations of $L_{i_m+1}, L_{i_m+2}. \ldots L_{k-1}$, the same node listing will be found by the program and hence checking all these permutations is redundant. So, checking these permutations must be avoided. For this purpose, after the node listing has been found, we must find the first node from right that was in the node listing i.e find i_m and then for all $L_i, i > i_m$, set $L_i = N - 1$ so that the loops for $L_i, i > i_m$ will be skipped. Thus, the modified part of the program for this optimization is as in Algorithm 12.

```
check whether L has a node listing.
if( yes )
{
 find its density in 'this_density'.
 if(this_density < current_density)
 {
  current_density = this_density;
  current_nl = this node listing;
 }

 /* find the last node in the node listing */
 for(i = k - 1; i >= 0; i++)
  { if L[i] was in node listing, break; }
 i++;
 for(; i < k; i++)
  L[i] = N - 1;
} /* end if( yes ) */
```

Algorithm 12: Skipping Redundant Permutations

This optimization is useful when we have overestimated the *length* of the node listing. It is also useful when we have overestimated the density. It can be easily seen that if we skip the last t nodes of the permutation, we have skipped $(N-1)^t$ permutations. Thus, more the amount of time the program spends in the for loop for finding the last node, more will be the optimization.

12.4.2 Reducing the number of prefixes

This optimization heuristic is based on the basic principle of tailoring the program according to the actual paths to be checked. It can be seen that we can reduce the execution time of the program if we can eliminate some of the prefixes, declaring that these prefixes need not be considered. We first find those nodes that are the first nodes in some path being considered. For example, in the path,

```
0 2 3 4 1; 3; 1;
```

the node 0 is considered in the "initial" traversal, so the first node is node 2. Once we find the set of the first nodes in the paths, we say that a prefix p is a "valid prefix" if it has at least one node which is a first node. Otherwise, we say that the prefix is "invalid prefix." Now, we say that when finding the density by the brute force method, we need not consider the invalid prefixes. The reason is as follows.

Let p be the length of prefix and k be the length of the remaining permutation being considered. Then the permutation will contain p nodes of the prefix followed by the k remaining nodes. Now since p does not have any of the first nodes, when mapping all the path, no node from p will be mapped. Thus, the node listing, if any, being generated from this permutation will be contained in the k remaining nodes. Clearly, this node listing will also be generated by some other prefix which is a valid prefix. For example, if the generated node listing begins with node 0, then the same node listing will also be generated by all the valid prefixes having a 0 in them. Thus, any node listing generated by an invalid prefix is also generated by some valid prefix. Thus, one may discount all the invalid prefixes. This heuristic cannot be applied to spiral graphs. For gadbad.2, this heuristic reduces the number of prefixes from 12696 to 6540.

The above heuristic logic can be extended even further. What we are really doing in the above logic is that we are ensuring that in any valid prefix, at least one of the nodes will be mapped when tracing the paths. Now, suppose that the first node in the prefix to match some node is node 1. Thus the 0th node does not contribute to the node listing, creating a "gap." Thus, we could have done without checking this prefix. Thus, in some way we need to avoid the gaps in the node listings. If the logic for avoiding the gaps is incorporated in the slaves, then they may cause too much overhead and slow down the entire process. Hence, we will incorporate this logic into the prefix generation only (as it will be done only once). The basic theme of this heuristic is to avoid a "gap" in the prefix itself. Thus, when a prefix of length p is generated, we trace all the path into the prefix and see whether all the nodes in the prefix are being traced. If yes, the prefix is valid, else it is not valid. The exact algorithm is given in Algorithm 13.

1. For each path, initialize a pointer ptr(p) that points to the first node in that path.
2. **for** each node n in the prefix **do**
   valid = false;
   for each path p in the graph **do**
      if ptr(p) $\neq$ end of path and ptr(p) $\rightarrow$ node == n **then**
         valid = true;
         ptr(p) = next node of the path p;
      end if
   end for
   if valid == false **then**
      prefix is invalid, discard it;
   end if
   end for
3. If no invalid node n is found, then the prefix is valid.

Algorithm 13: Heuristics for reducing the number of prefixes

In this algorithm, each node n of the prefix is checked for validity. Depending on the amount of optimization desired, one many choose to check lesser nodes for validity when deciding the validity of the prefix. We have implemented these heuristics and the user may choose the amount of optimization by setting the constant OPTIMIZATION_FACTOR (o) in file constants.h. This factor must not be greater than PREFIX_LEN(p). The heuristic will check only the first o nodes out of the prefix of length p when checking the validity of the prefix.

The following table gives some data about the efficacy of this heuristic.

Number of Nodes	Prefix Length	Optimization Factor	Prefixes before optimization	Prefixes after optimization	Reduction %
3	2	2	12	4	66.66
6	5	5	3750	516	86.24
24	3	3	12696	125	99
24	4	4	292008	625	99.79
24	5	5	6716814	3128	99.95

From the table, it can be easily seen that this optimization given a large reduction in the computational effort required. Also, it can been seen that as the number of nodes and the length of the prefix is increased, the reduction in the number of prefixes increases, so the “larger” the problem, more is the optimization. Thus, the heuristic “scales up” as the problem becomes larger, which is a desirable property.

These heuristics were incorporated into the MPI program. For a comparative study of MPI vs. PVM, the MPI program was re-implemented using the same heuristics and the same logic, but using PVM calls for message passing instead of MPI.

❑❑❑

CHAPTER 13

GUI Front End using GTK+/ GNOME

As a part of this project, we have developed a set of tools for working with flow graphs. However, it may be inconvienent for the user to type the commands on the shell prompt. So, for userfriendliness, we have developed a graphical user interface for these tools. This chapter describes the UI developed using the GTK+/GNOME widget toolkits. This UI was developed by Rahul Joshi. Here, we briefly describe the functionality of this UI.

13.1 The Main Window

To start the GUI program, execute the gnome_gui program. When the program starts, it will display a main window as shown in figure 13.1.

There a number of *panels* in which the user can open files for viewing or editing. The normal *open, save, close, new* functionalities have been provided with the File menu. Some of the frequently used commands are accessible from the *toolbar* below the menu bar. The Settings menu lets the user choose the *font size, tab positions* and *word wrap* options. The About menu displays some information about the program. Here is a description of all the menu items in the menu.

- File
  - New – Create a new file
  - Open – Open a file for analysis
  - Save – Save the file to disk
  - Close – Close the current file
  - Close All – Close all the files
  - Exit – Exit the program
- Operations
  - Paths

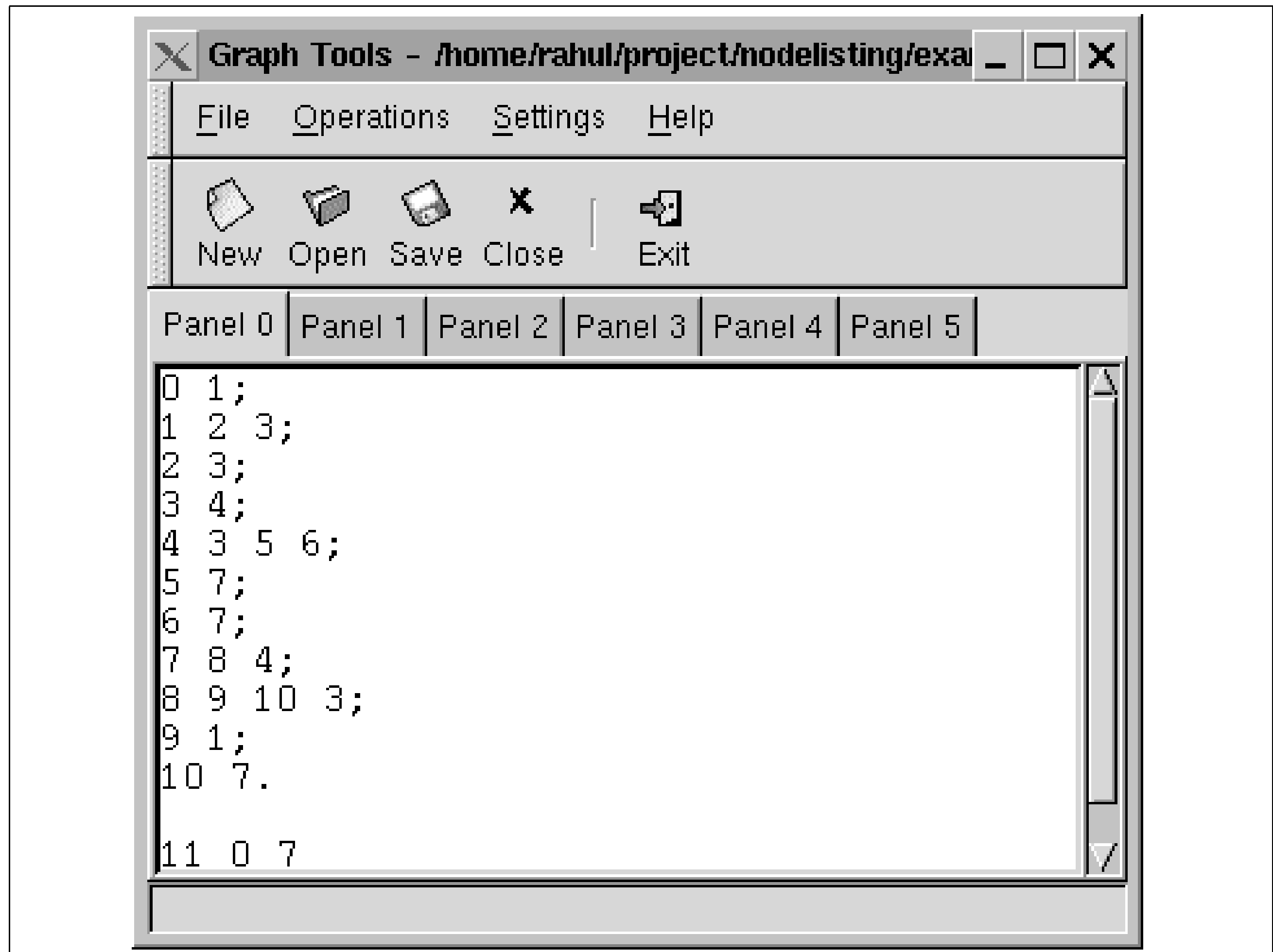


Figure 13.1: The main window of the GNOME UI

- ∗ Back Paths – Generate all the paths in a graph beginning with a back edge
- ∗ All Paths – Generate all the paths in a graph
- ∗ Elimpaths – Generate a list of nonredundant paths

– Maximal RFG

- ∗ Dominator Tree – Find the dominator tree of a graph
- ∗ Depth of Maximal RFG – Find the depth of a Maximal RFG
- ∗ Maximal RFG – Construct a Maximal RFG from its domiantor tree
- ∗ Check for Maximal RFG – Check whether the given graph is a maximal RFG or not
- ∗ Spiral Graph – Construct a spiral graph
- ∗ Check for Spiral Graph – Check whether the given graph is a subgraph of some spiral graph

– Heuristics

- ∗ Original Heuristics – Find node listing using heuristics
- ∗ Majority Merge – Find node listing using majority merge
- ∗ Simplified Heuristics – Find node listing using simplified heuristics

– Node Listings

- ∗ Verify Nodelisting – Verify the node listing

    - Verify Nodelisting (Trie) – verify the node listing using *trie* method
  - Miscellaneous
    - Acyclic Ordering – Find acyclic ordering
    - Dfnize – Dfnize the flow graph
    - Check Reducibility – Check for reduciblity of a flow graph
    - Check Subgraph – Check sibgraph relationship
  - Manual Density – not implemented
- Settings
  - Font – Set font size to small, medium or large
  - Tab Positions – Set tab positions to left, right, top or bottom
  - Word Wrap – Toggle word wrap setting
- Help
  - Help – Displays HTML help about the programs in Netscape Navigator.
  - About – Display information about the application

13.2 Working With Graphs

The various operations that can be performed using this program are available under the Operations menu. These operations correspond to the programs that we have listed previously in chapter 16. When an operation is selected, a popup dialog box is displayed wherein the user specifies the input and output files, as showin in figures 13.2 and 13.2. A brief description of the operation is also mentioned in the dialog box. If the user wishes so, the output will be displayed in the active panel, if the corresponding check box is checked.

The program uses the system() function call of the standard C library to execute the programs. Apart from that, extensive error checking has been implemented to prevent errors.

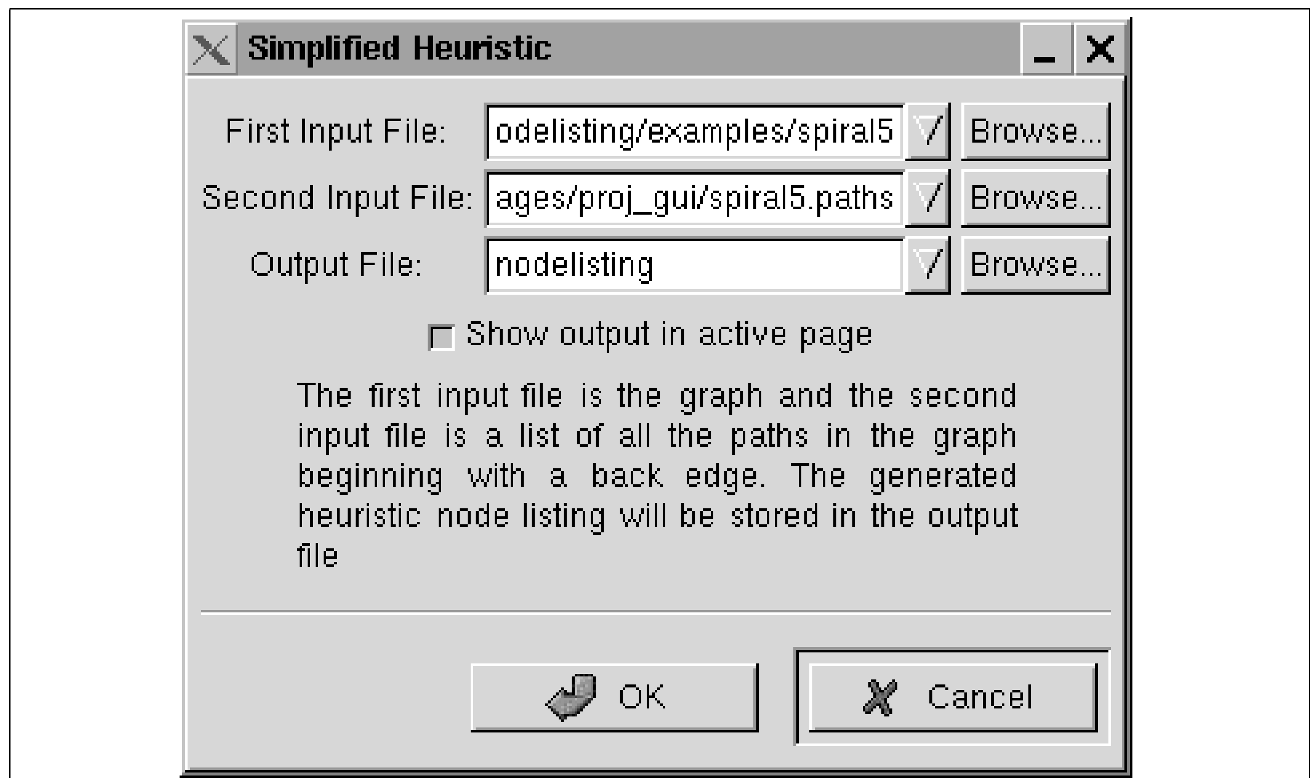


Figure 13.2: A dialog box for two input files

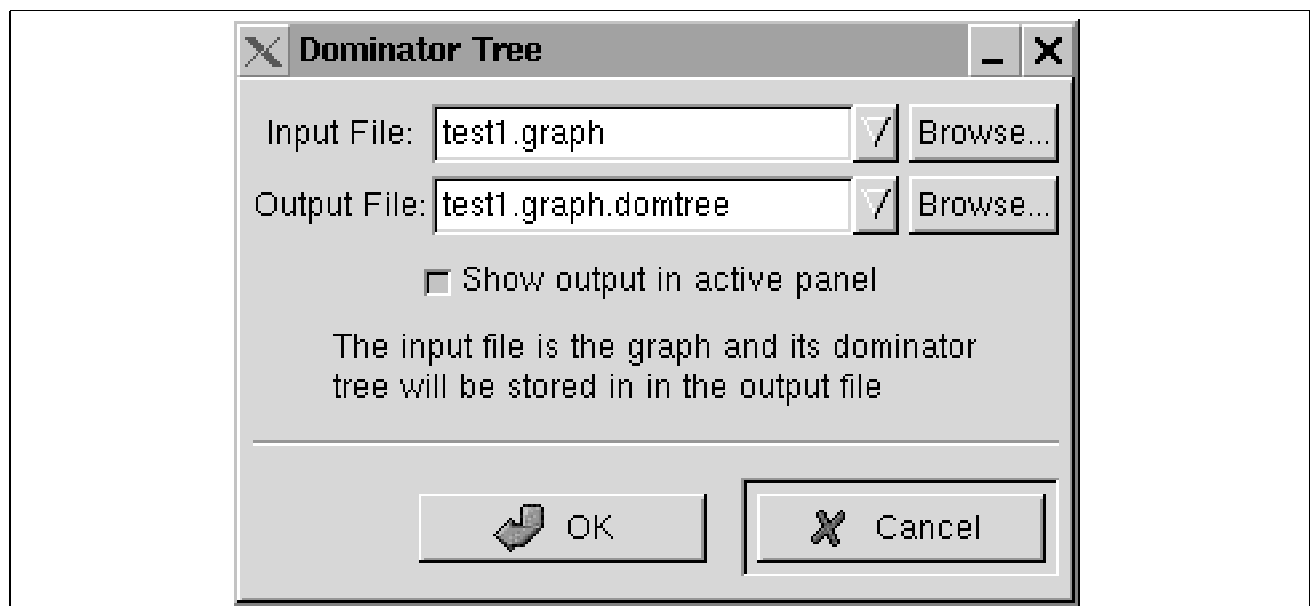


Figure 13.3: A dialog box for a single input file

13.3 Programming Details

This user interface was built using a *widget toolkit* available under Linux called as GTK+, which stands for GIMP Toolkit. This library provides a set of UI objects called *widgets* using which we can build the user interface and a method of installing *callback functions* for responding to the users interactions. Built on the top of the GTK+ library is the GNOME library that provides some additional widgets and routines for building consistent user interfaces under the X Window System. Apart from the pre–built widgets, this library also provides commonly used UI items like file selection dialogs, toolbars, status bars, message boxes. The programming language used was C. We cannot describe in detail the implementation of the UI, but more information about GTK+ and GNOME can be found in [46, 20, 17, 34].

❑ ❑ ❑

CHAPTER 14

Graphical User Interface using Qt

14.1 About Qt

Qt is a cross-platform C++ GUI application framework. It provides application developers with all the functionality needed to build graphical user interfaces. Qt is fully object-oriented, easily extensible, and allows true component programming. Since its commercial introduction in early 1996, Qt has formed the basis of many thousands of successful applications worldwide. Qt is also the basis of the popular KDE Linux desktop environment, a standard component of all major Linux distributions. Qt is a product of Trolltech.
Qt is supported on the following platforms:

1. MS/Windows - 95, 98, NT, and 2000
2. Unix/X11 - Linux, Sun Solaris, HP-UX, Digital Unix, IBM AIX, SGI IRIX and a wide range of others
3. Embedded - Linux platforms with framebuffer support.

14.2 Qt Object Model

The standard C++ Object Model provides very efficient runtime support of the object paradigm. On the negative side, its static nature shows inflexibility in certain problem domains. Graphical User Interface programming is one example that requires both runtime efficiency and a high level of flexibility. Qt provides this, by combining the speed of C++ with the flexibility of the Qt Object Model.
In addition to C++, Qt provides

1. a very powerful mechanism for seamless object communication dubbed signals and slots,
2. queryable and designable object properties,

3. powerful events and event filters,

4. scoped string translation for internationalization,

5. sophisticated interval driven timers that make it possible to elegantly integrate many tasks in an event-driven GUI.

6. hierarchical and queryable object trees that organize object ownership in a natural way.

7. guarded pointers, QGuardedPtr, that are automatically set to null when the referenced object is destroyed, unlike normal C++. Pointers become ”dangling pointers” in that case.

Many of these Qt features are implemented with standard C++ techniques, based on inheritance from QObject . Others, like the object communication mechanism and the dynamic property system, require the Meta Object System provided by Qt's own Meta Object Compiler(moc). The Meta Object System is a C++ extension that makes the language better suited for true component GUI programming.

14.3 Signals and Slots

Signals and slots are used for communication between objects. The signal/slot mechanism is a central feature of Qt and probably the part that differs most from other toolkits. In most GUI toolkits widgets have a callback for each action they can trigger. This callback is a pointer to a function. In Qt, signals and slots have taken over from these messy function pointers. Signals and slots can take any number of arguments of any type. They are completely typesafe: no more callback core dumps!
All classes that inherit from QObject or one of its subclasses (e.g. QWidget) can contain signals and slots. Signals are emitted by objects when they change their state in a way that may be interesting to the outside world. This is all the object does to communicate. It does not know if anything is receiving the signal at the other end. This is true information encapsulation, and ensures that the object can be used as a software component. Slots can be used for receiving signals, but they are normal member functions. A slot does not know if it has any signal(s) connected to it. Again, the object does not know about the communication mechanism and can be used as a true software component. You can connect as many signals as you want to a single slot, and a signal can be connected to as many slots as you desire. It is even possible to connect a signal directly to another signal. (This will emit the second signal immediately whenever the first is emitted.) Together, signals and slots make up a powerful component programming mechanism.
A Small Example
A minimal C++ class declaration might read:
class Foo
(
public:
Foo();
int value() const return val;
void setValue(int);
private:
int val;
);

A small Qt class might read:

```
class Foo : public QObject
(
QOBJECT
public:
Foo();
int value() const  return val;
public slots:
void setValue( int );
signals:
void valueChanged( int );
private:
int val;
);
```

This class has the same internal state, and public methods to access the state, but in addition it has support for component programming using signals and slots: This class can tell the outside world that its state has changed by emitting a signal, valueChanged(), and it has a slot which other objects may send signals to. All classes that contain signals and/or slots must mention QOBJECT in their declaration. Slots are implemented by the application programmer. Here is a possible implementation of Foo::setValue():

```
void Foo::setValue( int v )
(
if ( v != val )
(
val = v;
emit valueChanged(v);
)
)
```

The line emit valueChanged(v) emits the signal valueChanged from the object. As you can see, you emit a signal by using emit signal(arguments). Here is one way to connect two of these objects together: Foo a, b;

```
connect(a, SIGNAL(valueChanged(int)), b,SLOT(setValue(int)));
//note : a and b signify address of a and b respectively
b.setValue( 19 );
a.setValue( 79 );
b.value();
```

Calling a.setValue(79) will make a emit a signal, which b will receive, i.e. b.setValue(79) is invoked. b will in turn emit the same signal, which nobody receives, since no slot has been connected to it, so it disappears into hyperspace. Note that the setValue() function sets the value and emits the signal only if v != val. This prevents infinite looping in the case of cyclic connections (e.g. if b.valueChanged() were connected to a.setValue()). This example illustrates that objects can work together without knowing each other, as long as there is someone around to set up a connection between them initially. The preprocessor changes or removes the signals, slots and emit keywords so the compiler won't see anything it can't digest. Run the moc on class definitions that contains signals or slots. This produces a C++ source file which should be compiled and linked with the other object files for the application.

14.3.1 Signals

Signals are emitted by an object when its internal state has changed in some way that might be interesting to the object's client or owner. Only the class that defines a signal and its subclasses can emit the signal. A list box, for instance, emits both highlighted() and activated() signals. Most object will probably only be interested in activated() but some may want to know about which item in the list box is currently highlighted. If the signal is interesting to two different objects you just connect the signal to slots in both objects. When a signal is emitted, the slots connected to it are executed immediately, just like a normal function call. The signal/slot mechanism is totally independent of any GUI event loop. The emit will return when all slots have returned. If several slots are connected to one signal, the slots will be executed one after the other, in an arbitrary order, when the signal is emitted. Signals are automatically generated by the moc and must not be implemented in the .cpp file. They can never have return types (i.e. use void). Signals and slots are more reusable if they do not use special types. If QscrollBar::valueChanged() were to use a special type such as the hypothetical QRangeControl::Range, it could only be connected to slots designed specifically for QRangeControl.

14.3.2 Slots

A slot is called when a signal connected to it is emitted. Slots are normal C++ functions and can be called normally; their only special feature is that signals can be connected to them. A slot's arguments cannot have default values, and as for signals, it is generally a bad idea to use custom types for slot arguments. Since slots are normal member functions with just a little extra spice, they have access rights like everyone else. A slot's access right determines who can connect to it.
A public slots: section contains slots that anyone can connect signals to. This is very useful for component programming. You create objects that know nothing about each other, connect their signals and slots so information is passed correctly, and, like a model railway, turn it on and leave it running.
A protected slots: section contains slots that this class and its subclasses may connect signals to. This is intended for slots that are part of the class' implementation rather than its interface towards the rest of the world.
A private slots: section contains slots that only the class itself may connect signals to. This is intended for very tightly connected classes, where even subclasses aren't trusted to get the connections right. Of course, you can also define slots to be virtual. It is found to be very useful. Signals and slots are fairly efficient.

14.4 Meta Object Information

The meta object compiler (moc) parses the class declaration in a C++ file and generates C++ code that initializes the meta object. The meta object contains names of all signal and slot members, as well as pointers to these functions. The meta object contains additional information such as the object's class name. You can also check if an object inherits a specific class,
for example:
if (widget→inherits("QButton"))
//it is a push button, radio button etc.

14.4.1 Using the Meta Object Compiler

The Meta Object Compiler is the program which handles the C++ extensions in Qt The moc reads a C++ source file. If it finds one or more class declarations that contain the QOBJECT macro, it produces another C++ source file which contains the meta object code for this class. Among other things, meta object code is required for the signal/slot mechanism, runtime type information and the dynamic property system. The C++ source file generated by the moc must be compiled and linked with the implementation of the class (or it can be included into the class' source file).

1. The class declaration is found in a header (.h) file. If the class declaration above is found in the file myclass.h, the moc output should be put in a file called moc-myclass.cpp. This file should then be compiled as usual, resulting in an object file moc-myclass.o (on Unix) or moc-myclass.obj (on Windows). This object should then be included in the list of object files that are linked together in the final building phase of the program.
2. The class declaration is found in an implementation (.cpp) file If the class declaration above is found in the file myclass.cpp, the moc output should be put in a file called myclass.moc. This file should be included in the implementation file, i.e. myclass.cpp should include "myclass.moc" after the other code. This will cause the moc-generated code to be compiled and linked together with the normal class definition in myclass.cpp, so it is not necessary to compile and link it separately, as in Method 1.

Method 1 is the normal method. Method 2 can be used in cases where one for some reason wants the implementation file to be self-contained, or in cases where the QOBJECT class is implementation-internal and thus should not be visible in the header file.

14.4.2 Automating moc Usage with Makefiles

For anything but the simplest test programs, it is recommended to automate the running of the moc. By adding some rules to the Makefile of your program, make can take care of running moc when necessary and handling the moc output.

14.5 GUI for Node Listing Based Data Flow Analysis

The GUI for this project is written in Qt 2.1.0 under Linux. There is a basic main editor window which provides the user with standard file operations like 'open', 'save', 'save as', 'print', 'close' and so on. The operations to be performed on the graphs and/or their paths are available on the 'graph operations' , 'path operations' and some 'miscellaneous operations' menus on the main menu bar.

The following operations are allowed on graphs

1. listing all paths
2. listing all paths beginning with backedges
3. dominance relationship of a graph
4. reduciblity check

5. acyclic ordering of a graph
6. dfnize a graph
7. check for Max Rfg
8. check for Spiral Graph

The following operations are allowed on paths

1. elimation of redundant paths
2. majority merge heuristic algorithm for node listing

The following operations have miscellaneous inputs

1. construction of Max Rfg from Dominator tree
2. matrix construction
3. node listing of the graph
4. verify a node listing
5. checking for subgraph
6. simpified heuristic algorithm for node listing
7. construction of spiral graph

There is also a Help on programs' menu which mainly provides information on each of the above programs.The file main.htm gives the index to help and can be viewed through any browser. Learning Qt and the development of the graphical user interface was done by Medha Trivedi.

CHAPTER 15

Software Used

In this appendix, we list all the software that was used during the project. The implementation platform for this project was chosen as *Linux*, due to the availability of the large number of programming tools as well as the tools *lex* and *yacc*. Here is a list of all the software that was used.

- Red Hat Linux versions 5.2, 6.1 and 6.2
- Sun Solaris 2.6
- GNU C Compiler (GCC)
- GNU C++ Compiler (G++)
- GNU Debugger (GDB)
- *flex*, fast lexical analyzer generator
- *yacc*, an LALR(1) parser generator
- GNU Make
- RCS (Revision Control System)
- GNU *Awk*, a pattern scanning and processing language (GAWK)
- *groff*, a document formatting system for manual pages
- LaTeX2$_\varepsilon$, a document preparation system
- BIBTeX, bibliography generation tool
- *MakeIndex*, an index generation tool
- *dvips*, a DVI to Postscript converter
- Ghostview, a Postscript and PDF viewer
- *xfig*, Facility for interactive generation of figures under X11
- Qt, a cross–platform C++ GUI application framework
- GTK+, The GIMP Tool Kit, a library for creating GUI's for X Window System.
- GNOME, a set of libraries and application environment for developing consistent GUI's under X Window System
- Glade, a user interface builder for the GTK+ toolkit
- Parallel Virtual Machine, PVM version 3.3
- Local Area Multicomputer (LAM), an MPI programming environment
- LinuxThreads, a POSIX 1003.1c threads library

CHAPTER 16

Program Manual

As a part of this project, we have developed a library of tools for performing various operations on flow graphs and obtaining the data needed for the experiments. The input (be it a graph, a dominator tree, a list of paths etc.) is describes in a text file and it is first parsed by the program. The parser and the lexical analyzer were automatically generated by using the tools *Lex* and *Yacc*. In this appendix, we list the manual for the tools and programs developed during the project. The programs were developed by Rahul U. Joshi.

ENVIRONMENT

Two environment variables affect the functioning of these program. If the environment variable *"NL_DEBUG"* is set and is not *"FALSE"*, debugging mode is enables and the program produced debugging messages. In case debugging mode is enabled, the debugging output goes to the file indicated by the environment variable *"NL_DEBUG_OP"* or to the standard error if *"NL_DEBUG_OP"* is not set.

EXIT CODES

All the programs, on successful execution return an exit code `EXIT_SUCCESS` and on failure return an exit code `EXIT_FAILURE` . These macros are defined in the header file `stdlib.h` as 1 and 0 respectively. These exit codes can then be used in shell scripts to query the exit status of a program by checking,

```
< some command >;

# Check the exit code of the command
if [ $?  -ne 0 ]; then
    echo "$0:  Cannot execute command";
    exit 1;
fi
```

COMMENTS

All the input files, whether containing a graph description, paths etc. are allowed to contain comments. The comments are single line, they begin with a number sign (#) and continue till the end of the line,

as follows

```
# This is a spiral (2b) graph of 4 nodes.
```

ALLBACKPATHS

NAME

allbackpaths - Program to generate all acyclic paths in a reducible flow graph that begin with a back edge.

SYNOPSIS

allbackpaths <**input_graph**> > <**paths_file**>

DESCRIPTION

allbackpaths is a program to generate all the acyclic paths in a reducible flow graph that begin with a back edge. It takes as an input a description of the reducible flow graph from either a file or the standard input. It generates the paths and writes then to the standard output. Along with the actual path, it also prints the number of back edges and forward edges in the path. Each output line consists of a single path in the following form:

`Node1 Node2 ... NodeK ; NoOfBackEdges ; NoOfFwdEdges ;`

BUGS AND INEFFICIENCIES

The program generates the same path more than once in its output, so probably there is some overhead on other programs to look for identical paths in the output and neglect them.

SEE ALSO

allpaths, elimpaths, graphinput

ACYCLIC

NAME

acyclic - Program to generate acyclic ordering of a reducible flow graph.

SYNOPSIS

acyclic <**input_graph**> > <**paths_file**>

DESCRIPTION

acyclic is a program to generate the acyclic ordering of a reducible flow graph. An acyclic ordering of a reducible flow graph is a sequence of nodes (without repetitions) such that all the acyclic paths in the flow graph that start with the initial node (the header) are a subsequence thereof. The program takes as an input a description of the reducible flow graph from either a file or the standard input. It generates the acyclic ordering and writes it to the standard output. Along with the actual acyclic ordering, it also prints the (hypothetical) number of back edges and forward edges in the path. The output has the following format:

```
Node1 Node2 ...  NodeK ; NoOfBackEdges ; NoOfFwdEdges ;
```

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

allbackpaths, elimpaths, graphinput

ALLPATHS

NAME

allpaths - Program to generate all acyclic paths in a reducible flow graph.

SYNOPSIS

allpaths <**input_graph**> > <**paths_file**>

DESCRIPTION

allpaths is a program to generate all the acyclic paths in a reducible flow graph (those that begin with a back edge as well as those that begin with a forward edge). It takes as an input a description of the reducible flow graph from either a file or the standard input. It generates the paths and writes then to the standard output. Along with the actual path, it also prints the number of back edges and forward edges in the path. Each output line consists of a single path in the following form:

```
Node1 Node2 ...  NodeK ; NoOfBackEdges ; NoOfFwdEdges ;
```

BUGS AND INEFFICIENCIES

The program generates the same path more than once in its output, so probably there is some overhead on other programs to look for identical paths in the output and neglect them.

SEE ALSO

allbackpaths, elimpaths, graphinput

BRUTE

NAME

brute - Program to generate a C program that finds the density of a graph by using the brute force method.

SYNOPSIS

brute <**density**> <**no of nodes**>

DESCRIPTION

brute is a program to generate a C program that finds the density of a flow graph by using brute force method i.e. generating all possible node listing and finding the minimum density value. The program takes two inputs from the command line, the known density of the graph and the number of nodes of the graph. The program then generates a C function called "density()" that finds the density of the graph by brute force method. This function consists of (density x no of nodes) nested for loops. To use this function, append it at the end of the file *"template_parse"* present in the same directory as *brute*. Now compile the file using lex and yacc. A template makefile is also present in the directory. This program now takes the list of paths in the flow graph as its input and finds the density. For more information, read the "readme" file in the same directory as *brute*.

BUGS AND INEFFICIENCIES

No bugs known, but the program is *highly* inefficient. For even a moderate number of nodes, it *cannot* be run sequentially on a single processor machine.

SEE ALSO

graphinput, elimpaths, heuristic, readme file in "brute" directory

DEPTH

NAME

depth - Shell script to find the depth of a flow graph

SYNOPSIS

depth < <**paths_file**>

DESCRIPTION

depth is a shell script to find the depth of a flow graph given the list of either all the acyclic paths in the flow graph (as produced by the program *allpaths*) or the list of all acyclic paths that begin with a back edge (as produced by the program *allbackpaths*).

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

allpaths, allbackpaths, elimpaths, graphinput

DEPTH-MAX

NAME

depth-max - Program to find the depth of a maximal reducible flow graph

SYNOPSIS

depth <**dominator_tree**>

DESCRIPTION

depth-max is a program to find the depth of a maximal reducible flow graph given the dominator tree of the maximal rfg. The dominator tree is described in the same format as the output of *dominance*. The program finds the depth of the maximal rfg corresponding to that dominator tree and prints it on the standard output. This program is much more efficient that *depth*, especially for maximal rfgs with large number of nodes, as it may take a long time to generate all the acyclic paths that begin with a back edge.

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

depth, dominance, max-RFG

DFNIZE

NAME

dfnize - Program to dfnize a flow graph i.e. replace each node by its dfn number.

SYNOPSIS

dfnize <**graph_file**> > <**dfnized_graph_file**>

DESCRIPTION

dfnize is a program to dfnize a flow graph i.e. replace each node in the flow graph by its dfn number. The program takes as input the description of the graph to be dfnized and generates an isomorphic graph by replacing each node of the input graph by its dfn number. Such a graph is normally easier to work with because the edge $m \rightarrow n$ is a back edge in such a graph only if $n \leq m$.

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

graphinput

DOMINANCE

NAME

dominance - Program to find the dominance relation in a reducible flow graph.

SYNOPSIS

dominance <**input_graph**> > <**dominator_file**>

DESCRIPTION

dominance is a program to find the dominance relation in a reducible flow graph. It takes as its input the description of the graph as described in *graphinput* and produces as its output the dominator tree of the flow graph. The output of *dominance* has the following format:

```
Node0  list of children of Node0 in dominator tree  ;
Node1  list of children of Node1 in dominator tree  ;
.
.
.
NodeK  list of children of NodeK in dominator tree .

<root of the dominator tree>
```

In this output, the `<root of the dominator tree>` is nothing but the header of the flow graph as specified in the input file. Also the list of the immediate children of a node in the output is arranged according to the "natural" order of the children as described in the definition of a maximal reducible flow graph.

BUGS AND INEFFICIENCIES

There was one, but it has been removed.

SEE ALSO

max-RFG, graphinput

ELIMPATHS

NAME

elimpaths - Program to eliminate redundant paths from the input.

SYNOPSIS

elimpaths <**paths_file**> > <**non_redundant_paths_file**>

DESCRIPTION

elimpaths is a program to eliminate all the redundant paths from the input. That is the program takes a list of paths and from it eliminates any redundant paths and produced a list of non-redundant paths as its output. Thus, in the output, no path is a subsequence of any other path.

BUGS AND INEFFICIENCIES

Many were known, but all have been (hopefully) fixed.

SEE ALSO

Iallbackpaths, allpaths, graphinput

EXE.SH

NAME

Exe.sh - Driver Shell script to find heuristic node listing.

SYNOPSIS

Exe.sh <graph_file>

DESCRIPTION

Exe.sh is a driver shell script to find the heuristic node listing of a flow graph. The graph is specified as a command line argument. The shell script checks whether the graph is reducible, creates a list of non-redundant paths in the flow graph that begin with a back edge and then finds the heuristic node listing.

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

heuristic, reducible, allbackpaths, elimpaths, graphinput

GRAPHINPUT

NAME

graphinput - Description of the input format to describe a graph.

DESCRIPTION

This manual page describes the input format to describe a graph. Many programs take an input graph and produce some results. The input description of the graph has the following form:

```
Node0 <list of successors of Node0> ;
Node1 <list of successors of Node1> ;
.
.
.
NodeK <list of successors of Nodek> .

<NoOfNodes> <Header> <Depth>
```

Comments can also be included in the input file. A comment begins with a sharp (#) and continue till the end of line.

SEE ALSO

allbackpaths, elimpaths, graphinput

HEURSITIC

NAME

heursitic - Program to find the node listing for a reducible flow graph using “heuristic” algorithm.

SYNOPSIS

heuristic <graph_file> <paths_file>

DESCRIPTION

heuristic is a program to find the node listing of a reducible flow graph by using the “heuristic” node listing algorithm as given in Khedker Sir’s “Exe” file. The program takes as input the graph as well as the list of paths in the graph that begin with a back edge. It then prints the generated node listing to the standard output. The node listing is terminated with a period(.). You can directly append the list of all the paths that begin with a back edge to the output and check for the validity of the node listing using the *verifynl* program.

In case debugging is enabled, the program also prints the paths considered, the nodes added during the construction of the node listing as well as the node listing in a tabular form to the debug output stream. The program always gives the density of a flow graph as less then or equal to its depth.

BUGS AND INEFFICIENCIES

No bugs, but the node listing generated by the program is not the “optimal” or the “minimal” node listing.

SEE ALSO

graphinput, allbackpaths, elimpaths, verifynl, sheuristic

ISMAX

NAME

ismax - Program to find whether a flow graph is a maximal reducible flow graph or not.

SYNOPSIS

ismax <graph_file>

DESCRIPTION

ismax is a program to test whether the given flow graph is a maximal reducible flow graph or not. The input is a description of the flow graph of the same format as described in *graphinput*. The program finds whether the input graph is a maximal reducible flow graph or not by adding edges not present in the graph and testing whether the graph becomes irreducible or not. It then prints a message to that effect on the standard output.

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

graphinput, max-RFG

ISSPIRAL

NAME

isspiral - Program to find whether a given flow graph is a subgraph of any spiral graph.

SYNOPSIS

isspiral <dominator_file>

DESCRIPTION

isspiral is a program to find whether the given flow graph is a subgraph of some spiral graph having the same number of nodes. The input is the dominator tree of the flow graph. The format of the dominator tree is the same as the output of *dominance*. The output of the program is a message indicating whether or not the flow graph is a subgraph of some spiral graph.

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

dominance, spiral

MATRIX

NAME

matrix - Program to find the matrix of levels for a reducible flow graph.

SYNOPSIS

matrix <graph_file> <non_redundant_paths_file>

DESCRIPTION

matrix is a program to find the matrix of levels in a reducible flow graph. It takes the graph and the list of non-redundant paths in the graph and constructs the matrix for that graph and prints it on the standard output. The level 0 of the matrix is assumed to have all the nodes in the graph and hence it is not printed. All the other levels are printed. Each output line has the following format:

Level { List of nodes in that level }

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

graphinput, allbackpaths, elimpaths

MAX-RFG

NAME

max-RFG - Program to construct a maximal reducible flow graph.

SYNOPSIS

max-RFG <**dominator_file**> > <**maximal_graph**>

DESCRIPTION

max-RFG is a program to construct the maximal reducible flow graph given the dominator tree of the flow graph. The format of the dominator tree is the same as the output of *dominance*. Thus, the program expects that the children of a node are listed in their "natural" order. The output of the program is a description of the maximal reducible flow graph having the same format as described in *graphinput*. Thus, if we construct the dominator tree of a given reducible flow graph using *dominance*. and then construct the maximal reducible flow graph for that dominator tree, we expect that the original graph will be a subgraph of the maximal reducible flow graph.

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

dominance, graphinput, subgraph

MMHEURISTIC

NAME

mmheuristic - Program to find the node listing using the *majority merge* heuristic.

SYNOPSIS

mmheuristic <**path_file**>

DESCRIPTION

mmheuristic is a program to find the node listing of a flow graph by using the *Majority Merge* algorithm. The input is a list of the paths in the flow graph, in the same format as produced by *allpaths*. The program does not neglect the first node in the path, so it is better to give the input from the output of *allpaths*. It finds the node listing and also the density of that node listing and displays the results on the standard output.

BUGS AND INEFFICIENCIES

None known as of yet, but the node listing produced is not the minimal node listing.

SEE ALSO

heuristic, sheuristic, Exe.sh, allpaths

NL2B

NAME

nl2b - Program to find the node listing of a (2b) spiral graph.

REDUCE

NAME

reduce - Program to reduce a reducible flow graph to minimum number of nodes by using T_2 type 1 transformations.

SYNOPSIS

reduce <graph_input>

DESCRIPTION

reduce is a program to reduce a reducible flow graph to minimum number of nodes by using T_2 type 1 transformations. It takes as input the graph to be reduced and goes on applying T_2 type 1 transformations until no transformation can be applied. It then prints the resulting graph to the standard output.

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

graphinput

REDUCIBLE

NAME

reducible - Program to find whether a flow graph is reducible or not.

SYNOPSIS

reducible <graph_file>

DESCRIPTION

reducible is a program to test whether the given flow graph is reducible or not. The input is a description of the flow graph of the same format as described in *graphinput*. The program finds whether the input graph is reducible or not by repeated applications of T_1 and T_2 transformations. It then prints a message to that effect on the standard output.

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

graphinput

SHEURISTIC

NAME

sheuristic - Program to find the node listing for a reducible flow graph using "simplified heuristic" algorithm.

SYNOPSIS

sheuristic <**graph_file**> <**paths_file**>

DESCRIPTION

sheuristic is a program to find the node listing of a reducible flow graph by using the "simplified heuristic" node listing algorithm. The program takes as input the graph as well as the list of paths in the graph that begin with a back edge. It then prints the generated node listing to the standard output. The node listing is terminated with a period(.). You can directly append the list of all the paths that begin with a back edge to the output and check for the validity of the node listing using the *verifynl* program.

In case debugging is enabled, the program also prints the paths considered, the nodes added during the construction of the node listing as well as the node listing in a tabular form to the debug output stream. The program always gives the density of a flow graph as less then or equal to its depth. The program uses a more efficient and simple algorithm than *heuristic*, and hence the program is much better.

BUGS AND INEFFICIENCIES

No bugs, but the node listing generated by the program is not the "optimal" or the "minimal" node listing.

SEE ALSO

graphinput, allbackpaths, elimpaths,verifynl

SPIRAL

NAME

spiral - Program to generate a spiral graph.

SYNOPSIS

spiral <**node_order**>

DESCRIPTION

spiral is a program to generate a spiral graph given the order in which the nodes are added and also the rules by which the nodes are added. The order and rules are described in the input file in the following format:

```
Initial node
<rule> node1;
.
.
.
<rule> nodek;
```

where <rule> can be either 'a' or 'b' corresponding to rules (2a) and (2b). The "Initial node" is always added using rule 1. Then the nodes are added in the order node1, node2, ..., nodek using the rule corresponding to that node.

The program does not generate the spiral graph directly. Instead, it generates the dominator tree of the spiral graph in which all the children of a node are arranged in their "natural" order. The format of the output dominator generated by the program is described in *dominance*. Once the dominator tree of the spiral graph is generated, the spiral graph can be generated using the *max-RFG* program.

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

graphinput, dominance, max-RFG

SUB_OF_SPIRAL

NAME

sub_of_spiral - Program to check for a subgraph of a spiral graph.

SYNOPSIS

sub_of_spiral <**graph_description**>

DESCRIPTION

sub_of_spiral is a program to find whether the given graph is a subgraph of any spiral graph with the same number of nodes. It is a "brute–force" program, generating all the spiral graph and checking whether the given graph is a subgraph of any one of them. It generates all the spiral graphs having the given number of nodes by,

1. Generating all permutations of the nodes from $0 \ldots n-1$. The nodes will be added to the spiral graph in the order in which they appear in the permutation.

2. For a given permutation, adding the first node using rule (1) and adding the other nodes using all possible combinations of rules (2a) and (2b).

Thus, for n nodes, the program generates a total of $n! \times 2^{n-1}$ spiral graph and checks each of them. This program was written to give an experimental contradiction to the statement "Every reducible flow graph of n nodes is a subgraph of some spiral graph of n nodes."

BUGS AND INEFFICIENCIES

None known as of yet, but the brute force nature makes the program too inefficient. For efficient verification of subgraphs of spiral graph, use the program *isspiral*.

SEE ALSO

graphinput, spiral, isspiral

SUBGRAPH

NAME

subgraph - Program to check subgraph relationship.

SYNOPSIS

subgraph <super_graph> <sub_graph>

DESCRIPTION

subgraph is a program to check subgraph relationship between two graph. It takes as input two graphs, a "super graph" and a "sub graph". It then checks whether the "sub graph" is a subgraph of the "super graph" and prints a message to that effect. The input graphs must be described in the format as described in *graphinput*.

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

graphinput, max-RFG, dominance

VERIFY-ELIMPATHS

NAME

verify-elimpaths - Program to verify the list of non-redundant paths.

SYNOPSIS

verify-elimpaths <**paths_file**>

DESCRIPTION

verify-elimpaths is a program to verify that the list of non-redundant paths as produced by *elimpaths* is indeed correct. Thus, it is used to cross check the output of *elimpaths*. The input file is divided into two sections by a percent (%) sign as follows:

```
<List of non - redundant paths>

%

<List of all the paths>
```

The first section is a list of all the non-redundant paths and the second section is the list of all the paths from which the list of non-redundant paths was produced. The program verifies the list by first reading the list of all the non-redundant paths and checking that no path in this list is a subsequence of any other path in the list. If it comes across such a situation, it declares the list as invalid. After that, it starts reading the list of all the paths and checks that every path in this list is a subsequence of some path in the first list. If it finds some path in the second list which is not a subsequence of any path in the first list, it declares the list as invalid, else it declares the list as valid.

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

allbackpaths, elimpaths

VERIFYNL

NAME

verifynl - Program to verify the node listing.

SYNOPSIS

verifynl <node_list_file>

DESCRIPTION

verifynl is a program to verify that the given node listing for a graph is indeed a node listing for the graph. The input file had the following format:

```
NodeListing.

<List of paths>
```

The first part consists of the node listing terminated by a period(.). Following the node listing is the list of all the paths beginning with a back edge. This list can be the one that is produced either by *allbackpaths* or *elimpaths*. The program verifies the node listing by checking that every path in the list of paths is a subsequence of the node listing. Once the node listing is verified, it also prints the maximum value of the density of the graph by counting the maximum number of times the node appears in the node listing.

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

allbackpaths, elimpaths

VERNL-TRIE

NAME

vernl-trie - Program to verify the node listing.

SYNOPSIS

vernl-trie <**node_list_file**>

DESCRIPTION

vernl-trie is a program to verify that the given node listing for a graph is indeed a node listing for the graph. The input file has the same format as mentioned in *verifynl*. Once the node listing is verified, it also prints the maximum value of the density of the graph by counting the maximum number of times the node appears in the node listing. This program uses a data structure called *trie* for verifying the node listing and theoretically is more faster than *verifynl*.

BUGS AND INEFFICIENCIES

None known as of yet.

SEE ALSO

allbackpaths, elimpaths, verifynl

CHAPTER 17

Algorithms

Finding the acyclic ordering of a flow graph

1. Read the input graph and perform a depth first traversal on it to find the depth first numbers of the nodes.
2. Calculate in "indegree" of each node by without considering the self loops and the back edges. This is because acyclic ordering is found out on a graph with no back edges and self loops are back edge by definition.
3. Find a node in the graph with indegree = 0. If no such node can be found, then probably there is an error in the input graph, it may not be reducible.
4. List that node in the acyclic ordering. Make the indegree of that node as -1 so that it is not processed during the next iteration.
5. Decrease the indegree of all the successors of that node by 1, again without considering self loops and back edges.
6. Repeat the steps 3 to 5 until all the nodes are listed.

Algorithm 14: Finding the acyclic ordering of a flow graph

Checking the reducibility by T_1-T_2 transformation

We know the algorithm for checking flow graph reducibility by repeated application of T_1 and T_2 transformations. In this program, we use a modified form of this algorithm which does not require the application of T_1 transform and the T_2 transforms are applied "incrementally" so that many of the repeated calculations are avoided. Thus this algorithm is more efficient than the earlier one.

1. Read the graph from the input. During the reading step itself, eliminate self loops. Thus here we are applying a T_1 transformation implicitly.(However, after this step, we never need to apply a T_1 transformation.)

2. Initialize the number of predecessors array to 0. This array stores the number of predecessors of each node.

3. **for** each edge $i \rightarrow j$ in the graph **do**
   increment noOfPredecessors(j);
   remember i as the predecessor of j by setting thePredecessor(j) $:=$ i;
   end for

4. Find a node n not the initial node and having a singe predecessor m. If such a node cannot be found, end the computation.

5. Remove the node n from the graph and set noOfPredecessors(n) = 0.

6. **for** each edge $n \rightarrow i, i \neq m$ in the graph **do**
   if there is an edge $m \rightarrow i$ **then**
   decrement noOfPredecessors(i);
   end if
   add the edge $m \rightarrow i$ in the graph;
   set thePredecessor(i) $:=$ m;
   end for

7. **if** there was an edge $n \rightarrow m$ in the original graph **then**
   decrement noOfPredecessors(m);
   if n is the predecessor of m that we have remembered **then**
   thePredecessor(m) $:=$ some other predecessor of m.
   end if
   end if

8. Go to step 4.

Algorithm 15: Checking reducibility by T_1-T_2 transformations

Generate all the paths in a reducible flow graph

In this algorithm, we maintain a stack of nodes where each element on the stack is of the form `<node, count>` where count is the position of node in the acyclic path formed. Whenever a node is added to the path, we push all its successors on the stack with a count as 1 greater than that of the node, since the successor will be immediately after the node in the path.

1. Read the graph and store it in an adjacency matrix.
2. Initialize and empty stack of nodes, say `S`.
3. For all nodes in the graph, push `<node, 0>` on stack `S`.
4. Go on doing the following steps until the stack becomes empty.
5. Pop a `<node, count>` from the stack.
6. If `node` is already included in the path, we have obtained an acyclic path in the flow graph, so print it.
7. See the next node on the top of the stack (do not pop it) and flag all the nodes that will not be in the next path as false. Go to step 5.
8. Add `node` to path at the position indicated by its `count` and flag the node as true to indicate that it is present in the path.
9. Push all the successors of `node` on the stack with count one greater than the count for node.
10. Go to step 4

Algorithm 16: Generate all paths in a flow graph

Generate all the paths in a reducible flow graph that begin with a back edge

In this algorithm, we maintain a stack of nodes where each element on the stack is of the form `<node, count>` where count is the position of node in the acyclic path formed. Whenever a node is added to the path, we push all its successors on the stack with a count as 1 greater than the that of the node, since the successor will be immediately after the node in the path.

1. Read the graph and store it in an adjacency matrix.
2. Perform a depth first traversal of the graph and find the "dfn" numbers for all nodes, as given in [5].
3. Initialize and empty stack of nodes, say `S`.
4. For all nodes in the graph, push `<node, 0>` on stack `S`.
5. Go on doing the following steps until the stack becomes empty.
6. Pop a `<node, count>` from the stack.
7. If `node` is already included in the path, we have obtained an acyclic path in the flow graph, so print it.
8. See the next node on the top of the stack (do not pop it) and flag all the nodes that will not be in the next path as false. Go to step 5.
9. Add `node` to path at the position indicated by its `count` and flag the node as true to indicate that it is present in the path.
10. Initialize hasBackEdge = false, to indicate that currently we have found no back edge beginning from node.
11. Push all the successors of `node` on the stack with count one greater than the count for node. Since we are interested only in paths that begin with a back edge, when pushing the second node (`count` = 1), see to it that we push only those successors of node for which the edge $node \rightarrow successor$ is a back edge. If any such successor of node is found, make hasBackEdge true.
12. Once pushing of all the successors of node is completed, in case we were pushing the successors of the first node and hasBackEdge = false, then there was no back edge from the node, so there is no acyclic path beginning with a back edge that starts from this node. So, flag this node as false to indicate that it is not present in the path.
13. Go to step 5

Algorithm 17: Generate all paths in a flow graph that begin with a back edge

Finding the node listing of the graph by simplified heuristics

In this algorithm, the paths are processed in a decreasing order of the depth of the paths. For each path, we assign a “span” (a sequence of nodes between two consecutive back edges) to the corresponding level. This algorithm will always give density $\leq$ depth of the graph. It is much simpler than the one implemented in heuristic since the reverse traversal is avoided and the algorithm is much easier to understand.

1. Read the graph and find the depth first numbers (dfn) for each node by performing a depth first traversal.
2. Read all the paths that begin with a back edge and store them in a linked list. While reading the paths, do not store the first node in the path as it will be covered in the initial traversal.
3. Now add the path to the node listing one by one beginning from paths having maximum depth to paths having minimum depth (1).
4. To add a path to the node listing:
   (a) Initialize level $:=$ 0
   (b) add the first node to level = 0
   (c)
   ```
   for all the further nodes do
     if we traversed a back edge then
       level := level + 1;
       add node to level;
     end if
   end for
   ```

Algorithm 18: Simplified heuristic node listing

Finding the dominators

Finding the dominators is the similar to the iterative data flow analysis. The method is akin to forward data flow analysis with intersection as the confluence operator. For each node n, we calculate $D(n)$, the set of dominators of n. In the real program, we also perform an "inverse transitive closure" operation to find the immediate dominators of the nodes so as to construct the dominator tree.

1: D(n_0) $:= \{n_0\}$;
2: **for** n in $N - \{n_0\}$ **do**
3: $\quad D(n) := N$;
4: **end for**
/* End Initialization */
5: **while** changes to any $D(n)$ occur **do**
6: $\quad$ **for** n in $N - \{n_0\}$ **do**
7: $\quad\quad D(n) := \{n\} \cup \left(\bigcap_{p \in predec(n)} D(p) \right)$;
8: $\quad$ **end for**
9: **end while**

Algorithm 19: Dominator computing algorithm

V

REFERENCES

Appendix A- Some Referenced Papers

In this appendix, we present the following main papers that were referenced during the project work.

- *Node Listings Applied to Data Flow Analysis* by Ken Kennedy [22].
- *Node Listings for Reducible Flow Graphs*[1] by Al Aho and J.D. Ullman [4].

[1]We would like to thank the Academic Press for allowing us to reprint this paper

Node Listings Applied to Data Flow Analysis

K. W. Kennedy
Rice University

ABSTRACT

A new approach to global program data flow analysis which constructs a "node listing" for the control flow graph is discussed and a simple algorithm which uses a node listing to determine the live variables in a program is presented . This algorithm combined with a fast node listing constructor due to Aho and Ullman has produced an $O(n \log n)$ algorithm for live analysis.The utility of the node listing method is demonstrated by an examination of the class of graphs for which "short" listings exist. This class is quite similar to the class of graphs for "understandable" programs.

1. Introduction

When analyzing computer programs at compile time for code optimization, one encounters a class of problems which requires the construction of "data flow" information from the control flow graph. Many algorithms for the construction of such information have appeared in the literature most of these solve a specific problem and require $O(n^2)$ "extended" or "bit vector" steps where n is the number of vertices in the control graph. Recently Ullman[U] published an algorithmic method which can be used to solve a number of these problems in $O(n \log n)$ extended steps.

The purpose of the current work is to follow a new line of attack: from the control flow graph of a program we construct an intermediate representation of the flow called "node listing", which is then used to solve data flow problems.

This paper is primarily devoted to an introductory treatment of the basic concepts surrounding the node listing. The final section, however, investigates the class of graphs for which the node listing method produces linear time algorithms for data flow analysis.

2. Control Flow Analysis

The flow analysis of a program usually begins with the program expressed in some intermediate text which is scanned and subdivided into *basic blocks,* sequence of instructions which are always executed in order. After the last instruction in a block, control may transfer to any one of a number of basic blocks called *successors* of the block just executed.

We may represent a program by its *control flow graph* (or *flow graph*) in which each node represents a basic block and each edge represents a possible block-to-block transfer. A flow graph is therefore a triple $G = (N, E, n_0)$ where

1. N is a finite set of nodes,
2. E is a finite set of edges (a subset of $N \times N$), and
3. n_0 is the unique node with no predecessors, called the *program entry node*.

In these terms we may define the set of successors $S(x)$ of a block x:

$$S(x) = y \in N | (x, y) \in E.$$

Similarly,we define $P(x)$, the set of *predecessors* of x:

$$P(x) = y \in N | (y, x) \in E.$$

A *path* in the control flow graph from n_1 to n_k is a sequence of nodes $(n_1, n_2, \ldots, n_k)$ such that $(n_1, n_{i+1}) \in E, 1 \leq i < k$. The *path length* of $(n_1, n_2, \ldots, n_k)$ is $k - 1$, the number of edges used in traversing the path. A *simple path* is a path in which no node is repeated except possibly the first (which may also be the last). A *simple cycle* is a simple path $(n_1, n_2 \ldots, n_k)$, $k > 1$, such that $n_1 = n_k$.

3. Data Flow Analysis

We shall consider data flow analysis by studying a representative problem, that of locating "live" variables within a program. Given an item(variable X), which is defined at various points in a program. We wish to determine for each point p in the program flow graph whether or not X will be used after control leaves p. We say that X is *live at* p if it can be used again and *dead at* p otherwise. The "live" information would be useful in register allocation for example, since the value of a variable which can never be used again need not be saved.

For simplicity, we formulate another version of this problem: for each block b in the program, determine the set *live(b)* of variables X for which there is a path from the entry point of b to a use of b, which path is *X-clear* (contains no redefinition of the variable X). We can now translate this problem to one of solving a system of boolean equations. Let *inside(b)* be the set of variables X which are live on entry to block b because there is a use of X within b which is not preceded by a redefinition. Let *thru(b)* be the set of variables X for which there exists an X-clear path through b. Note that the sets inside(b) and thru(b) can be computed by a local examination of block b.

Now there exists an X-clear path from the entry of b to a use of X if and only if there exists such a path to a use within b or through b to a successor of b and there to a use. In equation form:

$$(\star) live(b) = inside(b) \cup \bigcup_{x \in S(b)} (thru(b) \cap live(x)) \ .$$

The solution to this system of equations clearly provides a solution for our simplified live analysis problem.

Several methods for the solution of $(\star)$ have been proposed [Ke1,Ke2,HU1,Ki]; curiously, all the proposed methods involve $O(n^2)$ algorithms even though Ullman's method [U] provides an $O(n \log n)$ algorithm for a related but somewhat different problem. The author knows of no published algorithm to solve $(\star)$ which is faster than $O(n^2)$; in particular the method of Ullman cannot be adapted to live analysis in a straight forward way.

We here present the simplest way of solving the live equations in hopes of finding its sources of inefficiency. Suppose we begin with all the live sets empty and iterate through the graph applying equation $(\star)$ until none of the live sets change. This method adapted from Hecht and Ullman [HU1].

Algorithm A: Live Analysis (Hecht and Ullman)

Input:

1. A flow graph $G = (N, E, n_0)$, $|N| = n$, with the nodes numbered from 1 to n in some suitable manner. Each node is referred to by its number.

2. Sets *thru(j)* and *inside(j)* for all j, $1 \leq j \leq n$

Output: Sets live(j), $1 \leq j \leq n$.
Method:

1. Initially, let live(j) = inside(j), $1 \leq j \leq n$.

2. Do step c for $j = 1, 2, \ldots, n$ in order. If any live(j) changes for any j, repeat step b; otherwise halt.

3. Apply equation $(\star)$ to block j.

Hecht and Ullman have shown that this algorithm is correct and that step b will be executed at most $d+2$ times where d is equal to the maximum number of backward branches in any simple path within the flow graph [HU1]. Since step c requires $|S(j)| + 1$ bit-vector steps and it is applied to every node in the graph for each execution of b, a total of $(d + 2)(|E| + |N|)$ bit-vector steps are required. It can be shown that [HU1,Ke3] that d is $O(|E|)$ in the worst case so the algorithm may require $O(|E|^2)$ bit-vector steps ($O(n^2)$ whenever the graph is restricted so that $|E| \leq k|N| = kn$ for some fixed k). There seem to be two major areas of inefficiency in this approach. First, an extra pass through the program is required to discover that no sets have changed. This and the testing for changed sets on every pass produces a lot of unnecessary work that might be avoided if we could somehow know when to halt the iteration. Second, iteration over every node in each pass seems to be overkill. The problem is to iterate exactly enough times to transmit information along any simple path in the program. The "node listing" method attempts to overcome these difficulties.

4. Node Listing

To gain some motivation for the node listing concept, consider the flow graph in figure 1 below.

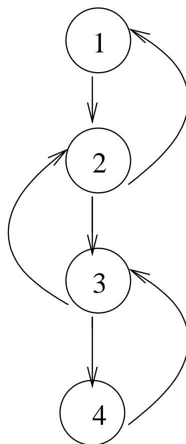


Figure 1. A ''(4,1) climbing graph''.

On this graph the Hecht-Ullman approach to live analysis could require 5 iterations over the four nodes. However, if equation $(\star)$ were applied to nodes in order (1,2,3,4,3,2,1) then all the live sets would be correctly computed and fewer than two iterations through the graph would be required. The node listing is really a specification of the order in which an equation is to be applied to nodes of a graph. As it happens, these specifications are often quite short and the algorithms which use them are correspondingly fast.
We define a *basic path* in a flow graph $G = (N, E, n_0)$ to be a simple path $(n_1, n_2, \ldots, n_k)$ such that no shorter simple path from n_1 to n_k is contained as a subsequence of $(n_1, n_2 \ldots, n_k)$. In figure 2 below, the simple path (1,2,3,4,5) is not a basic path because it contains the basic path (1,2,4,5).

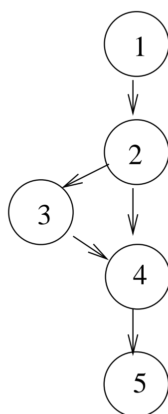


Figure 2. Simple and basic paths

In dealing with data flow problems, we will be concerned with basic paths, because as we shall see, longer simple paths can add nothing to the data flow information being propagated. For example, suppose that we are trying to determine if variable X is live in node 1 of figure 2. Suppose also that there is an exposed use of X in node 5. We wish to know if there is an X-clear path from node 1 to node 5. In determining this, we need not consider the path (1,2,3,4,5) because if this path is X-clear then the path (1,2,4,5) must be X-clear also. The basic path restriction is weaker than a simple path restriction and should allow us to find shorter listings.
We now define a *node listing* for a program flow graph $G = (N, E, n_0)$ to be a sequence

$$l = (n_1, n_2 \ldots, n_m)$$

of nodes from N (where nodes may be repeated) such that every basic path in G is a subsequence of l. That is, if

$$(x_1, x_2 \ldots, x_k)$$

is a basic path in G, then there exist indices

$$j_1, j_2 \ldots, j_k$$

such that $j_i < j_{i+1}, 1 \leq i < k$, and $x_i = n_{j_i}, 1 \leq i < k$.
THEOREM 1: *For any flow graph there exists a node listing of length* $\leq n^2$ *where* $n = |N|$.

Proof: Suppose $x_1, x_2, \ldots x_n$ are all the nodes of the graph, then
$l = (x_1, x_2, \ldots, x_n, x_1, x_2, \ldots, x_n, x_1, x_2, \ldots, x_n)$ with n repetitions of $(x_1, \ldots, x_n)$ is certainly such a listing.

A node listing is said to be *minimal* if there exists no shorter node listing for the same program flow graph. We will be concerned with finding minimal or near minimal listings.
Before continuing, let us examine the utility of this concept. Suppose we have a node listing for a given flow graph G; then the following algorithm computes the live sets in G.
ALGORITHM B: Live Analysis via Node Listing.
Input:

1. A flow graph $G = (N, E, n_0), |N| = n$.
2. A node listing $l = (x_1, x_2, \ldots, x_n)$ for G.
3. The sets *thru(j)* and *inside(j)* for each node $j \in N$.

Output: The sets *live(j)*for each $j \in N$.
Method:

1. Initially, let live(j) = inside(j), $1 \leq j \leq n$.
2. Perform step (c) once for each node x_j in l in *reverse* order,then halt.
3. Apply equation ($\star$) at x_j.

THEOREM 2: *Algorithm B terminates and correctly computes the "live" sets.*
Proof: Termination is trivial.
To show correctness we must show that live(j) is correctly computed for each node j.
Let j be an arbitrary node in the graph and suppose X is live on entry to j. Then there must be an X-clear path in G to a use of X. Furthermore, there must be a *basic* X-clear path to such a use since every X-clear path contains a basic X-clear path (removing nodes cannot cause a path to lose the X-clear property).
Let $(j_1, j_2, \ldots, j_k)$ be such a path where $j = j_1$. Suppose also that X is not live due to a use within j (otherwise it is marked live in step $B1$). The fundamental property of node listings assures us that $(j_1, j_2, \ldots, j_k)$ is a sequence of l, so the nodes of our basic path will be processed in the order: $j_k, j_{k-1}, \ldots, j_2, j_1$.
When node j_{k-1} is processed, X will be put in live(j_{k-1}) since it is live in j_k an X cannot be redefined in j_{k-1} (since the path is X clear). A chain of similar arguments allows us to conclude that when $j_1 = j$ is processed X will be put in live(j_1) since it was put in live(j_2) in the previous step.
Thus if any variable is live on entry to any node, that variable will be put in the live set for that node during the algorithm. If a variable is not live at arbitrary node j, then it cannot be added to live(j) by the correctness of equation($\star$) used in $B3$ [Ke3].
THEOREM 3: (Complexity of Algorithm B)
Suppose that each node in G has at most k successors for fixed k. Then the total number of bit-vector steps required by the algorithm is $|l|(k+1)$.
Proof: Step $B3$ requires $k+1$ bit-vector steps and it is executed exactly $|l|$ times where $|l|$ denotes the length of the listing l.

The speed of algorithm B depends directly on the length of a node listing for the graph to which it is applied. The question immediately arises: how short can these listing be? The next few sections deal with this question.

5. FLOW GRAPH REDUCIBILITY

Ullman [U] has defined two transformations which can be performed on a flow graph G. Transformation $T1$ is the removal of the self-looping edge (x, x) from G. Let y be a node having a single predecessor x; then transformation $T2$ is the replacement of the nodes x and y and the edges (x, y) by the single node z. The new graph contains the edge (z, z) only if G contained either (y, x) or (x, x). A flow graph is said to be *reducible* if repeated application of $T1$ and $T2$ until neither transformation is possible yields the trivial flow graph (a single node). Hecht and Ullman [HU1] have shown that the property of reducibility is independent of the order in which $T1$ and $T2$ are applied. An important result, also due to Hecht and Ullman [HU2] will be useful later.
DEFINITION: *A node x is said to dominate a node y if every path from n_0 to y contain x.*
THEOREM 4. *A flow graph is reducible if and only if for each cycle C of G there is an entry node of C which dominates all other nodes in C.*
Theorem 4 indicates that all loops in a reducible flow graph are single-entry.
It is interesting to note that reducibility is a common property of program. In fact, all "structured" programs have this property [HU2]. It is therefore reasonable to restrict our concern to node listings for reducible flow graphs.

6. LISTING GRAPHS

We can approach the node listing problem through an equivalent problem through an equivalent problem, that of constructing an acyclic graph which contains every basic path in G. Our motivation comes from the following lemma.
LEMMA 1: *Let $G = (N, E)$ be an acyclic flow graph, that is, there does not exist a sequence $(x_1, \dots, x_k)$ of nodes in G such that $x_1 = x_k$ and $(x_i, x_{i+1}) \in E, 1 \leq i < k$. Then there exists a node listing of length $|N|$ for G. Furthermore, this node listing contains every path, not merely the basic paths.*
Proof: Apply Knuth's "topological sort" [Kn] algorithm to the acyclic program flow graph. The result is a linear listing $(y_1, \dots, y_n)$ for G with one copy of each node in G.
Let $(x_1, \dots, x_k)$ be a path in G. Let m_i be the index of x_i in the listing for G, i.e., $x_i = y_{m_i}$. If (x_i, x_{i+1}) is an edge in E, then bt the order-preserving property of the topological sort $m_i < m_{i+1}$. The sequence of edges $(x_i, x_{i+1}, 1 \leq i < k$ therefore implies $m_1 < m_2 < \dots < m_k$ so the basic path $(x_1, \dots, x_k)$ is a subsequence of the generated listing. Since the path was chosen arbitrarily, all such paths are subsequences and the generated listing is a node listing.
The reader should note that this proof is constructive in that it provides an algorithm for producing a listing (the topological sort) in time proportional to the number of nodes and edges in G.
We now attempt to generalize the method of Lemma1.
DEFINITION: *A directed graph $L = (N_L, E_L)$ is said to be a listing graph for $G = (N, E, n_0)$ if*

1. *L is cycle free*

2. *There exists a function $\phi : N_L \Rightarrow N$ which is onto.*

The nodes of a listing graph may be thought of as "representing" nodes of G, that is, we say $y \in N_L$ *represents* $x \in N$ if $\phi(y) = x$. Every node in G is represented by some node in L; however, there may be several representations of any given node. The inverse of ϕ,

$$\phi^{-1} : 2^N \Rightarrow 2^{N_L}$$

defines the representations of $x \in G$; that is, $\phi^{-1}(x)$ is the set of representations of x.
DEFINITION *Consider the path $P = (x_1, \ldots x_k)$ in G. We say that P is reflected in $L = (N_L, E_L)$ if there exist nodes $y_1 \ldots , y_k$ such that*

1. $y_1 \in \phi(x_i), 1 \leq i < k$ *and*
2. $(y_i, y_{i+1}) \in E_L, 1 \leq i < k$*, i.e.,* $(y_i, \ldots , y_k)$ *is a path in L.*

A listing graph L is said to be *complete* for G if every basic path in G is reflected in L.
LEMMA 2: *Let $G = (N, E, n_0)$ be a program flow graph and let $L = (N_L, E_L)$ be a complete listing graph for G. Then there exists a node listing of length $|N_L|$ for G.*
Proof: By Lemma 1, there exists a node listing $l(L)$ of length $|N_L|$ for L. We produce a node listing $l(G)$ by replacing each node y in $l(L)$ by $\phi(y)$. Suppose $(x_1, \ldots , x_k)$ is a basic path in G; let $(y_1, \ldots , y_k)$ be its reflection in L. Then $(y_1, \ldots , y_k)$ is a subsequence of $l(L)$ by Lemma 1, so

$$(\phi(y_1), \phi(y_2), \ldots , \phi(y_k)) = (x_1, \ldots , x_k)$$

is a subsequence of $l(G)$ by construction.
Since the algorithm for producing a node listing for G from its complete listing graph G is linear in the number of nodes and edges in L we may view the problem of constructing a complete listing graph for G as equivalent to the problem of constructing a node listing for G.
The next theorem allows us to restrict our attention to strongly connected subgraphs of a flow graph.
THEOREM 5. *Let $G(N, E, n_0)$ be a program flow graph and let $(C_i = (N_i, E_i)|1 \leq i \leq m)$ be the set of maximal strongly connected components of G. If there exists a node listing l_i for each C_i, then there exists a node listing of length*

$$|N_0| + \sum_{i=1}^{m} |l_i|$$

where N_0 is the set of nodes in N which are not in any of the C_i.
Proof: Let $G' = (N', E', n_0)$ be the acyclic graph derived from G by replacing each component C_i by a single c_i. Then

$$N' = N_0 \cup (c_i, 1 \leq i \leq m)$$

and by Lemma 1, there exists a node listing l' of length

$$|N_0| + m$$

for G'. Now replace each node c_i in l' by the listing l_i. The length of the resulting listing l is

$$|N_0| + \sum_{i=1}^{m} |l_i|$$

We need to show that l is a node listing for G.
Let $P = (x_1, \ldots, x_k)$ be a basic path in G.
Case 1: All the nodes of P are in some strongly connected component C_i. Then P must be a subsequence of the listing l_i and hence a subsequence of l.
Case 2: The nodes of P are not wholly contained in any C_i. Suppose we break down the path into contiguous parts $P_j, 1 \leq j \leq p$, such that

1. each P_j is a path wholly contained in some C_i or wholly contained in the acyclic part of G, and

2. the path P is formed by concatenating the P_j end to end in order of increasing j.

Create a new path P' by concatenating the P_j end to end while transforming any P_j which is wholly contained in C_i to the single node c_i. The result is a path in G' which must be a subsequence of l' by lemma 1. All that remains is to show that if P_j is contained in C_i then it is a subsequence of l_i. This follows immediately if we show P_j to be a basic path.
LEMMA 3. *Let $P = (x_1, \ldots, x_k)$ be a basic path in flow graph G, and let $C = (N_c, E_c)$ be a subgraph of G formed by taking a subset of nodes of G and all edges among those nodes. Let $P_c = (x_j, x_{j+1}, \ldots, x_q)$ be any contiguous portion of the path P which is wholly contained in C. Then P_c is basic in C.*
Proof: Suppose not; then there exists a shorter path P_c' from x_j to x_q in C which is wholly contained in P_c. But if this is so then by replacing P_c by P_c' in the path P, we can get a shorter path P' which is wholly contained in P, a contradiction of the assumption that P is basic.
The proof of the lemma completes the argument that the listing l of length

$$|N_0| + \sum_{i=1}^{m} |l_i|$$

is a node listing for G and hence the theorem is proved.

7. SHORT NODE LISTINGS

In this section we investigate the usefulness of node listing for data flow analysis by examining the class of graphs for which node listings for which node listings are "short", that is bounded in length by $k|N|$ for some fixed k. Our hope is to show that there exists short listings for many of those graphs which require large amounts of computation by one of the "standard" algorithms [Ke1, HU1]. The restriction we shall make is actually a bit stronger than the one described above.
DEFINITION: *A flow graph $G = (N, E, n_0)$ is said to be k-listable if there exists a complete listing graph $L = (N_L, E_L)$ for G such that*

$$|\phi^{-1}(x)| \leq k, \forall x \in N.$$

With this definition, lemma 1 may be restated (in slightly weaker form) as follows:
LEMMA 1': *All acyclic flow graphs are 1-listable.*
We next turn our consideration to the class of 2-listable graphs.
LEMMA 4: *If $G = (N, E, n_0)$ is a control flow graph which contains a node x such that x is on every cycle in G, then G is 2-listable.*
Proof: We construct the listing graph $L = (N_L, E_L)$ by building two acyclic graphs L_0 and L_1 and taking their union.
Let L_0 be the acyclic graph constructed from G by removing every edge leading out of node x and renaming the nodes in the resultant graph:

$$N_{L_0} = \ (y,0)|y \in N$$
$$E_{L_0} = \ ((y_1,0),(y_2,0))|(y_1,y_2) \in N \text{ and } y_1 \neq x$$

Let L_1 be the acyclic graph constructed from G by removing the edges leading into x and renaming.

$$N_{L_1} = ((y,1)|y \in N - (x)) \cup (x,0)$$
$$E_{L_1} = (((y_1,1),(y_2,1))|(y_1,y_2) \in E \text{ and } y_1,y_2 \in N - (x)) \cup (((x,0),(y_1,1))|(x,y_1) \in E)$$

The union of these two graphs $L = L_0 \cup L_1$ is acyclic because both L_0 and L_1 are acyclic and there are no edges of the form $((y_1,1),(y_2,0))$ in L.
Define $\phi : N_L \Rightarrow N$ as follows:

$$\phi((y,0)) = \phi((y,1)) = y, \forall y \in N.$$

We must show that L is a complete listing graph for G. Let $P = (y_1, \ldots, , y_k)$ be a basic path in G. If P does not pass through x then it must be included in both L_0 and L_1. Suppose P contains x. Let P_0 be the part of p up to and including x and P_1 be the part after x. By lemma 3 P_0 and P_1 are both basic, so P_0 is reflected in L_0 while P_1 is reflected in L_1. The edge between x and its successor y in P is represented in L_1 by the edge $((x_1,0),(y_1,1))$ which was included in L by construction. Thus P is reflected in L and L is a complete listing graph for G.

There are two representations $((y,0),$ and $(y,1))$ in L of every node in G except x which has but one. Therefore G is 2-listable.

Note that we have used a subtle fact which is implicit in our definition of node listing, that is, the path (x,x), if it exists in G, is not basic.This is because self-looping blocks can be completely analyzed by local methods and are therefore considered basic blocks in this analysis.

It has been shown [Ke1,U] that the class of "seashell graphs", whose form is depicted in figure 3 below, require $O(|N|^2)$ bit vector steps to analyze using the interval method for live analysis.

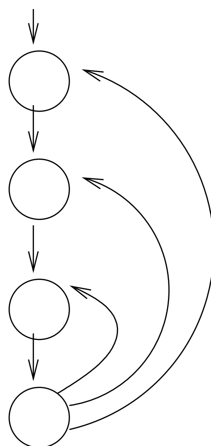


Figure 3. The seashell graph with 4 nodes.

However, the node listings for these graphs are of length $2|N|-1$ by lemma 4, so the node listing approach will be clearly superior on this class of graphs (i.e., $O(|N|)$).
We next turn our attention to a somewhat larger class of graphs which are 2-listable. We shall define this class by defining "acceptable" graph-transformations.
Let $G = (N, E, n_0)$ be a control flow graph. Furthermore, suppose there exists a function t

$$t : N \Rightarrow (0, 1)$$

which defines the "type" of a node: if $t(x) = 1$, we say that x is *expandable*, otherwise x is *non-expandable* (a basic block). By definition, $t(n_0) = 0$.
DEFINITION: *An interval transformation on $G = (N, E, n_0, t)$ is one which produces a graph $G' = (N', E', n_0, t')$ by replacing any expandable node x in G by a region $R = (N_R, E_R, h_R)$ with the following restrictions*

1. *All predecessors of x in G become predecessors of h_R in G'.*
2. *R is connected*
3. *Every cycle in R contains h_R*
4. *The region head h_R is non-expandable*

In applying an interval transformation, edges of the form (y, x) in G are replaced by edges (y, h_R) in G' while each edge (x, y) may be replaced by several edges $(x_R, y), x_R \in N_R$. The name "interval transformation" arises because the regions R are Cocke-Allen intervals [AC1].
DEFINITION: *An interval transformation is said to be single-exit if all the exits from the expanded region come from the same node, i.e., there exists a node e_R in R such that every edge of the form (x, y) in G is replaced by the edge (e_R, y) in G'.*
Figure 4 depicts a single-exit interval transformation. The non-expandable nodes are drawn as rectangles.

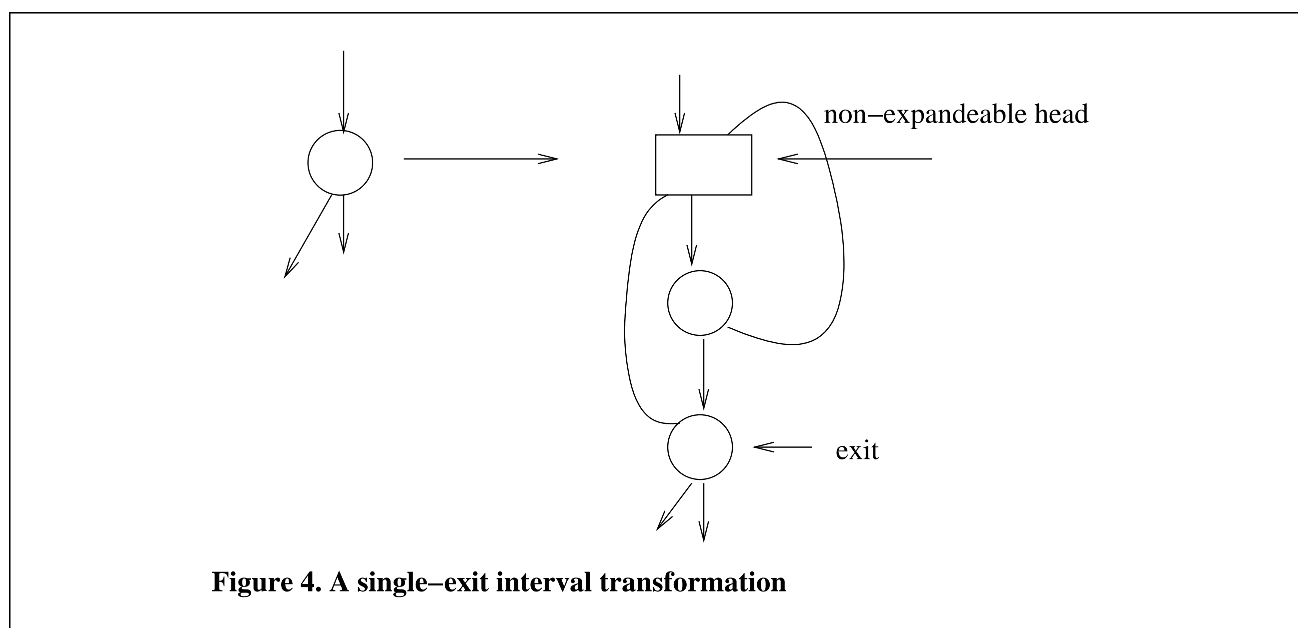


Figure 4. A single–exit interval transformation

DEFINITION: *An out-graph for a single entry strongly connected region is a subgraph of R which contains*

1. *every basic path from a node in N_R to an exit from N_R,*
2. *every basic path from a node in $N_R - (h_R)$ to a latching node (a predecessor of h_R in N_R), and*

3. *for each properly contained, single-entry, strongly-connected subregion $S = (N_S, E_S, h_S)$, every basic path from a node in $N_S - (h_S)$ to a latching node of S.*

An in-graph for R is a subgraph which contains every basic path from the head of R to another node of R.
The following theorem is an adaptation of a result by Hecht and Ullman [HU3].

THEOREM 6. *If $G = (N, E, n_0)$ is a reducible flow graph then there exists an acyclic in-graph for G which consists of all the forward branches in G.*
Proof: The *dag* of a reducible flow graph is the subgraph formed by eliminating from G those edges removed by an application of $T1$ during the reduction of G. Hecht and Ullman [HU3] have shown that the dag of a reducible flow graph is acyclic and unique. Furthermore, it contains every cycle-free path from the header of R to a node within R, because any "backward branch" from a node on such a path must by Theorem 4 branch to a node which is already on the path. Hence the dag contains every basic path to a node within R.
DEFINITION: it A 2n-acceptance transformation is a single-exit interval transformation such that the expanded regions has an acyclic out-graph.
The transformation in figure 4 above is a 2n-acceptable, while the region shown in figure 5 can never be the target of a 2n-acceptable transformation.

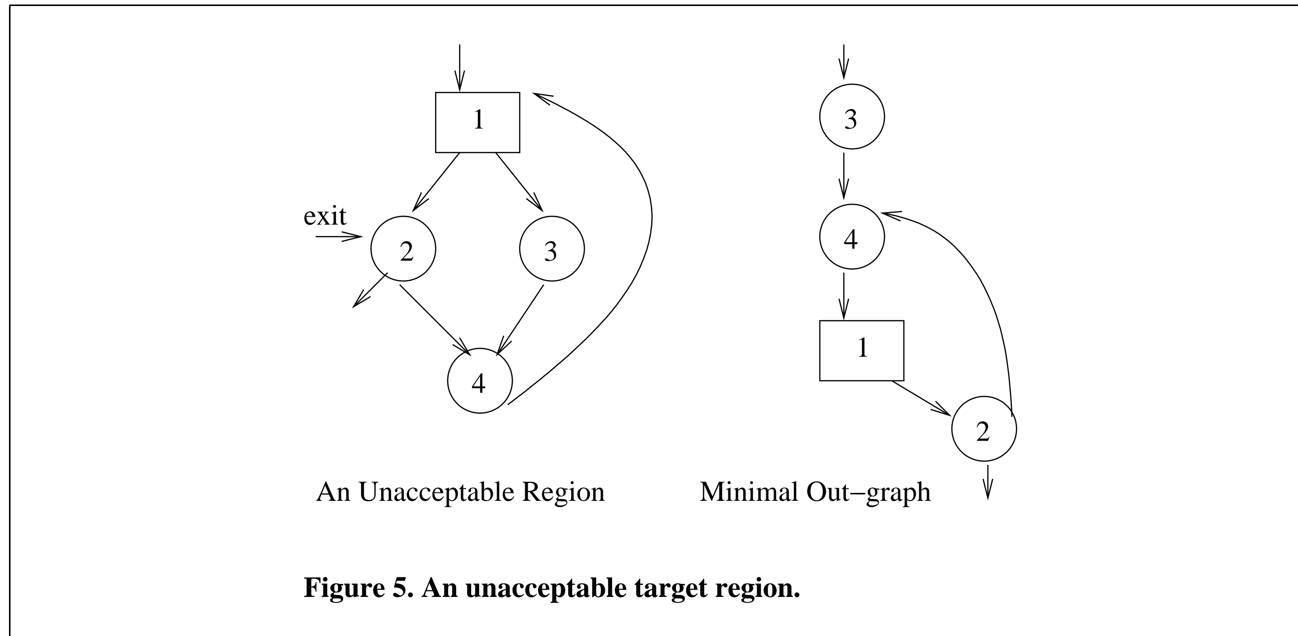


Figure 5. An unacceptable target region.

LEMMA 5. *If $G = (N, E, n_0)$ is a reducible flow graph and $G' = (N', E', n_0)$ is produced from G by an interval transformation, then G' is also reducible.*
Proof: Hecht and Ullman [HU2] have shown that every interval is reducible. Therefore, we can reduce the expanded region R to a single node by the application of a sequence of $T1$ and $T2$ transformations. The result is the original graph G which is reducible by another sequence of such transformations. If these two sequences are applied in the order described, the graph G' will be reduced to a single node.
LEMMA 6.*Let $R = (N_R, E_R, h_R)$ be a single-entry strongly connected region with an acyclic out-graph out(R). If $R' = (N_{R'}, E_{R'}, h_R)$ is formed from R by applying a 2n-acceptable transformation*

to any node in R except h_R, then R' has an acyclic out-graph.

Proof: Let out(R') be formed by inserting the out-graph for the expanding region $R_e = (N_e, E_e, h_e)$ in place of the expanded node and replacing branches into and out of the expanded node by branches to the header of R_e and from the exit of R_e respectively. Clearly out(R') is acyclic so we need only show that it is a valid out-graph for R'.

1. Let P be a basic path from a node within R' to an exit. If P contains no nodes of R_e then P must be in out(R) and therefore in out(R'). Assume that P does not contain nodes of R_e. The nodes within R_e must be contiguous in P because of the single-entry single-exit restriction on R_e; that is, no basic path to an exit of R' can pass through the exit node of R_e twice. Therefore we may replace the nodes of R_e in P by the single-expanded node from R. The resulting path P_0 is basic in R (since if it were not P would be not basic in R') and is contained in out(R'). The portion of the path within R_e is basic in R_e by lemma 3 and hence is contained in out(R_e). Therefore the whole path is contained in out(R') by construction.

2. Let P be a basic path in a subregion $S = (N_S, E_S, h_S)$ of R' from a node other than the head h_S to a latching node for S.

   (a) Suppose S does not contain the expanded node. Then P is in out(R) and therefore in out(R').

   (b) Suppose S is the expanded region R_e. Then P is in out(R_e) and hence in out(R').

   (c) Suppose S contains the expanded node. On any path through R_e to a latching node of S the nodes of R_e must be contiguous by the same argument as in case a above. Therefore the path P_0 in which the nodes from R_e are replaced by the single expanded node is in out(R) and hence P is in out(R') by lemma 3 and the construction.

Note that case $b3$ includes the special case $R' = S$, so P may be any basic path to a latching node for R'.

THEOREM 7: *Let $G = (N, E, n_0)$ be a control flow graph formed by beginning with a single expandable node and repeatedly applying 2n-acceptable transformations to produce a sequence of graphs $(G_0, G_1, \ldots, G_m, G)$ which ends in G. Then G is 2-listable.*
Proof: Let L_0 be the acyclic out-graph for G (guaranteed by lemma 6) in which each node y is renamed $(y, 0)$ as in the proof of lemma 4. Since G is reducible by theorem 6 there exists an acyclic in-graph for G as well. Let L_1 be this graph with each node y rename $(y, 1)$. Form $L = L_0 \cup L_1$. For each node $(x, 0)$ in L_0 which is a latching node for a header h_x (of some subregion of G) insert the branch $((x, 0), (h_x, 0))$ in N_L. The resulting graph is acyclic because both L_0 and L_1 are acyclic and branches between them go only one way. We need only show that L is a complete listing graph for G. Suppose P is a basic path in G. There are two case to consider

1. The path P contains no branch from a latching node to its region head. In this case P consists entirely of forward branches [HU3] and must be contained in L_1.

2. The path P contains at least one branch from a latching node to its region head. Let (y, h_0) be the last such branch in P. Hecht and Ullman [HU3] have shown that the portion P_0 of P up to and including y must be entirely contained in the region headed by h_0 (otherwise the path P would be a cycle through h_0). Hence, P_0 is contained in the out-graph of R_0 (by lemma 6)

and in the out-graph of G (also by lemma 6). The portion P_1 of P beginning with h_0 consists entirely of forward branches and is therefore reflected in L_1. finally, the edge $((y, 0), (h_0, 1))$ is in L by construction so the entire path P is reflected in L.

But there are at most two representations of each node in G so

$$|\phi(x)| \leq 2$$

and the theorem is proved.

Consider the family of "spiral graphs" defined by transformations of the form shown in figure 6. It can be easily seen that both generating transformations are 2n-acceptable. Therefore the spiral

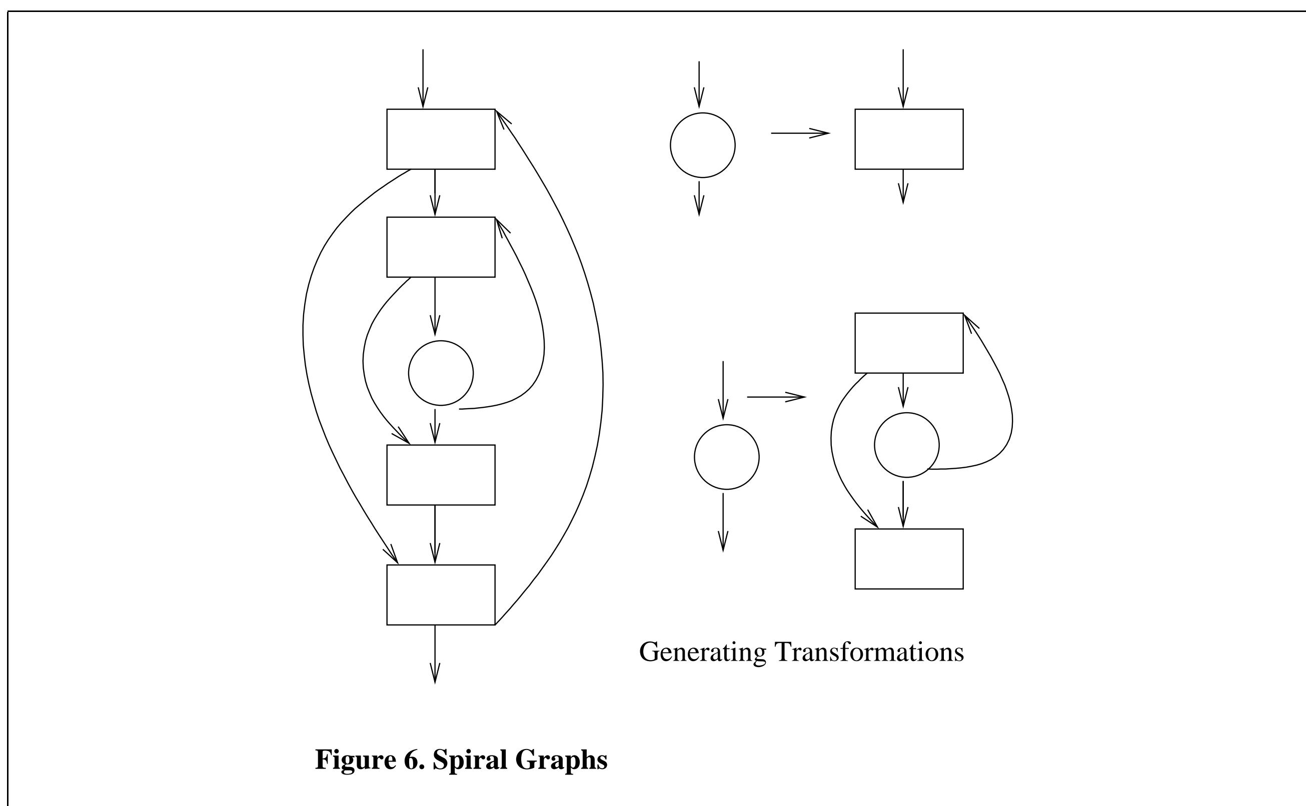


Figure 6. Spiral Graphs

graphs are 2-listable. This family is important because it has been shown that live analysis on spiral graphs requires $O(|N^2|)$ bit-vector steps when either of the "standard" algorithms is applied [Ke3]. A more interesting case is presented by the transformations of Bohm and Jacopini [BJ] (figure 7) which are often taken as standard for "structured" programming [DDH]. COROLLARY: *Any flow graph formed from a single node by repeated application of the transformations of Bohm and Jacopini is 2-listable.*

Proof: Clearly transformations 2 through 5 are 2n-acceptable. Only transformation 1 is in question because it does not have a single block as its head. However, transformation 1 can be eliminated if we include the four 2n-acceptable transformations shown in figure 8.

The result now follows immediately from theorem 7.

Although it is not our intention to exhaustively investigate the 3-listable graphs, the following theorem should give some indication of the addition power obtained by allowing 3 copies of any node.

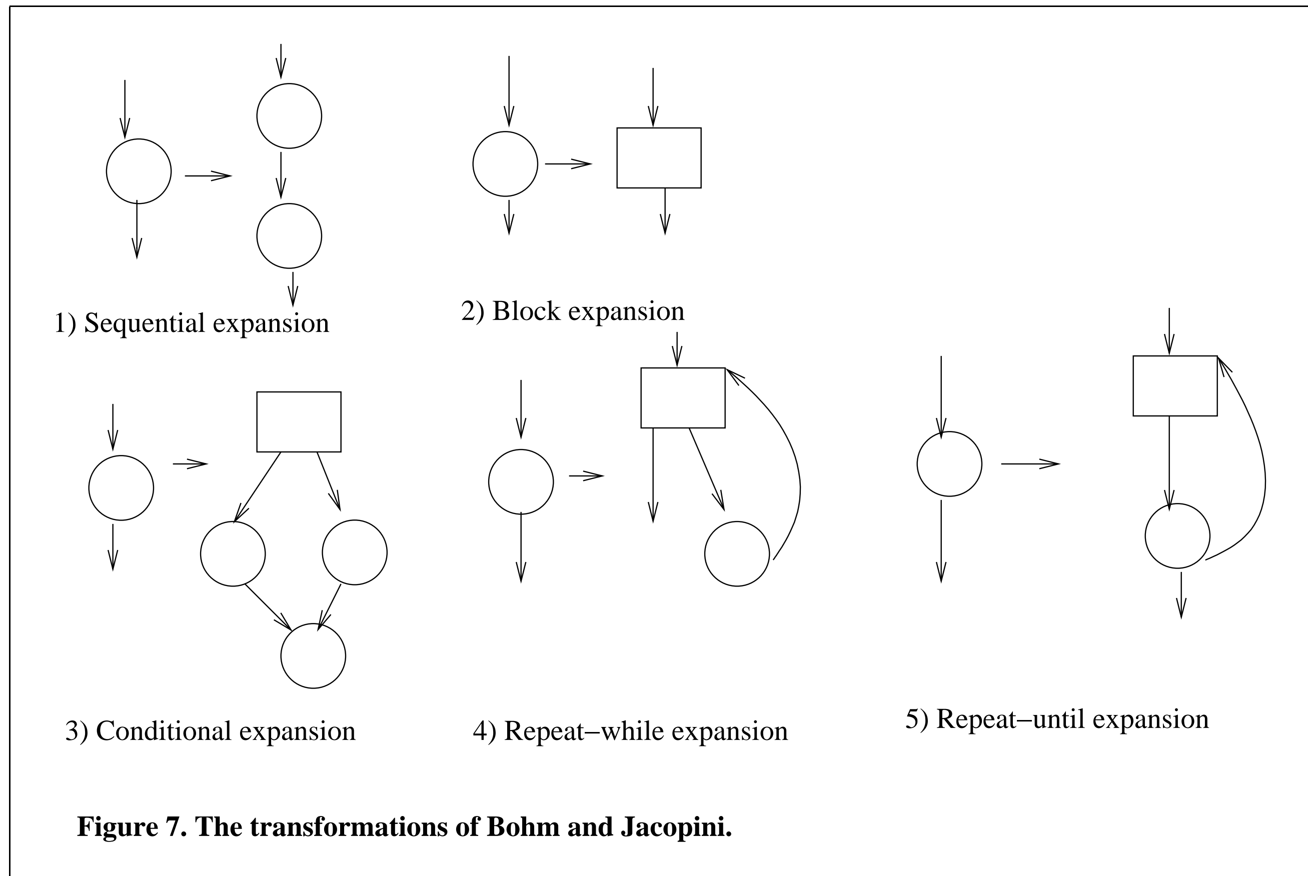


Figure 7. The transformations of Bohm and Jacopini.

THEOREM 8: *Let $G = (N, E, n_0)$ be a control flow graph generated from a single node by repeated application of single-exit interval transformations in which the out-graph of the expanded region can be made acyclic by duplicating the head of that region once. Then G is 3-listable.*
Proof: The same construction as in the proof of theorem 7 will work if modified so that the branches into the out-graph of the expanded region go to both copies of the head. The proof is then straightforward and follows the proof of theorem 7. The number of representations of any node is bounded by 3 since we only make a third copy of a non-expandable node.

Theorem 8 allows us to consider the "double-exit loop" or "loop conditional"(figure 9) in which a special condition inside the loop causes a branch to a piece of code for exceptional processing before returning to the main execution stream.

The need for such generalized loop exits in goto-less programming has been discussed in the literature [Kn2,Z]. It is therefore gratifying that the addition of such a control structure will not increase the complexity of data flow analysis unreasonably.

8. SUMMARY AND CONCLUSIONS

We have defined the concept of a "node listing" and examined its usefulness in the solution of global data-flow analysis problems. The class of flow graphs with "short" listings is large and include most of those flow graphs which would be generated by structured programming and many flow graphs on which the standard data-flow algorithms do quite poorly.
Recently, Aho and Ullman [AU2] have devised an algorithm which, given a reducible flow graph

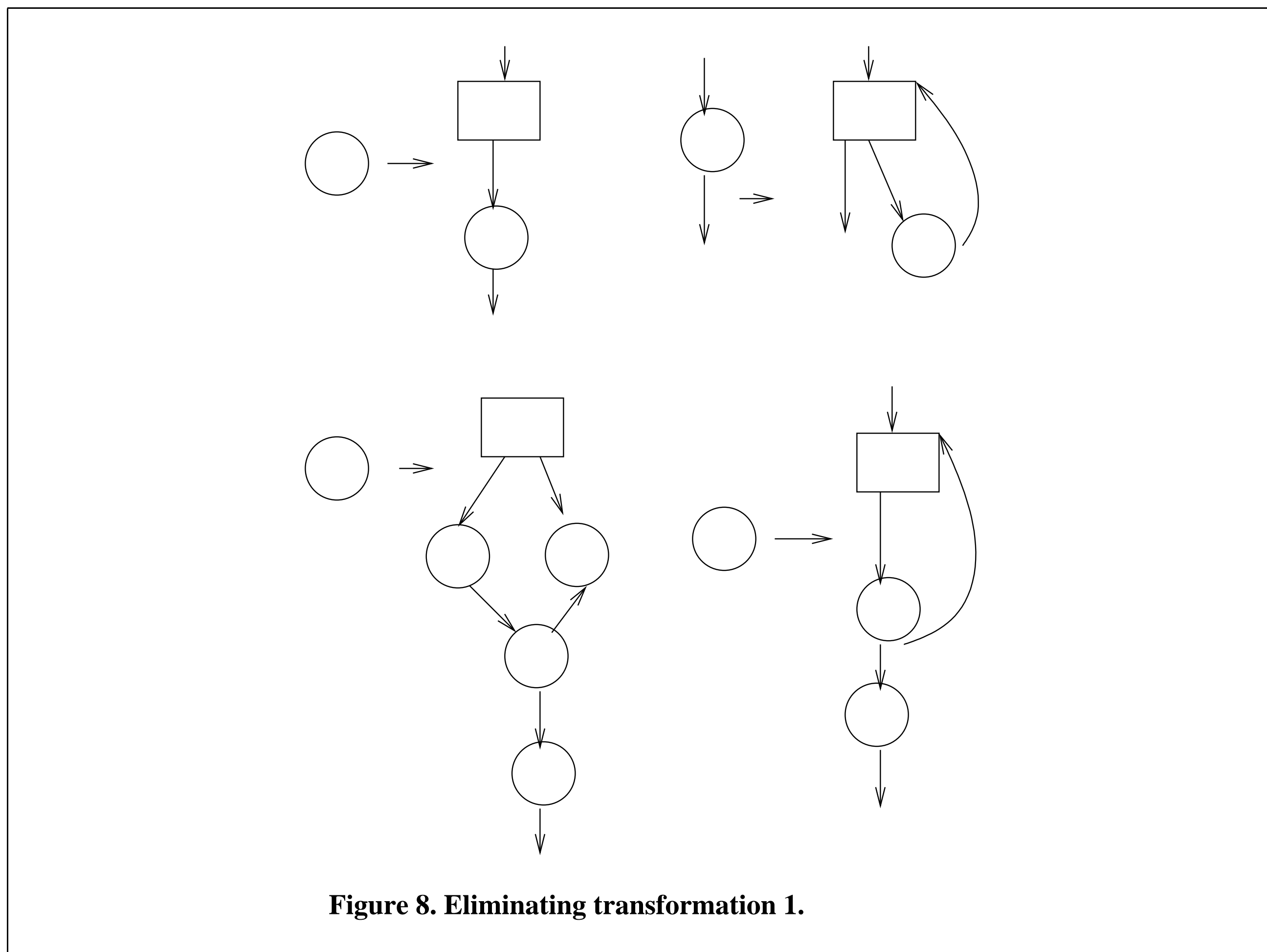


Figure 8. Eliminating transformation 1.

$G = (N, E, n_0)$, will construct a node listing for G of length proportional to $|N|\log|N|$ in time proportional to $|N|\log|N|$. Thus the previously open question of whether or not there exists an $O(|N|\log|N|)$ algorithm for live analysis has been resolved via the node listing approach.

The node listing method also may have implications in another important area of computer science. Recently, much attention has been centered around the question "What makes a program well-structured ?" Many authors have attempted to define which control structures should be avoided if a program is to be easily read and understood by another person, but no real quantitative measures have been put on this quality of "understandability". However, the length of a program's node listing is in one sense a measure of its understandability to the compiler. We have seen here that the compiler and the human agree that certain control structures such as those of Bohm and Jacopini are simple. It does not, therefore, seem unreasonable to view the complexity of a control structure in terms of the node listing expansion that it may cause.

ACKNOWLEDGEMENTS.

The author wishes to thank Fran Allen, Meera Blattner, Ashok Chandra, John Cocke, Rodney Farrow, Amelia Fong, George Markowsky and Barry Rosen for their help and encouragement.

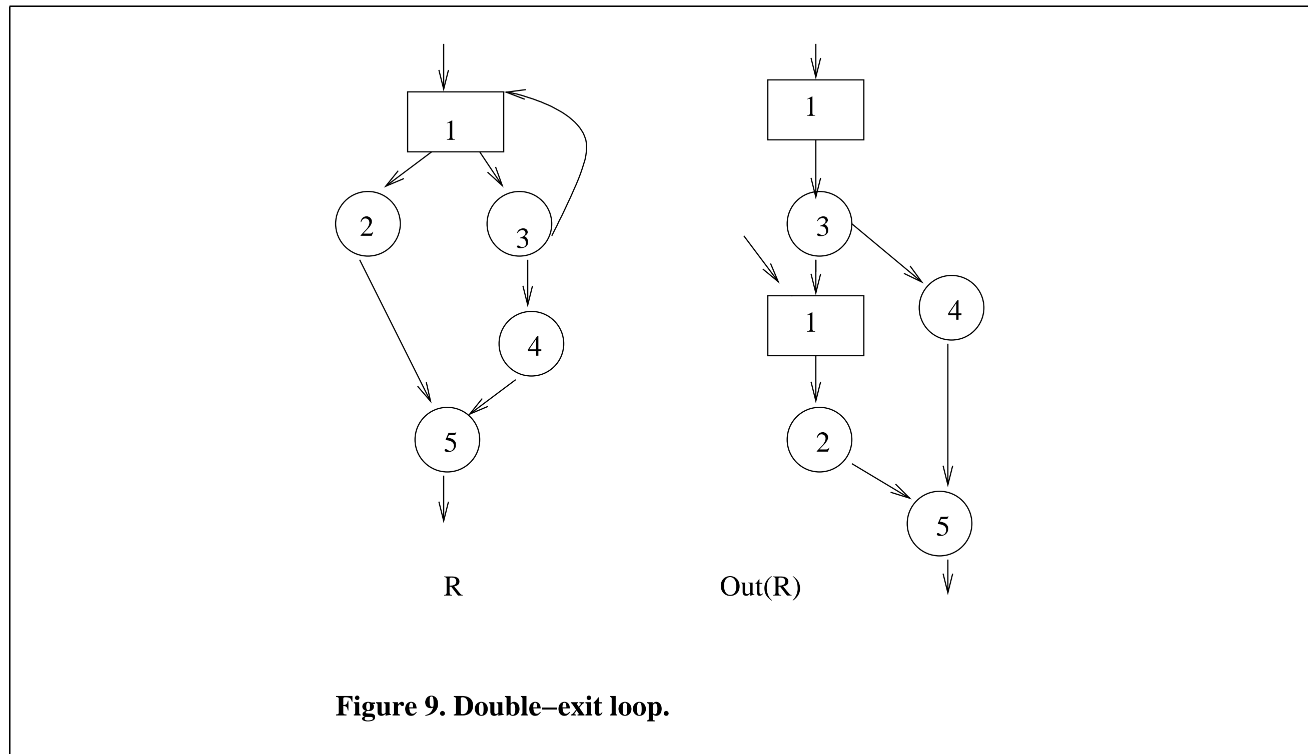


Figure 9. Double–exit loop.

REFERENCES

AU1 Aho,A.V. and Ullman,J.D.,*The Theory of Parsing,Translation and Compiling* Vol. 2, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

AU2 Aho,A.V. and Ullman,J.D., Private Communication.

A1 Allen,F.E., Control Flow Analysis, *SIGPLAN Notices* Vol. 5,No. 7, pp. 1-19, July 1970.

A2 Allen,F.E., “A Method for Determining Program Data Relationships”, *Lecture Notes in Computer Science*, Vol. 5, *International Symposium on Theoretical Programming*, Springer-Verlag, Berlin,1974.

AC Allen,F.E. and Cocke,J. “Graph Theoretic Constructs for Program Control Flow Analysis”, IBM Research Report RC4633, T.J. Watson Research Center, Yorktown Heights, New York, Nov. 1973.

BJ Bohm,C. and Jacopini,G., “Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules,” *CACM*, Vol. 19, No. 5, May 1966.

DDH Dahl,O.J., Dijkstra,E.W. and Hoare,C.A.R., *Structured Programming*, Academic Press, New York, 1972.

HU1 Hecht,M.S. and Ullman,J.D., “Analysis of a Simple Algorithm for Global Flow Problems”, *Conf. Record of ACM Symposium on Principles of Programming Languages,* Boston, Mass., pp. 207-217, October 1973.

HU2 Hecht,M.S. and Ullman,J.D., “Flow Graph Reducibility,” *SIAM J. Computing*, Vol. 1, No. 2, pp. 188-202, June 1972.

HU3 Hecht,M.S. and Ullman,J.D., “Characterizations of Reducible Flow Graphs”, *JACM*, Vol. 21, No. 3, pp. 367-375, July 1974.

Ke1 Kennedy,K., “Safety of Code Motion”, *International J. Computer Math,* Vol. 3, pp. 5-15, Dec. 1971.

Ke2 Kennedy,K., “A Comparison of Algorithms for Global Flow Analysis”, Technical Report 476-093-1 Dept. of Mathematical Sciences, Rice Univ., Houston, Texas, February 1974.

Ki Killdall,G.A., “A Unified Approach to Global Program Optimization”, *Conf. Record of ACM Symposium on Principles of Programming Languages*, Boston,Mass., pp.194-206, October 1973.

Kn1 Knuth,D.E., *The Art of Computer Programming*, Vol. 1, *Fundamental Algorithms*, Addison-Wesley, Reading, Mass., 1968.

Kn2 Knuth,D.E., “Structured Programming with goto Statements”, draft, Jan 1974.

Sc Schaefer,M. , *A Mathematical Theory of Global Program Optimization”*, Prentice-Hall, Englewood Cliffs, N.J., 1973.

U Ullman,J.D., “Fast Algorithms for the Elimination of Common Subexpressions”, *Acta Informatica*, Vol. 2,No. 3, pp. 191-213, 1973.

Z Zahn,C.T., “A Control Statement for Natural Top-down Structural Programming”, *Proc. Symposium on Programming Languages*, Paris, 1974.

Node Listings for Reducible Flow Graphs

A. V. Aho
Bell Laboratories, Murray Hill, New Jersey

AND

J. D. Ullman
Princeton University, New Jersey 08540

K. Kennedy recently conjectured that for every n node reducible flow graph, there is a sequence of nodes (with repetitions) of length $O(n \log n)$ such that all acyclic paths are subsequences thereof. Such a sequence would, if it could be found easily, enable one to do various kinds of global data flow analyses quickly. We show that for all reducible flow graphs such a sequence does exist, even if the number of edges is much larger than n. If the number of edges is $O(n)$, the node listing can be found in $O(n \log n)$ time.

1. Introduction

Much of the research in global data flow analysis has centered around a class of reducible flow graphs, first defined by Allen [2] and shown empirically by Knuth [3] to include virtually all the flow graphs arising from naturally occurring FORTRAN programs. One general approach to solving global data flow problems is the technique of iteratively converging on the maximum fixed point of a set of equations. Hecht and Ullman [4] showed that for the usual equations on bit vectors, e.g., [5–8], convergence could be obtained when one had visited the nodes in such a way that every cycle-free path was a subsequence of the nodes actually visited.

An ordering of nodes based on depth-first search was used in [4] to show that convergence will be very rapid on reducible flow graphs, assuming the evidence of [3] that programs are not only reducible but of small loop nesting depth. The same technique of propagating data along acyclic paths can be applied to Kildall's [9] lattice-theoretic generalization of the bit vector data flow algorithms, at least in a restricted subcase [10].

Kennedy [1] suggested that for those data flow problems for which propagation along acyclic paths suffices, a solution could be expedited by finding for each flow graph, a (*strong*) *node listing*, an ordering of the nodes of a flow graph which includes every acyclic path as a subsequence. For example, the flow graph of Fig. 1 has a node listing $abcba$, while the method of [4] would require visiting nodes a, b, and c in that order, four times.

Kennedy [1] also mentioned the notion of a *weak node listing*, an ordering of the nodes such that every acyclic path P is either a subsequence of the listing or there is another acyclic path which is a proper subsequence of P and has the same source and destination. Clearly every strong node listing is a weak node listing, so a construction that yields short strong node listings also yields short weak node listings. We consider weak node listings briefly when we mention lower bounds.

In [1] Kennedy showed that for a subclass of the reducible graphs, essentially those produced from constructed from **begin** $\cdots$ **end**, **while** $\cdots$ **do**, and **if** $\cdots$ **then** $\cdots$ **else** statements, a node listing

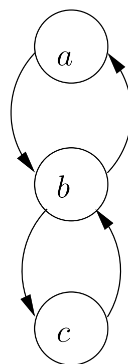


FIG. 1. Example Flow Graph

of length $2n$ exists for any such n node graph [2]. Thus, for these graphs, which reflect structured programs that do not use **break** statements, there exists a linear algorithm to do the kinds of data flow analyses covered by [4, 10][3]. Kennedy also posed the question of how long a node listing is necessary for an arbitrary reducible flow graph, and conjectured $O(n \log n)$ is sufficient [4]. In this paper we show that this conjecture is correct even if the number of edges is much greater that n. If the number of edges is $O(n)$, then the node listing can be found in $O(n \log n)$ time.

2. BASIC DEFINITIONS

A (*directed*) *graph* is a pair (N, E) where N is a set of *nodes* and $E \subseteq N \times N$ is a set of *edges*. If (n_1, n_2) is in E, we write $n_1 \to n_2$, and say n_1 is *predecessor* of n_2 and n_2 a *successor* of n_1. We say n_1 is the *tail* and n_2 the *head* of the edge $n_1 \to n_2$. A *path* in G is a sequence of nodes $n_1, n_2, \dots, n_k, k \geq 1$, such that $n_i \to n_{i+1}$ for all $i, 1 \leq i < k$. If $n_i = n_j$ for some $i \neq j$, the path has a *cycle*. Otherwise it is *acyclic*.

A *flow graph* is a triple $G = (N, E, n_0)$, where (N, E) is a graph, n_0 in N is the *initial node*, and there is a path from n_0 to each node in N. In a flow graph, node n_1 *dominates* node n_2 if every path from the initial node to n_2 passes through n_1. For example, in Fig. 1, assuming a is the initial node, we have a dominating all nodes, b dominating both itself and c, and c dominating only itself.

Reducible flow graphs were defined originally in [2] in terms of "intervals." The characterization we shall find most useful here is that of [14], in terms of the following transformations T_1 and T_2 on flow graphs.

T_1: Remove a *loop*, i.e., an edge $n \to n$ for some node n.

T_2: Suppose n has a unique predecessor m, and n is not the initial node. Then replace m and n by a new node, say p. For $q \neq m, n$, there is an edge $q \to p$ if there was previously an edge $q \to m$. For $r \neq m, n$, there is an edge $p \to r$ if there was previously an edge $m \to r$ or $n \to r$ or both. There is an edge $p \to p$ if previously there was an edge $m \to m$ or $n \to m$ or both. Under this transformation, we say that m *consumes* n.

[2]This same class of graphs was considered by Graham and Wegman independently, and another completely different but equally efficient approach was discovered. Geschke also demonstrated that certain data flow problems could be solved easily for this class, and his algorithm can be shown linear

[3]By "linear," we mean of linear time complexity in the number of nodes of the flow graph. Bit vector operations, or lattice meet and function applications in the more general framework of, are deemed to take one "time unit."

[4]For arbitrary graphs, the problem is equivalent to finding a sequence of digits $1, 2, \dots, n$ such that every permutation of $1, 2, \dots, n$ is a subsequence. Newey shows a sequence of length proportional to n^2 is necessary and sufficient here.

A flow graph is *reducible* if it can be transformed into a single node by repeated applications of T_1 and T_2.

EXAMPLE 1. The flow graph of Fig. 1 is reduced by the sequence shown in Fig. 2. □

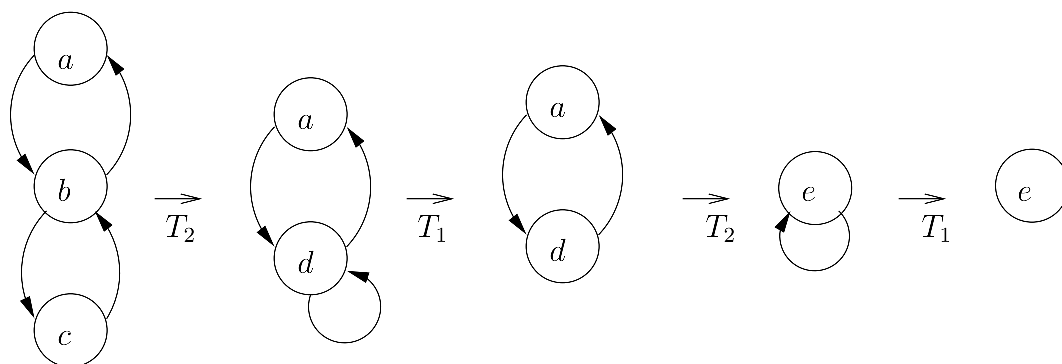


FIG. 2. Reduction of a reducible flow graph.

As a flow graph is reduced, each node in every derived graph *represents* a set of nodes and edges of the original graph, and each edge *represents* a set of edges of the original. Initially, each node and edge represents itself. If we apply T_1 to eliminate edge $n \to n$, then afterwards n represents what it and the edge $n \to n$ previously represented. If T_2 is applied, with m consuming n to form p, then p represents what m, n and the edge $m \to n$ previously represented. An edge $q \to p$ represents what $q \to m$ represented, and edge $p \to r$ represents what $m \to r$ and $n \to r$ represented. Edge $p \to p$ represents what $m \to m$ and $n \to m$ represented.

EXAMPLE 2. After the first step of Fig. 2, node d represents b, c and the edge $b \to c$. Edge $d \to d$ represents $c \to b$. At the penultimate step, e represents a, b, c and the edges $b \to c$, $c \to b$ and $a \to b$. At the end, e represents all nodes and edges, of course. □

A *region with header* h of a flow graph $G = (N, E, n_0)$ is a set of nodes N' and edges $E' \subseteq N' \times N'$, such that if $m \to n$ is in E, then

(i) if n is in N' and $n \neq h$, then m is in N', and

(ii) if m and n and in N' and $n \neq h$, then $m \to n$ is in E'.

That is, the only way a region can be entered from outside is through the header , and E' includes all those edges in $N' \times N'$, with the possible exception of some which enter the header from inside the region.

EXAMPLE 3. In the flow graph of Fig. 1, $N' = \{b, c\}$ is a region if E' is either $b \to c$ alone or $\{b \to c, c \to b\}$. □

3. BASIC RESULTS

The following characterization of reducible flow graphs is taken from [15].

LEMMA 1. *All and only reducible flow graphs can have their edges partitioned uniquely into two sets the forward edges, and back edges , having the following properties.*

(1) The flow graph with the back edges deleted forms a flow graph with no cycles, and if any back edge is added, a cycle results.

(2) For each back edge $n \to m$, m dominates n.

Using Lemma 1, it is possible to show the following result, taken from [4].

LEMMA 2. *Let* $P = n_1, n_2, \ldots, n_k$ *be an acyclic path in a reducible flow graph, and let* $n_{i_1-1} \rightarrow n_{i_1}, n_{i_2-1} \rightarrow n_{i_2}, \ldots, n_{i_r-1} \rightarrow n_{i_r}$ *be the sequence of back edges along* P*, in that order. Then* n_{i_j} *dominates* $n_{i_{j-1}}$ *for all* $j, 1 < j \leq k$. □

Since dominance is easily seen to be transitive (see [6], for example), Lemma 2 implies that each head of a back edge along an acyclic path dominates all the previous heads of back edges.

We now prove a lemma which will be central to the development of our theorem.

LEMMA 3. *Let* $R = (N', E')$ *be a region with header* h *of flow graph* $G = (N, E, n_0)$*. Let* P *be an acyclic path in* G *which begins at some node outside* R*. Then* P *traverses no back edge of* R*.*

Proof. Suppose P has in sequence two distinct nodes n, m of R, where $n \rightarrow m$ is a back edge. By definition of region, P reaches h before n, or h is n. Since G is a flow graph, there is a path Q from n_0 to h, and we can assume without loss of generality that Q is acyclic. Thus, no node of R except h appears on Q by definition of region again. Then Q followed by a portion of P from h to n forms a path from n_0 to n that avoids m, unless $m = h$. In the first case, we contradict Lemma 1 which says that m dominates n. In the second case m appears twice on P, since $h = m = n$ is ruled out by assumption. □

It follows from Lemma 1 that there is an ordering of the nodes of a reducible flow graph such that any path which uses no back edges is a subsequence thereof. In particular, any topological sort of the flow graph with back edges removed suffices. Let us call such an ordering *acyclic*. It follows from Lemma 3 that if a path enters a region R through its header, it mush follow a subsequence of an acyclic ordering of R until it leaves R.

Our final preliminary result concerns *parses* of reducible flow graphs. As we reduce a reducible flow graph by T_1 and T_2, the nodes represent regions at all times. Reduction by T_1 does not increase the number of nodes in the region represented by the node to which T_1 is applied, although it does add some edges. Thus, only T_2 builds regions with progressively larger number of nodes. We may therefore state the following lemma, whose proof is found in [16].

LEMMA 4. *Let* $R = (N', E')$ *be a region of some flow graph* G *represented by some node during the reduction of* G*, with* N' *not a singleton. Then* N' *can be partitioned into two nonempty disjoint sets of nodes* N_1 *and* N_2*, such that* (N_1, E_1) *and* (N_2, E_2) *are regions, where* $E_1 = E' \cap N_1 \times N_1$ *and* $E_2 = E' \cap N_2 \times N_2$. □

Note that there may be edges in E' that are in neither E_1 nor E_2. These edges have for heads the header of (N_1, E_1) or (N_2, E_2) and have tails in the other regions.

4. SPIRAL GRAPHS

We now introduce a special kind of reducible flow graph, called spiral graph, for which we give two methods of constructing a node listing. The next section shows that any given flow graph can be reduced to a spiral graph; the node listing of the spiral graph can be used to help construct a node listing for the given graph.

The class of *spiral* flow graphs is defined recursively as follows.

1. A node with no edge is a spiral flow graph.

2. If $G = (N, E, n_0)$ is a spiral graph and n is a new node, then

(a) $(N \cup \{n\}, E \cup E' \cup \{n \rightarrow n_0\}, n_0)$ is a spiral graph, where E' is the set of edges from nodes in N to n, and

(b) $(N \cup \{n\}, E \cup E' \cup \{n \rightarrow n_0\}, n)$ is a spiral graph, where E' is an in (a).

3. Nothing else is a spiral graph.

These two constructions are illustrated in Fig. 3. The primary distinction between these two constructions is whether n or n_0 is the header of the constructed graph.

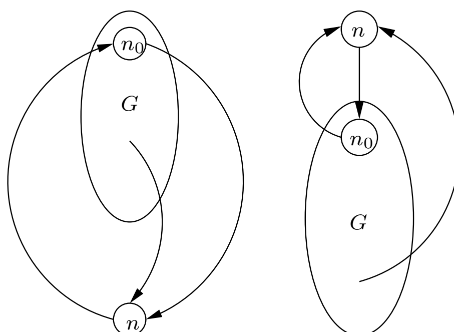


FIG. 3. Construction of spiral graphs.

Note that each spiral graph has a specific order in which the nodes were added during its construction. Except for edges $n \rightarrow n_0$ added in case 2, all edges "spiral outwards," that is their heads were added after their tails and therefore the heads are further from the center.

If n is added to a spiral graph by rule (2a), call it a *trailing* node, and if added by rules (1) or (2b), call it a *leading* node. Note that the initial node of a spiral graph is always a leading node. The following lemma summarizes some results that are easily proved by induction on the number of nodes in a spiral graph.

LEMMA 5.*(i) Every spiral graph is reducible; all spiral graphs from which it is formed are regions.*
(ii) Edges added using rule (2a) except $n \rightarrow n_0$, and the edge $n \rightarrow n_0$ added by rule (2b) are forward; the other edges are back edges.
(iii) Each leading node dominates all previously added nodes.

Proof. (i) By the inductive hypothesis, G of Fig. 3(a) or (b) is reducible. Reducing it to a single node results in a pair of nodes with edges between them. This resulting graph is clearly reducible.
(ii) An easy induction shows that the edges designated as forward form an acyclic graph, and that no other edge can be added without forming a cycle . By Lemma 1 and part (i), this selection of forward and backward edges is unique.
(iii) By part (ii), back edges enter each leading node from all previously added nodes. The result then follows from Lemma 1.□

We now need a recursive method of constructing node listings for spiral graphs.

LEMMA 6. *Let G be a spiral graph formed from nodes $n_1, n_2, \ldots, n_k$ added in that order. Let G' be the spiral graph consisting of nodes $n_1, n_2, \ldots, n_{j-1}$ and all edges between them in G. Let G'' be the (spiral) graph formed from $n_j, n_{j+1}, \ldots, n_k$ and all edges between them. Let A and B be node listings for G' and G'', respectively, and let $\hat{A}$ and $\hat{B}$ respectively denote the nodes of G' and G'' in an acyclic order. Then $A\hat{B}\hat{B}\hat{A}B$ is a node listing for G.*

Proof. Suppose an acyclic path P begins in G', and then enters G''. If P follows more than one back edge upon or after leaving G', it can never return to G'. In proof, Lemma 5(2) tells us that every back edge enters a leading node. By Lemma 2, the second back edge enters a node which dominates the head h of the previous back edge. By Lemma 5(3) applied to h, the header of G' cannot be reached without passing through h again. Thus, if P leaves G' and returns, it does so after following exactly one back edge. The portion of P until the return to G' is thus a subsequence of $A\hat{B}\hat{B}$. Once in G' for the second time, P may not follow any back edge by Lemma 3. Thus P is a subsequence of $A\hat{B}\hat{B}\hat{A}$ until it again leaves G'. In that event, it cannot re-enter G' at all, since to do so it would have to pass through two leading nodes which dominate the header of G', which we just argued as impossible. Thus P is a subsequence of $A\hat{B}\hat{B}\hat{A}B$.

The case in which P begins in G'' rather than G', or where P begins in G' but never returns are easier to handle than the case above, so we omit further details. □

We extend Lemma 6 to apply to the partition of a spiral graph into three parts, the middle part being a single node.

LEMMA 7. *Let G be as in Lemma 6 with G' formed as before from $n_1, n_2, \ldots, n_{j-1}$ and G'' formed from $n_{j+1}, n_{j+2}, \ldots, n_k$. Let A and B be node listings for G' and G''; let $\hat{A}$ and $\hat{B}$ be acyclic orderings of these graphs. Then $A\hat{B}\hat{B}\hat{A}n_j\hat{B}\hat{B}\hat{A}B$ is a node listing for G.*

Proof. Straightforward generalization of Lemma 6. □

We now need to show not only that each spiral graph has a short node listing, but also that given an arbitrary weighting on the nodes, there is a node listing in which the node of heaviest weight appear the fewest times. The motivation for considering weights is that arbitrary reducible graph will be reduced to subgraphs of spiral graphs. In so doing, the nodes of the spiral graph will represent regions of varying sizes and the weight of a node in the spiral graph will reflect the size of the region it represents.

In what follows we need certain constants which we assign as follows: $a = 2/\log\frac{3}{2}$ [5] and $b = (\frac{3}{2})^{1/2}$. Note that $a \log b = 1$.

LEMMA 8. *Let G be a spiral graph formed from nodes $n_1, n_2, \ldots, n_k$ in that order. Let n_i have weight w_i, and let $W = \sum_{i=1}^{k} w_i$. Then G has a node listing in which n_i appears at most $a \log(bW/w_i)$ times, for $1 \le i \le k$.*

Proof. If $k = 1$, the result is trivial. Suppose $k > 1$ and assume the lemma holds for spiral graphs of fewer than k nodes. It is easy to see that one of the following two cases must occur.

(1) For some j, $1 < j \le k$, we have $\frac{1}{3}W \le \sum_{i=1}^{j-1} w_i \le \frac{2}{3}W$, or
(2) For some j, we find $w_i > \frac{1}{3}W$, $\sum_{i=1}^{j-1} w_i < \frac{1}{3}W$ and $\sum_{i=j+1}^{k} w_i < \frac{1}{3}W$.

In each of these cases we partition G and apply one of Lemmas 6 or 7.

Case 1. $\frac{1}{3}W \le \sum_{i=1}^{j-1} w_i \le \frac{2}{3}W$. We partition G as in Lemma 6, letting G' consist of nodes $n_1, n_2, \ldots, n_{j-1}$ and G'' be the remaining nodes. Let $W' = \sum_{i=1}^{j-1} w_i$ and $W'' = \sum_{i=j}^{k} w_i$. By the inductive hypothesis applied to G' and G'',there is a node listing A for G' in which n_i appears at most $a \log(bW'/w_i)$ times for $1 \le i < j$. Also, there is a node listing for G'' in which n_i appears at most $a \log(bW''/w_i)$ times for $j \le i \le k$. By Lemma 6 there is a node listing for G in which for $j \le i \le k$, n_i appears at most $2 + a\log(bW''/w_i) = alog(2^{2/a}bW''/w_i)$ times. Since $W'' \le \frac{2}{3}W$, and $\frac{2}{3}2^{2/a} = 1$, n_i appears at most $a \log(bW/w_i)$ times. A similar argument prevails in the case

[5] All logarithms are to the base 2

$1 \le i < j$.

Case 2. $w_j > \frac{1}{3}W$, $\sum_{i=1}^{j-1} w_i < \frac{1}{3}W$ and $\sum_{i=j+1}^{k} w_i < \frac{1}{3}W$. Partition G into G', n_j and G'' as in Lemma 7, letting G' consist of $n_1, n_2, \ldots, n_{j-1}$ and G'' consist of $n_{j+1}, n_{j+2}, \ldots, n_k$. By the inductive hypothesis and the fact that $\sum_{i=1}^{j-1} w_j$ and $\sum_{i=j+1}^{k} w_i$ are both less than $W/3$, there are node listings A and B for G' and G'', such that n_i appears at most $a \log(bW/3w_i)$ times for $i \neq j$. By Lemma 7, there is a node listing for G in which n_i appears at most $4 + a\log(bW/3w_i) = a\log(2^{4/a}bW/3w_i)$ times. Since $\frac{1}{3}2^{4/a} = \frac{3}{4}$, we have our result in this case. Finally, there in one occurrence of n_j in the node listing for G. Since $a\log(bW/w_j) \ge a \log b = 1$, the proof is complete. □

5. The Main Result

We shall now show how to construct an $O(n \log n)$ length node listing for any reducible flow graph. Basically the method is to partition each reducible flow graph into pieces, none of which has more than two-thirds the whole. The pieces are themselves regions, and node listings for them can be found recursively. Then we form a subgraph of a spiral graph by reducing each of these regions R to a single node n_R. The desired node listing is found by taking a node listing for the spiral graph, substituting an acyclic ordering for each region R represented by node n_R, and preceding the result by a node listing for each region in the partition.

LEMMA 9. *Let $G = (N, E, n_0)$ be a reducible flow graph with $k > 1$ nodes. Then we can find a set of disjoint regions $R_1, R_2, \ldots, R_m$, whose union includes all nodes of G, having the following properties:*

1. *none of $R_1, R_2, \ldots, R_m$ has more that $\frac{2}{3}k$ nodes ;*
2. *there is a sequence of regions $S_1, S_2, \ldots, S_m$ such that:*
   - *(a) $S_1 = R_1$,*
   - *(b) for $i > 1$, S_i consists of S_{i-1} and R_i with one the predecessor of the other,*
   - *(c) S_m is G.*
3. *The graph formed from G by reducing each of $R_1, R_2, \ldots, R_m$ to a single node with no loops is a spiral graph with zero or more edges removed.*

Proof. By Lemma 4, every region of more than one node is the union of two regions, one of which is the predecessor of the other. Using an argument of [17], we observe that if T is any region of more than $\frac{2}{3}k$ nodes, then either:

1. it is composed of two nonempty regions, one of which has more that $\frac{2}{3}k$ nodes, or
2. it is composed of two regions the larger of which has between $\frac{1}{3}k$ and $\frac{2}{3}k$ nodes.

Thus, the algorithm in Fig. 4 will generate the sequence of pairs (S_m, R_m), (S_{m-1}, R_{m-1}), $\ldots$, (S_1, R_1). This construction proves parts (1) and (2) of the lemma.

For part (3), we prove by induction on i that, after reduction, S_i is a spiral graph with some edges possibly missing. The basis $i = 1$ is trivial. For the induction, if R_i is the predecessor of S_{i-1} when

```
begin
   T ← G ;
   while T has more than (2/3)k nodes do
      begin
         let T be composed of regions T1 and T2,
            with T1 having no fewer nodes than T2 ;
         print(T, T2) ;
         T ← T1 ;
      end ;
   print(T, T)
end
```

FIG. 4. Computing the sequences of regions.

S_i is formed, then the result is immediate from construction (2b) in the definition of spiral graph. If S_{i-1} is the predecessor of R_i, then construction (2a) suffices. □

EXAMPLE 4. Consider the flow graph in Fig. 5.

The graph is composed of regions $\{1, 2\}$ and $\{3, 4, \ldots, 10\}$. The latter has more that $\frac{2}{3} \times 10$ nodes, so we print the pair $(S_m, R_m) = (\{1, 2, \ldots, 10\}, \{1, 2\})$. Then we work on $\{3, 4, \ldots, 10\}$, which can be partitioned in one of two ways, either by separating out 3 or 10. Supposing the latter, we have $(S_{m-1}, R_{m-1}) = (\{3, 4, \ldots, 10\}, \{10\})$. Then working on $\{3, 4, \ldots, 9\}$ we separate it into $\{3\}$ and $\{4, 5, \ldots, 9\}$. The former is R_{m-2}, and the latter has no more that $\frac{2}{3} \times 10$ nodes, so it is both S_{m-3} and R_{m-3}. The sequences of regions are thus found to be:

i	R_i	S_i
1	$\{4, 5, \ldots, 9\}$	$\{4, 5, \ldots, 9\}$
2	$\{3\}$	$\{3, 4, \ldots, 9\}$
3	$\{10\}$	$\{3, 4, \ldots, 10\}$
4	$\{1, 2\}$	$\{1, 2, \ldots, 10\}$. □

LEMMA 10. *Let G be a reducible flow graph partitioned into R_1, R_2, …, R_m as in Lemma 9. Let R_i have node listing A_i and acyclic ordering $\hat{A}_i$ for $1 \le i \le m$. Let H be the spiral graph constructed from G by reducing the R's to single nodes, then possibly adding some edges to make a spiral graph. Let B be a node listing for H. Let C be constructed from B by replacing in B each occurrence of node n_i of H representing R_i by the acyclic ordering $\hat{A}_i$. Then $A_1 A_2 \cdots A_m C$ is a node listing for G.*

Proof. Let P be an acyclic path in G. We can write P as $P_1 P_2$, where P_1 consists of the prefix of P until just before P leaves one of R_1,R_2,…,R_m in which it began. Surely P_1 is a subsequence of $A_1 A_2 \cdots A_m$. Consider the path Q in H consisting of those nodes of H representing the regions R_1,R_2,…,R_m through which P_2 travels in G. Q must be acyclic else P_2 enters the same region twice. Since regions can only be entered at their headers, the acyclicness of P_2, and hence of P, would be contradicted. This Q is a subsequence of B, and by Lemma 3, P_2 is a subsequence of C. Thus P is a subsequence of $A_1 A_2 \cdots A_m C$. □

THEOREM 1. *Every reducible flow graph of k nodes has a node listing of length no more than $k + ck \log k$, where $c = 3/\log \frac{3}{2} = 5.13$.*

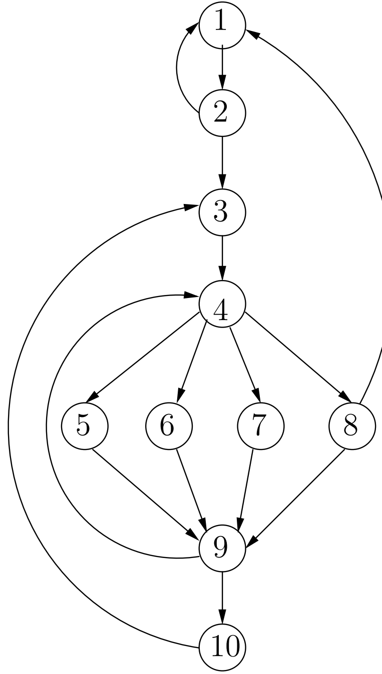


FIG. 5. Reducible Flow Graph

Proof. We proceed by induction on k. The basis $k = 1$ is immediate, so suppose the result for flow graphs of fewer than k nodes. By Lemma 10, we need only bound from above the length of the node listing $A_1 A_2 \cdots A_m C$ described in that lemma. Let w_i be the number of nodes in region R_i. Then by the inductive hypothesis, A_i has a node listing of length at most $w_i + cw_i \log w_i$.

By Lemma 8, the spiral graph H formed from G as in Lemma 10 has a node listing B in which the node representing R_i appears no more than $a \log (bk/w_i)$ times. Thus the node listing C has length at most $\sum_{i=1}^{m} aw_i \log (bk/w_i)$. Hence $A_1 A_2 \cdots A_m C$ has length bounded above by

$$\sum_{i=1}^{k}(w_i + cw_i \log w_i + aw_i \log (bk/w_i))$$

$$= \sum_{i=1}^{m} w_i + \sum_{i=1}^{m}(c-a)w_i \log w_i + a\sum_{i=1}^{m} w_i \log bk$$

$$= k + \sum_{i=1}^{m}(c-a)w_i \log w_i + ak \log k + ak \log b \quad (1)$$

Since $w_i \leq \frac{2}{3}k$ for all i, by Lemma 9, we have

$$\sum_{i=1}^{m}(c-a)w_i \log w_i \leq \sum_{i=1}^{m}(c-a)w_i \log \frac{2}{3}k = (c-a)(k \log k - k \log \frac{3}{2}). \quad (2)$$

Substituting (2) into (1) yields $k + ck \log k - ck \log \frac{3}{2} + ak \log \frac{3}{2} + ak \log b$. It suffices to observe that $-c \log \frac{3}{2} + a \log \frac{3}{2} + a \log b = -3 + 2 + 1 = 0$ by our choice of a, b and c. $\square$

In order for the method proposed in [1] to be feasible, it is necessary not only that short node listings exist, but that they can be easily constructed, else we might spend more time constructing the node listing than using it to drive an "efficient" data flow analysis algorithm. Fortunately, the construction we have proposed can be carried out efficiently if the number of edges in not too large.

THEOREM 2. *A node listing for a reducible flow graph of n nodes with at most $2n$ edges*[6] *can be constructed in $O(n \log n)$ time.*

Proof. It is straightforward to check that the constructions in Lemmas 6, 7, 8, and 10 and Theorem 1 require time proportional to the length of the node listing generated. The only possible problem concerns Lemma 9, where we proved the existence of sequence of regions $R_1, R_2, \ldots, R_m$ without showing how they could be found effectively. However, in [18] a method of parsing reducible flow graphs in less that $O(n \log n)$ time is presented, so the actual construction of $R_1, R_2, \ldots, R_m$ poses no problem. □

COROLLARY. *There is an $O(n \log n)$ bit vector step algorithm to compute live variables for flow graphs in which no node is the tail of more than two edges.*

Proof. Visit each node in the $O(n \log n)$ length node listing, performing at most three bit vector steps per node, as proposed in [1].

6. LOWER BOUNDS ON THE LENGTHS OF NODE LISTINGS

Recently, Markowsky and Tarjan [19] have shown that there exist reducible flow graphs with n nodes all of in-degree and out-degree two or less for which no weak node listing is shorter that $(n/2) \log n$. Thus, our construction is optimal to within a constant, for both strong and weak node listings.

7. EXTENSIONS

We might wish to apply the node listing technique to (1) finding live variables in flow graphs with e edges, where $e \gg n$, the number of nodes, or to (2) "forward" problems such as "reaching definitions" or "available expressions" [e.g., 5, 6, 16], whether or not $e >> n$. In each case, the work involved in visiting a node is proportional to the number of edges in (for "forward" problems) or out (for "backward" problems like live variables) of the node. We can extend Theorems 1 and 2 to consider *weights* of nodes, where weight can be defined as the in or out degree.

By the same method as we have used we can show that in $O(n \log n + e)$ time we can construct a node listing for a reducible flow graph with n nodes and $e \geq n - 1$ edges, such that the sum of the weights of the nodes in the listing is at most $W \log W$, where W is the sum of the weights of the nodes in the graph. Letting $W = e$ then yields an $O(e \log e)$ algorithm for all known bit vector oriented data flow analysis problems on reducible flow graphs. This result is comparable to the time bound for such problems obtained in [11,16].

A short length for weak node listings would have practical significance, since for certain problems such as "reaching definitions" or "live variable" (but not for "available expressions") a weak node listing is sufficient to perform a global flow analysis.

Fredrickson [20] recently reported that the constant c in Theorem 1 can be reduced to 2.01 by a finer treatment of Lemmas 6 and 7 and a further refinement of the parameters in Lemmas 8 and 9.

[6] This restriction on edges follows from the usual assumption that branches are two-way. Thus no flow graph resulting from a program can have more than twice as many edges as nodes.

ACKNOWLEDGMENTS

The authors wish to thank Brenda Baker for her helpful comments on the manuscript.

REFERENCES

1. K. KENNEDY, Node listing techniques applied to data flow analysis, *in* "Proc. 2nd ACM Conference on Principles of Programming Languages," January 1975, pp. 10-21.

2. F. E. ALLEN, Control flow analysis, *SIGPLAN Notices 5* (1970), 1-19.

3. D.E.KNUTH, An empirical study of FORTRAN programs, *Software: Practice and Experience* 1 (1971), 105-134.

4. M. S. HECHT AND J. D. ULLMAN, A simple algorithm for global data flow problems, *SIAM J. Computing* 4 (1975), 519-532.

5. J.COCKE, Global common subexpression elimination, *SIGPLAN Notices 5* (1970), 20-24.

6. A. V. AHO AND J. D. ULLMAN, "The Theory of Parsing, Translation and Compiling, Vol. 2, Compiling," Prentice-Hall, Englewood Cliffs, N. J., 1973.

7. M. SCHAEFER, "A Mathematical Theory of Global Flow Analysis," Prentice-Hall, Englewood Cliffs, N. J., 1973.

8. K. KENNEDY, A global flow analysis algorithm, *Int. J. Comp. Math.* 3 (1971), 5-15.

9. G. A. KILDALL, A unified approach to global program optimization, *in* "Proc. ACM Symposium on Principles of Programming Languages," October, 1973.

10. J. KAM AND J. D. ULLMAN, Global optimization problems and iterative algorithms, *J. Assoc. Comput. Mach.* 23 (1976), 158-172.

11. S. L. GRAHAM AND M. WEGMAN, A fast and usually linear algorithm for global flow analysis, *J. Assoc. Comput. Mach.* 23 (1976), 172-202.

12. C. M. GESCHKE, Global program optimizations, Ph. D. Thesis, Carnegie–Mellon University, 1972.

13. M. NEWEY, Notes on a problem involving permutations as subsequences, STAN-CS-73-340, Computer Science Department, Stanford University, Stanford, California, 1973.

14. M. S. HECHT AND J. D. ULLMAN, Flow graph reducibility, *SIAM J. Computing* 1 (1972), 188-202.

15. M. S. HECHT AND J. D. ULLMAN, Characterizations of reducible flow graphs, *J. Assoc. Comput. Mach.* 21 (1974), 367-375.

16. J. D. ULLMAN, Fast algorithms for elimination of common subexpressions, *Acta Informatica* 2 (1973), 191-213.

17. P. M. LEWIS II, R, E. STEARNS, AND J. HARTMANIS, Memory bounds on recognition of context-free and context-sensitive languages, *in* IEEE Conference Record of 6th Annual ACM Symposium on Switching Circuit Theory and Logical Design, October, 1966, pp. 190-202.

18. R. E. TARJAN, Testing flow graph reducibility, *J. Comput. System Sci.* 9 (1974), 355-365.

19. G. MARKOWSKY AND R. E. TARJAN, Lower bounds on the lengths of node sequences in directed graphs. IBM Watson Research Center, Yorktown Heights, N. Y., March 1975.

20. G. N. FREDRICKSON, Refinements to Aho and Ullman's node listing algorithm. Technical Report TR-404, Department of Computer Science, University of Maryland, September 1975.

Appendix B- Papers Written During the Project

In this appendix, we enumerate the following papers that were written during the project in order to document the results that we have obtained.

- *Some Interesting Results About Applications of Graphs in Compilers*
- *Parallel Processing on Linux with PVM and MPI*
- *Software for Parallel Processing*
- *Parallel Programming on PARAM*
- *Graphical User Interface using Qt*

Appendix C- PVM and MPI Function List

In this appendix we list the PVM and MPI functions used in the brute force programs implemented on the *PARAM 10000* supercomputer. These serve as a representative of the type of services provided by PVM and MPI. More information about PVM functions can be found in [1] and that about MPI can be found in [25].

PVM Function Table	
Function	Description
pvm_send()	Send a message in the active send buffer to the process with the given task ID and with the given tag.
pvm_recv()	Receive a message into the active receive buffer from the given process and having the given tag.
pvm_initsend()	Initialize the active send buffer to send data.
pvm_pkint()	Pack an array of integers into the send buffer. Similar functions for other data types too.
pvm_upkint()	Unpack the data from the active receive buffer into the user buffer.
pvm_mcast()	Multicast (i.e. broadcast to many but not all) a message in the active send buffer.

MPI Function Table	
Function	Description
MPI_Send()	Blocking send. Sends a message with the given tag to the specified process in the given communicator.
MPI_Recv()	Blocking receive. Receives a message with the given tag from the specified process in the given communicator.
MPI_Bcast()	Broadcasts a message to all the processes in the given communicator.
MPI_Probe()	Probes or a message with the given tag from the given process.
MPI_Comm_rank()	Gets the rank of the process in the given communicator .

Apart from these functions, many more features of PVM and MPI were used but are not documented for want of space.

❑❑❑

About the Project Report

In this project report, we have tried to document all the aspects of the project, right from the observations and the theoretical results known previously and those that were derived by us during the course of the project, the various algorithms developed and used during the project and the various programs and implementations carried out during the project. The project gave us an opportunity to learn some of the rich software tools available under a Linux system. One of them was a typesetting system called LaTeX, using which this project report was developed. The LaTeX file was called *report.tex* and it was converted into a Postscript file using the following commands:

```
$ latex report
$ bibtex report ..................................... generate bibliography
$ makeindex -s myright.ist report ...................... generate index
$ latex report
$ latex report ....................... run till all references resolved
$ dvips -o report.ps report.dvi ......................create a .ps file
```

The Postscript file was then directly fed to a Postscript Printer for taking the printouts.

Here, we briefly describe LaTeX as used in the project report. This project report was typeset in LaTeX and extensive use of various LaTeX features was made. New environments like *man, proof, observation, result, example* etc. were defined. The bibliographic database was maintained using BIBTeX using the *plain* style. The figures for this report were drawn using the software *xfig*, from where they were converted into combined Encapsulated Postscript (EPS)/*pstex* format for inclusion in the report. This format enables inclusion of mathematical formulae into the figures. Some hacking with the *xfig* file format was required to get the things correct. Index was generated using the *MakeIndex* utility. More information about LaTeX and the associated utilities can be found in [29, 15, 19]. The following LaTeX packages were used:

Package	Use
times	For using the Times font in the document
oldgerm	For Gothic style font in the certificate
graphics	For inclusion of Encapsulated Postscript files
subfigure	For managing and numbering subfigures
calc	For maintaining user defined counters
theorem	For theorems, results and the likes
fancyhdr	For customizing headers and footers
vpage	For customizing the page size and margins
amssymb	For special mathematical symbols like $\therefore$
amsmath	For typesetting mathematics
algorithmic, algorithm	For typesetting algorithms
pifont	For special Postscript characters like ✶
multicol	For multicolumn typesetting
makeidx	For automatic index generation
textfit	For arbitrarily enlarging text size in chapter numbers
titlesec	For customizing the chapter, part titles
longtable	For typesetting tables
ulem	For underline and likes
setspace	For single and double spacing
pstricks	For some Postscript tricks

In general, we can conclude that typesetting the project report using LaTeX was much more efficient than a WYSIWYG environment like Microsoft Word. Also, we had to spend lesser time with the report than we would had to do otherwise to produce the same quality document. The work of setting up LaTeX, learning LaTeX and trying out different packages was done by Rahul Joshi.

❑❑❑

References

[1] Al Geist, Adam Beguelin, Jack Dongarra, Robert Manchek, Weicheng Jiang and Vaidy Sunderam. *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, Massachusetts, 1994. `www.netlib.org/pvm3/book/pvm-book.html`.

[2] Antonio Larrosa Jiménez. *KDE Tutorial*. KDE Website, 2000. `www.kde.org`.

[3] Arnold D. Robbins. *Effective AWK Programming - A Users Guide to GNU Awk*. Specialized System Consultants, Free Software Foundation, April 1999.

[4] A. V. Aho and J. D. Ullman. Node Listing for Reducible Flow Graphs. *Journal of Computer and System Sciences*, 13(3):286–299, December 1976.

[5] A. V. Aho, Ravi Sethi and J.D.Ullman. *Compilers - Principles, Techniques, and Tools*. Addison Wesley, 1986.

[6] Clay Breshears and Asim YarKhan. A Beginner's Guide to PVM Parallel Virtual Machine. Technical report, Joint Institute of Computational Science, University of Tennessee, USA, 1998. `www-jics.cs.utk.edu/PVM/pvm_guide.html`.

[7] C. L. Liu. *Elements of Discrete Mathematics*. McGraw-Hill Book Company, 1977.

[8] David Pitts and Bill Ball. *Red Hat Linux 6 Unleashed*. Techmedia, 1999.

[9] D. M. Dhamdhere. *Systems Programming and Operating Systems*. Tata McGraw-Hill Publishing Company Ltd., 1999.

[10] Donald E. Knuth. *The TEXbook*. Addison Wesley, 1986. `www.cse.iitb.ernet.in`.

[11] Ellis Horowitz and Sartaj Sahni. *Fundamentals of Data Structures*. Computer Science Press Inc., 1994.

[12] Ellis Horowitz, Sartaj Sahni and Sanguthevar Rajasekaran. *Computer Algorithms*. Galgotia Publications Pvt. Ltd., 1999.

[13] Emily Angerer Crawford. PVM: An Introduction to Parallel Virtual Machine. Technical report, Office of Information Technology, High Performance Computing, 1996. `www.hpc.gatech.edu/seminar/pvm.html`.

[14] Etienne Bernard. A little manual for Lex and Yacc. A beginners guide to lex and yacc, September 1997.

[15] Harvey J. Greenberg. A Simplified Introduction to LaTeX.
`www.cudenver.edu/~hgreenbe/gtm.html`.

[16] Havoc Pennington. *GTK+ / Gnome Application Developement*. Red Hat Advanced Developement Labs, New Riders Publishing, 1999.
`www.redhat.com`.

[17] Horacio J. Pena. *Gnome Developers' Information*. The GNOME Project, 1998.

[18] John R. Levine, Tony Mason and Doug Brown. *lex & yacc*. O'Reilly Associates, 1995.

[19] Jon Warbrick. Essential LaTeX. Another LaTeX tutorial, February 1995.

[20] Karl Eichwalder, Miguel de Icaza, Fredrico Mena and others. *Gnome User Interface Library Reference Manual*.
`www.gnome.org`.

[21] Keith Reckdhal. Using Imported Graphics in LaTeX. Describes using .eps files in LaTeX, December 1997.

[22] Ken Kennedy. Node Listings Applied to Data Flow Analysis. In *Proceedings 2nd ACM Conference of Principles of Programming Languages*. Association of Computing Machinery, January 1975.

[23] Linux Documentation Project. *Linux Manual Pages*.
`www.linuxdoc.org`.

[24] Linux Documentation Project. *Linux Texinfo Documentation*.
`www.linuxdoc.org`.

[25] Marc Snir, Steve Otto, Steven Huss–Lederman, David Waker and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 1996.

[26] Matt Welsh. *Linux Installation and Getting Started*. Specialized System Consultants Inc., 1998.
`www.linuxdoc.org`.

[27] Medha G. Trivedi. *Graphical User Interface using Qt*. IEEE, 2001.

[28] Medha G. Trivedi. *Parallel Programming on PARAM*. IEEE, 2001.

[29] Michel Goossens, Frank Mittelbach and Alexander Samarin. *The LaTeX Companion*. Addison Wesley, 1994.

[30] M. K. Dalheimer. *Programming with Qt*. O'Reilly, 2000.

[31] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North – Holland Inc., New York, 1977.

[32] M. S. Hecht and J. D. Ullman. Characterizations of Reducible Flow Graphs. *J. ACM*, 21(3):367–375, July 1974.

[33] Nathan Thomas. An Introduction to C Developement on Linux. Technical report, Red Hat Software Inc., 1999.
`www.redhat.com`.

[34] Peter Mattis, Spencer Kimball and Josh MacDonald. GLib, GDK and GTK+ Reference Manuals, 1999.
`www.gtk.org`.

[35] Rahul U. Joshi. *Parallel processing on Linux with PVM and MPI*. Linux Gazette, April 2001.
`www.linuxgazette.com`.

[36] Richard M. Stallman. *GNU Make Manual*. Free Software Foundation, September 1999.

[37] Richard M. Stallman and Roland H. Pesch. *Debugging with GDB – The GNU Source Level Debugger*. Free Software Foundation, April 1998.

[38] Richard Stones and Neil Matthew. *Beginning Linux Programming*. Wrox Press, 1999.

[39] Robert Kiesling. *The teTEX HOWTO: The Linux teTEX Local Guide*. Linux Documentation Project, 1998.
`www.linuxdoc.org`.

[40] Robert Lafore. *C Programming using Turbo C++*. Techmedia, 1998.

[41] Robert Lafore. *Object Oriented Programming in Turbo C++*. Galgotia Publications Pvt. Ltd., 1998.

[42] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Book Company, 1997.

[43] Shane Hebert. Message Passing Interface (MPI) FAQ, 1997.
Newsgroup `comp.parallel.mpi`.

[44] Stephen Prata. *Advanced UNIX – A Programmer's Guide*. BPB Publications, 1986.

[45] Thomas Niemann. A Guide to Lex and Yacc, 1997.

[46] Tony Gale and Ian Main. GTK v1.2 Tutorial, 1999.
`www.gtk.org`.

[47] TrollTech. *QT Tutorial: The 14 Steps*. TrollTech, 2000.

[48] T. Sato and Brian V. Smith. *XFIG Users Manual*, 1999.

[49] Uday P. Khedker. *A Generalized Theory of Bit Vector Data Flow Analysis*. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, 1997.

[50] Vinay V. Kakade. *Software for Parallel Processing*. April 2001.

[51] Walter F. Tichy. Introduction to RCS Commands, 1993.

[52] Yashwant Kanetkar. *Unix Shell Programming*. BPB Publications, 1992.

[53] Yashwant Kanetkar. *Let us C*. BPB Publications, 1996.

[54] Yedidyah Langsam, Moshe J. Augenstein and Aaron M. Tanenbam. *Data Structures in C and C++*. Prentice Hall of India, 1996.

[55] Yukiya Aoyama and Jan Nakano. *RS/6000 SP: Practical MPI Programming*. International Techical Support Organization, IBM Corporation, Austin, Texas, 1999. `www.redbooks.ibm.com`.

Index

A

acyclic 114
acyclic ordering 20, 28, 39
acyclic restricted node listing 28
Al Aho 16, 21, 62, 71, 165
allbackpaths 113
allpaths 115
antichains 50
articulation point 77
available expressions 14
awk 84

B

back edge 7, 38, 72
basic block 5
basic path 17
binary parsable rfg 52
brute 116

C

code optimization 1
control flow analysis 2
control flow graph 5
cross edge 38

D

dag 11
data flow analysis 2
density 25
density of spiral graphs 30
depth 117
depth first ordering 16
depth-max 118
dfnize 119
dominance 120
dominator tree 7, 145
dominators 6, 72, 145

E

elimpaths 121
Exe.sh 122

F

feasible region 46
flow graph 5, 6
forward edge 8, 38

G

Glade 99
GNOME 99
graphinput 123
GTK+ 99

H

heursitic 124
higher density 31

I

inorder traversal 62
inverse T_1 transformation 76
inverse T_2 transformation 77
    type 1 77
    type 2 78
ismax 125
isspiral 126

J

J. D. Ullman 16, 21, 62, 71, 165

K

K. Kennedy 16, 19, 148

L

level of a node 49
lex 84
live variables 14
lower density 32

M
Majority Merge 23
make 85
manual 111
matrix 127
matrix of levels 50
max-RFG 128
maximal reducible flow graphs 30, 37, 77
Message Passing Interface 87
minimal node listing 18, 73
minimal reducible flow graphs 79
mmheuristic 129

N
natural loop 8
natural ordering 39
nl2b 130
non redundant paths 43

P
Parallel Virtual Machine 87
PARAM 10000 Supercomputer 87
partial ordering 49
Pascal's triangle 68
postorder traversal 39

R
RCS 85
reachability 42
reaching definitions 14
reduce 131
reducible 132
reducible flow graphs 8
redundant paths 43
region 10
restricted node listing 28

S
sheuristic 133
simple path 17
spiral 134
spiral graphs 33, 34, 57
strong node listing 18
sub_of_spiral 135
subgraph 136

T
T_1 transformation 9, 76, 141
T_1-T_2 analysis 9
T_2 transformation 9, 77, 141
  type 1 77
  type 2 77

V
verify-elimpaths 137
verifynl 138
vernl-trie 139
very busy expressions 14

W
weak node listing 19
  maximal rfgs 54
weighted dominator tree 64

Y
yacc 84