Computing Static Slices Using Reduced Graph

Thesis submitted in partial fulfillment of
the requirements for the degree
Of
Bachelor of Technology
In
Computer Science and Engineering

By
Rajalaxmi Sahoo
Roll No: 110cs0516

Under the Guidance of
Prof. D. P. Mohapatra
May, 2014

Department of Computer Science and Engineering

National Institute of Technology Rourkela
Rourkela, 769008

Certificate

This is to certify that the project entitled "Hierarchical slicing" submitted by Rajalaxmi Sahoo B.TECH student in the Department of Computer Science and Engineering, National Institute of Technology, Rourkela, India, in the partial fulfillment for the award of the degree of Bachelor of Technology. The thesis fulfills all requirements as per the regulations of this Institute and in our opinion has reached the standard needed for submission. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Prof. D. P. Mohapatra

Department of Computer Science and Engineering

National Institute of Technology Rourkela

Acknowledgement

On the submission of my Thesis report, I would like to extend my sincere thanks to my supervisor Dr. D.P. Mohapatra, for his constant motivation and support during my project work in the last one year. I truly appreciate and value his esteemed guidance and encouragement from the beginning to the end of this thesis. He has been our source of inspiration throughout the thesis work and without his invaluable advice and assistance it would not have been possible for us to complete this thesis.

I would also like to thank Mr. Subhrakanta Panda for his help in completing this thesis.

Rajalaxmi Sahoo

Declaration

I hereby declare that all the work contained in this report is my own work unless otherwise acknowledged. Also, all of my work has not been previously submitted for any academic degree. All sources of quoted information have been acknowledged by means of appropriate references.

Rajalaxmi Sahoo

NIT Rourkela

Abstract

Presently there is a colossal interest of programming items for which numerous software are produced. Before launching software testing must be done. Anyhow after tasteful completing the testing procedure we can't ensure that a product item is mistake free and it is clear that not all the lines in the source code are answerable for the error at a specific point. We accordingly require not taking the entire source code in the testing procedure and just concentrate on those areas that are the cause of failure by then. So as to discover those high-chance territories we need to develop a intermediate representation that specifies conditions of a program exactly. From the graph we process the slices. Slice is an independent piece of the system. Our program calculates the slices by not using the original graph but by using the reduced graph.

TABLE OF CONTENTS

Chapter 1

1. Introduction

Now a day's every work is done with the help of computer for which software are developed. These product results are getting to be very unpredictable and their quality have been essential limited by the expense and time elements.

- Statistics shows that almost 55% of the software's built in these days are not useful because of their inability to meet the requirements. Hence Software testing activity is very important for launching new software in the market. Various methods are already developed for software testing from which intermediate graph is one of convenient form. Slicing is an important technique which is widely used in software testing. Some of these applications are Program Understanding, Debugging, Testing, Software maintenance, Parallelization, Program Specialization and reuse. Slicing has a huge application in software testing. Various type of slice technique exists.

1.1 Motivation of project

After effectively doing the product testing we can't ensure that our product is completely error free and it is evident that not all the lines of the source code are answerable for the error .so we have to test only those areas where possibility of error to occur is very high. This can be done by using program slicing.

1.2 Objective of our project

Our objective is first to reduce intermediate graph, then compute slice using Horowitz Algorithm, then to compute slice using our proposed Algorithm, Improve our proposed Algorithm and compare the slices obtained from Horowitz and our algorithm.

1.3 Organization of the project

The organization of rest of project is as below:

In chapter 2 we represent the basic definitions related to our project

In chapter 3 we present some of the work related to this area.

In chapter 4 we explain the different algorithms proposed by us.

In chapter 5 we give an overview about Eclipse and implementation result.

In chapter 6 we write conclusion and some of future work related to this area.

Chapter 2

2 Basic Definitions

In this chapter we describe some of the basic definitions which will be used in our project.

2.1 Program slicing

Program slice consist of a set of statements which affect the value of a variable at a particular point of interest .That point of interest is called slicing criterion.

A slicing criterion consists of a pair of statement and variable.

Example:-

Figure 2.1

The source code

```
1       BEGIN
2         READ(A,B)
3          SUM:=0;
5          SUM=A+B;
6        WRITE(SUM)
7       END.
```

Slice On Criterion (6,{sum}).

```
        BEGIN
              READ(A,B)
               SUM:=0;
               SUM=A+B;
             WRITE(SUM)
    END.
```

Slice on criterion <5,{A}>.

```
            BEGIN
            READ(A,B)
END
```

2.2Types of program slicing

Different types of slicing that exists are as follows:

2.2.1 Forward Slicing

In this case the slice is computed by working forward from the given point finding those statements that can be affected by changes to the specified variables.

2.2.2 Backward slicing

In this case the slice is computed by working backward from the given point finding those statements that can be affected by changes to the specified variables.

2.2.3 Static slicing

A static slice is a slice which contains the statements which affect the value of a variable at particular point for all possible input.

2.2.4 Dynamic slicing

A dynamic slice is a slice which contains the statements which affect the value of a variable at particular point for a particular input.

```
1   Public class Demo
    {
2       Static int addition(int x, int y)
          {
3                 Return (x+y);
          }
4         Public static void main(final String[] arg)
          {
5             int count=1;
6             int sum = 0;
7             while(count<11){
8                   sum = addition(sum,count);
9                   count=addition(count,1);
                        }
10                  System.out.println("sum="+sum);
11                  System.out.println("count="+count);
                    }
      }
```

Figure 2.2

Forward slice w.r.t criterion (2,sum)

```
1   Public class Demo
    {
2       Static int addition(int x, int y)
          {
3                 return (x+y);
          }
4         Public static void main(final String[] arg)
          {
5             int sum = 0;
6             while(count<11){
7                   sum = addition(sum,count);
                        }
8                   System.out.println("sum="+sum);
                    }
    }
```

Figure 2.3

Backward slice w.r.t criterion (7,i)

```
1   Public class Demo
    {
2       Static int addition(int x, int y)
          {
3                 return (x+y);
          }
4       Public static void main(final String[] arg)
          {
5           int count=1;
6           while(count<11){
7                 count=add(count,1);




                                        }
                }

        }
```

Figure 2.4

Source program 2

```
Import java.util.*;
class Demo1
{
   Public static void main(String args[])
{
            1   int sum=0;
            2   int num;
            3   int count=1;
            4   int product=1;
            5   Scanner s;
            6   s=new Scanner(System.in);
            7   num=s.nextInt();
            8   While(count<=num)
        {
            9       sum=sum+count;
            10      product=product*count;
            11      count=count+1;
        }
            12  System.out.println("sum is :"+sum);
            13  System.out.println("product is :"+product);
        }
}
```

Figure 2.5

Static slicing w.r.t criterion (13,prod)

```
Import java.util.*;
class Demo1
{
   Public static void main(String args[])
{
            1   int num;
            2   int count=1;
            3   int product=1;
            4   Scanner s;
            5   s=new Scanner(System.in);
            6   num=s.nextInt();
            7   While(count<=num)
        {
            8       product=product*count;
            9       count=count+1;
        }
            10  System.out.println("product is :"+product);
        }
}
```

Figure 2.6

Dynamic slicing w.r.t criterion (13,sum,i=5 & n=9)

```
Import java.util.*;

class Demo1

{

   Public static void main(String args[])

{

            1   int sum=20;
            2   int num,count;
            3   int prod=10;
            4   Scanner s;
            5   System.out.println("enter the input");
            6   s=new Scanner(System.in);
            7   num=s.nextInt();
            8   count=sc.nextInt();
            9   if(count<=num)
        {
            10      sum=sum+product;
        }
            11  System.out.println("sum is :"+sum);
                }

}
```

Figure 2.6

2.3 Intermediate Representation

2.3.1 Control Flow Graph

A Control Flow Graph is an intermediate representation. It has an entry section called START and an exit section called STOP, where every

node refers to the statement of the program. There is a coordinated edge from one node to other in the control flow graph. Edges in a CFG are of two sorts. One is T edge different is F edge. An edge is known as a T edge, if control flows along that edge when the predicate at the node is evaluated to be genuine and An edge is known as a F edge, if control flows along that edge when the predicate is to be false.

Example-

```
x=10;
count=5;
while(count>0){
if(x<20)
inc(x);
count=count-1;
}
```

Figure 2.7

Control flow graph for the Figure-2.7

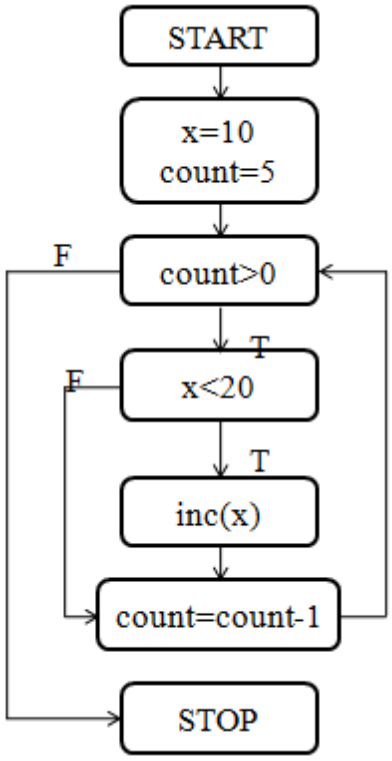


Figure 2.8

2.3.2 Data Dependence Graph

Data dependency in a control flow graph exists from node 1 to 2 if the following conditions are satisfied
[1]A variable say V is defined in Node 1

[2] Node 2 uses the variable V for computation

[3]Control can flow from 1 to 2
Example-

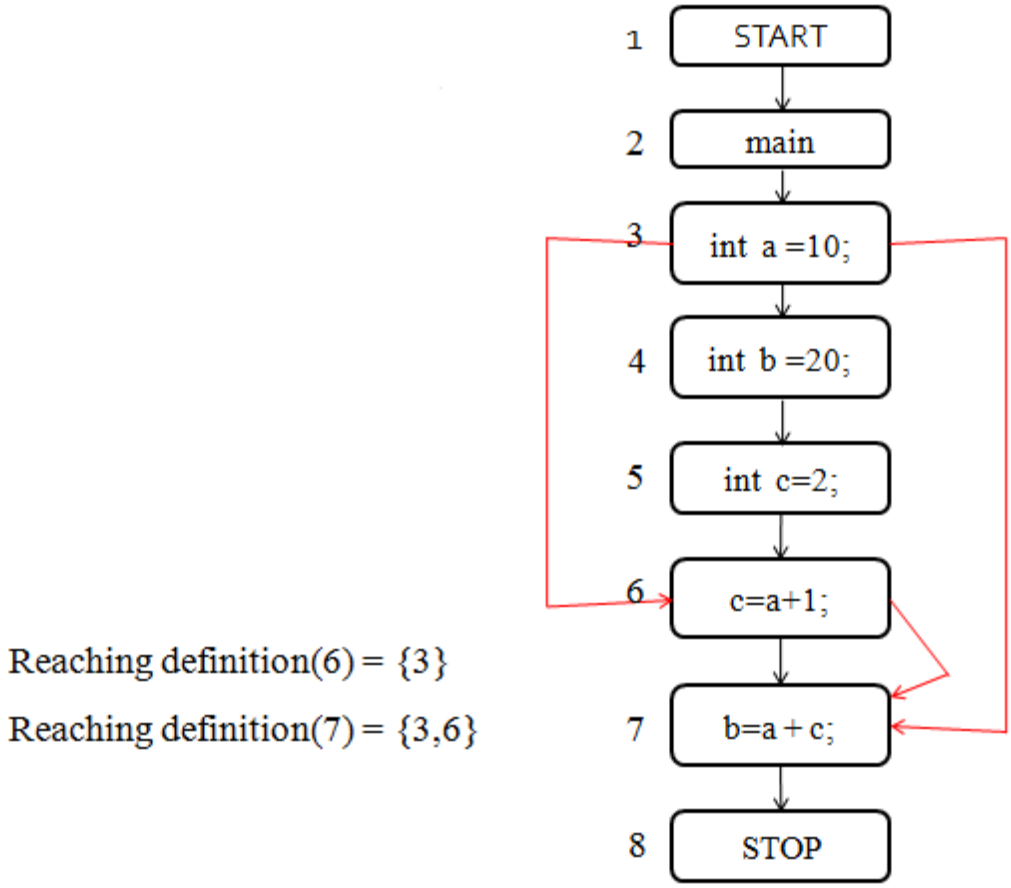


Figure-2.9 Data dependence graph for the **Figure 2.8**

2.3.3 Program dependence graph

Ferrante et al. presented a new mechanism of program representation called Program Dependence Graph (PDG).

PDG of an OOP is a directed graph in which

[1] nodes represent statements and predicates

[2] edges represent data/control dependence among the nodes

Example-

```
integer  i, sum, x
1.  read(x)
2.  sum = 0
3.  i = 1
4.  while(i<=x)
5.        sum = sum + 1
6.        i = i + 1
     endwhile
7.  write(sum)
8.  write(i)
```

Figure 2.10 A sample program

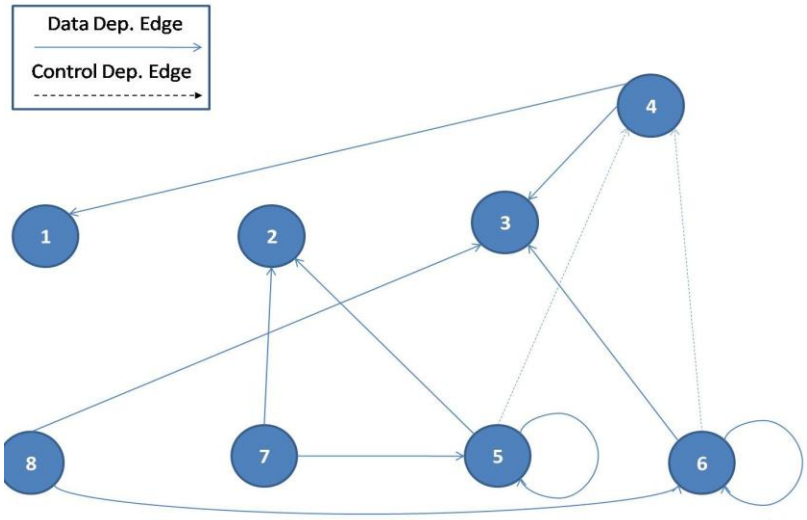


Figure 2.11 Program Dependence Graph(PDG) for the example **Figure 2.10**

2.3.4 System dependence graph

PDG cannot handle programs with multiple procedures. Horwitz et al. proposed an intermediate representation called as system dependence graph (SDG). SDG is based on procedure dependence graphs. Slice is

computed as a graph reachability problem .Same as PDG except that it includes vertices & edges for call statements, parameter passing & transitive dependence due to calls

Example-

Figure 2.12

```
void main()
{
      int i = 1;
      int sum = 0;
      while (i  < 11)
      {
            sum = add (sum, i);
            i = add (i, 1);
      }
      printf (“sum  =  %d\n”,  sum);
      printf (“i  =  %d\n”, i);
}
int add (int a, int b)
{
      return (a + b);
}
```

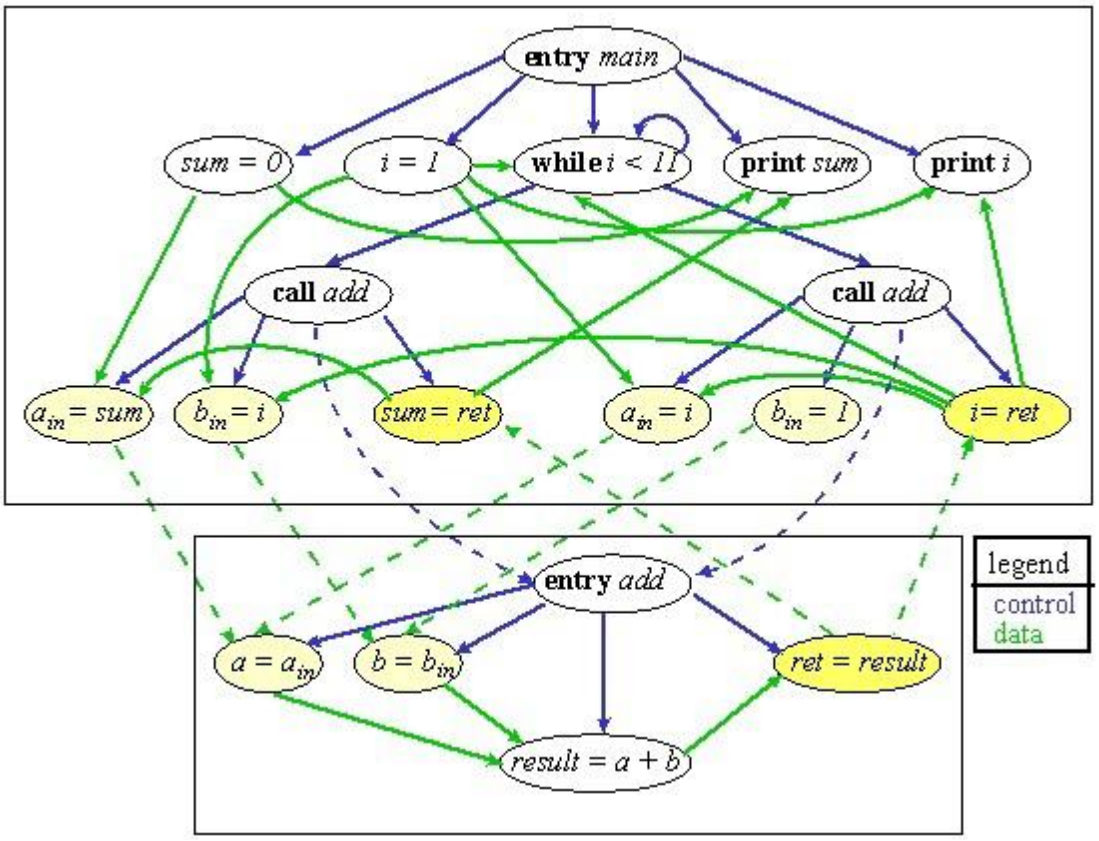


Figure 2.13 System Dependence Graph (SDG) for the example **Figure 2.12**

Chapter 3

Related Work

Many theories have been already proposed regarding computation of slicing and intermediate representation of program. Program slicing was proposed by **Weiser** in 1982.According to him A slicing criterion of a program P is a tuple (i, V), where i is a statement in P and V is a subset of the variables in P. Program slicing is a method for automatically decomposing programs by analyzing their data flow and control flow[1] .According to **Horwitz** consider the problem of inter-procedural slicing-generating for which he introduce a new kind of graph called a system dependence graph, which solve the problem of inter-procedural dependency[5]. According to **Ferrante** he develops an intermediate program representation, called the program dependence graph (PDG), that makes explicit both the data and control dependency [4]. According to **Larsen and Harold** SDG had no provision to incorporate the O-O features like class, inheritance, polymorphism, etc. Larson and Harrold were the first to consider these O-O features for slicing by extending the SDG for OOPs. According to **Chen et al.** discussed different dependencies possible in a Java program and proposed slicing of classes based on Program Dependence Graph (PDG). In their method, the program dependency graph consists of a set of independent PDGs. In slicing of classes, the slicing criterion taken is <s, v, class>, where s is

the statement number, v is the variable and class is the name of the class to be sliced. The slice is computed by traversing backward from s and marking all the statements and data members used in the class based on the PDG. According to **Wang et al.** proposed slicing of Java programs by using compressed byte code traces. They represented the byte code corresponding to an execution trace of a Java program. Then, through backward traversal of the execution trace, they determined the control and data dependencies on the slicing criterion. This approach requires the trace table to be represented for each method. If a program will have too many methods, then this approach will be disadvantageous to compute slices. According to **Harrold et al.** have proposed traversal algorithms to identify the dangerous edges for safe regression test selection. The dangerous edge is designed to be an edge e such that for each input i causing P to cover e, P(i) and P'(i) may behave differently due to differences between P and P', where P and P' are the programs under consideration and the modified program respectively. The dangerous edge is identified by traversing the proposed Java Interclass Graph (JIG). This method compared two nodes of P and P' in the JIG to identify the execution path of a test case in P and P', so that it can be known whether any edge is dangerous or not. According to **Jeffrey and Gupta**, proposed a method for prioritizing the test cases for regression testing based on the coverage of relevant slice of the output of a test case. They assigned test case weights to the test cases to determine their

priority. They determined the test case weight by summing up the number of statements present in the relevant slice and number of statements exercised by the test case. According to **Korel et al** prioritized the regression test suite by considering the state model of the system. Whenever, the source code was modified, the corresponding change in its state model was identified. These modified transitions along with the runtime information were used to prioritize the test cases. According to Tao et al. Tao et al. applied hierarchical slicing for regression testing of object-oriented programs. In their approach, they have proposed to maintain separate graphs for packages, classes, methods and statements even if they were not affected by the change. This approach requires more space for intermediate graph representation. This is because with the increase in the program complexity, there will be an increase in the number of packages, classes, methods and statements which are required to be represented as separate graphs.

Chapter 4

Our Work

4.1 Edge Reducing Algorithm

This algorithm is known as edge reducing algorithm. This is used to remove redundant edge from a graph.

Edge Reducing Algorithm

Input- Intermediate Graph G (N, E), where N is the set of nodes, E is the set of edges.

Output- A graph containing a reduced set of edges F

```
F: = E; // Initialize F.
for each (u,v) ε F do
      G := E - (u,v);
      S := u; // S is a temporary set.
        for each (x,y) ε G do
            If x c S then
                S := S U {y};
            End If
        End for
        If v c S then
        E := E - (u,v);
        End If
End for
F := E; // F is the set of redundant free edges.
```

Explanation

An edge (a,c) is said to be redundant if the vertex ‘c’ can be reached via other vertices and hence can be removed.

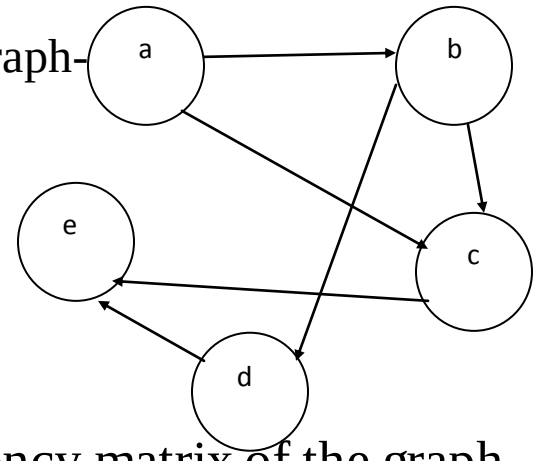
Ex:- from the graph a set A={ (a,b), (b,c), (a,c), (c,d)}

Here as (a,b) and (b,c) both are present , vertex ‘c’ can be reached from vertex ‘a ’ via vertex ‘b’ . Hence (a, c) can be deducted from the set A.

Example-

Figure 4.1

Input graph-

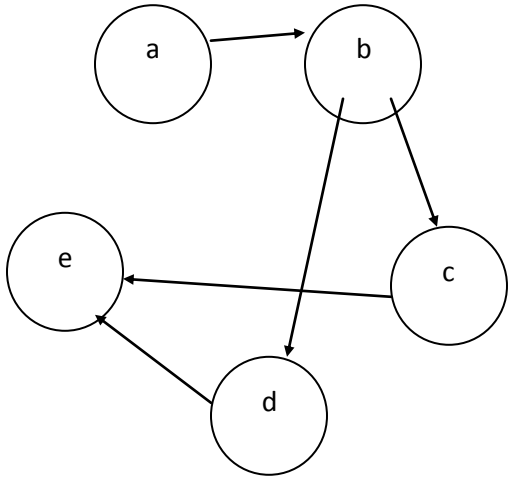


Adjacency matrix of the graph

[0 1 1 0 0]
[0 0 1 1 0]
[0 0 0 0 1]
[0 0 0 0 1]
[0 0 0 0 0]

Reduced graph:

Figure 4.2



Adjacency matrix of the graph

[0 1 0 0 0]
[0 0 1 1 0]
[0 0 0 0 1]
[0 0 0 0 1]
[0 0 0 0 0]

4.2 Horowitz 2-Phase algorithm

This algorithm uses a slicing criterion which is generally represented by <S, V>, where S is the statement and V may be a variable .The static backward slice of a statement in a program is then calculated using a two pass graph reachability algorithm proposed by Horwitz.

Steps

Input : A graph G(V,E)

Output: Slice

Steps

1.pass1:

1.1.traverse the graph in backward direction through the edges except parameter-out edges.

2.pass 2

2.1.traverse the graph in backward direction except parameter-in and call edges.

3.Find the slice by taking union of slice from pass 1 and pass 2.

Algorithm

declare
G: a system dependence graph
V:a set of vertices in G
Kinds: set of kinds of edges
v, w: vertices in G
worklist: a set of vertices in G
begin
worklist := v
while worklist != Ø do
select and remove a vertex Y from worklist
mark v
for each unmarked vertex w such that there is an edge w->v whose kind is not in *Kinds* do
Insert w into workList
od
od
end

4.3 Our Proposed Slicing Algorithm

This algorithm is used to compute the slice from a given input intermediate graph. In this algorithm we have not taken any edge restrictions so it can be applied to any program.

Steps for the Algorithm

Input: A graph G(V,E)
Output : slice
Steps
1.Pass1:
1.1.Traverse G in forward direction from the slicing criterion
2Pass2:
2.1.Traverse G in backward direction from each node in pass1

Algorithm:

Input: A graph
Output: Slice

1. current_node = desired_node (slicing criterion), mark it as visited.
2. insert current_node in Queue, Q.
3. traverse in forward direction through all dependency edges from current-node.
4. for each node during traversal
5. If not visited
6. Mark it visited.
7. Insert node in Q.
8. End if
9. Move next.
10. while Q not empty
11. current_node = dequeue (Q).
12. add current_node to set U.

13. traverse in backward direction through all dependency edges from current-node.
14. for each node during traversal
15. If not visited
16. Mark it visited.
17. Add node to U.
18. End if
19. Move next.
20. End for
21. End while

Chapter 5

Implementation and results

This chapter consists of the details of our implementation and the results.

5.1 Tools used

We use the following tools in order to implement and code the programs and finally to get the result.

1. Eclipse

Eclipse is a multi-language software development environment which is used to write java programs.

5.2 Screenshots of implementation

```
Program Main      procedure A(x,y)      procedure Add(a,b)      Procedure Increament(z)
Sum:=0;           call Add(x,y);        a=a+b;                  call Add(z,1)
i:=1;             call increament(y)    return                  return
While i<11 do     return
   call A(sum ,i)
od
End(sum ,i)
```

Input program

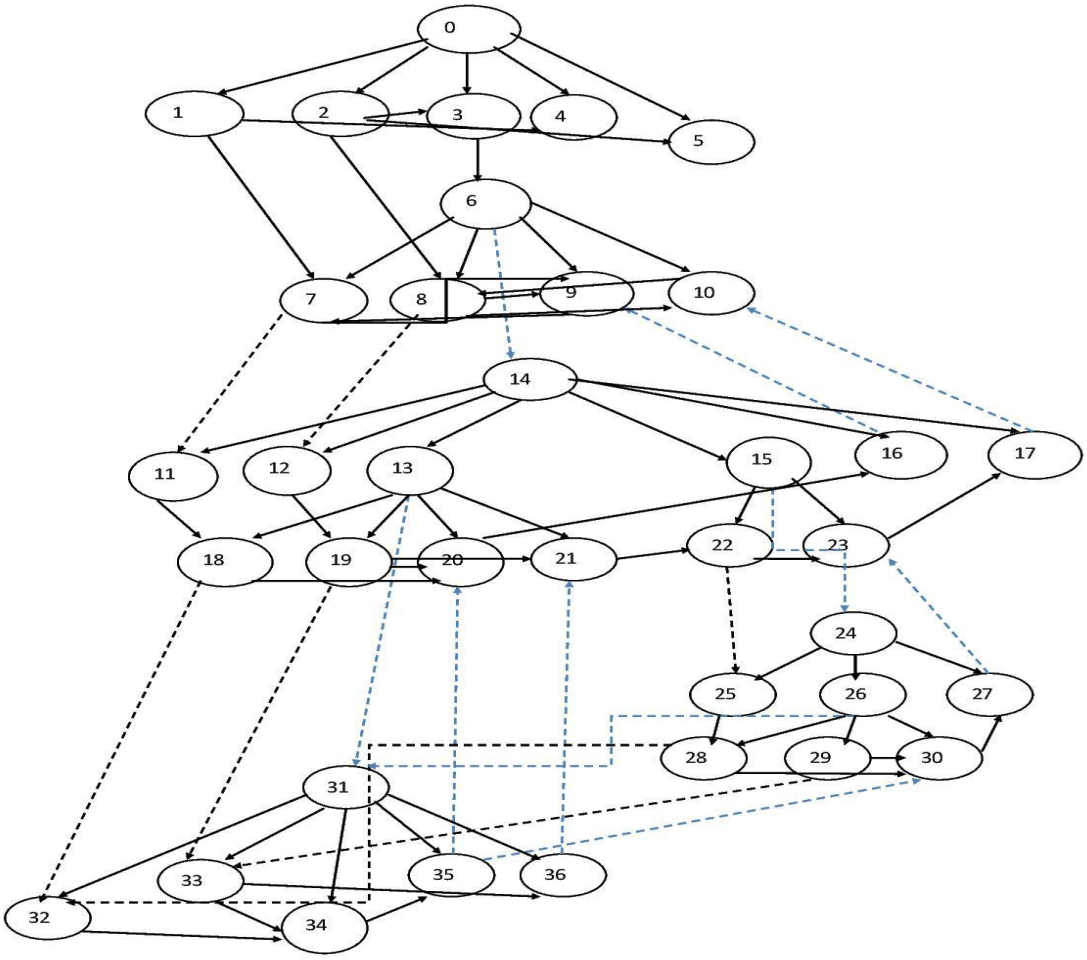


Figure 5.1 Input graph(SDG of the input program)

Where Node 0,1,2…. represents

0:Enter Main
1:sum = 0
2:i=1
3:while i< 11
4:FinalUse(sum)
5:FinalUse(i)
6:call A
7:x_in=sum

8:y_in=i
9:sum=x_out
10:i=y_out
14:Enter A
11:x=x_in
12:y=y_in
13:call Add
15:call Inc
16:x_out=x
17:y_out=y
18:a_in:=x
19:b_in:=y
20: x=a_out
21: y=b_out
22:z_in=y
23:y=z_out
24:Enter Inc
25: z=z_in
26:call Add
27: z_out=z
28:a_in=z
29:b_in=1
30:z=a_out
31:Enter Add
32:a=a_in
33:b=b_in
34:a=a+b
35:a_out=a
36:b_out=b

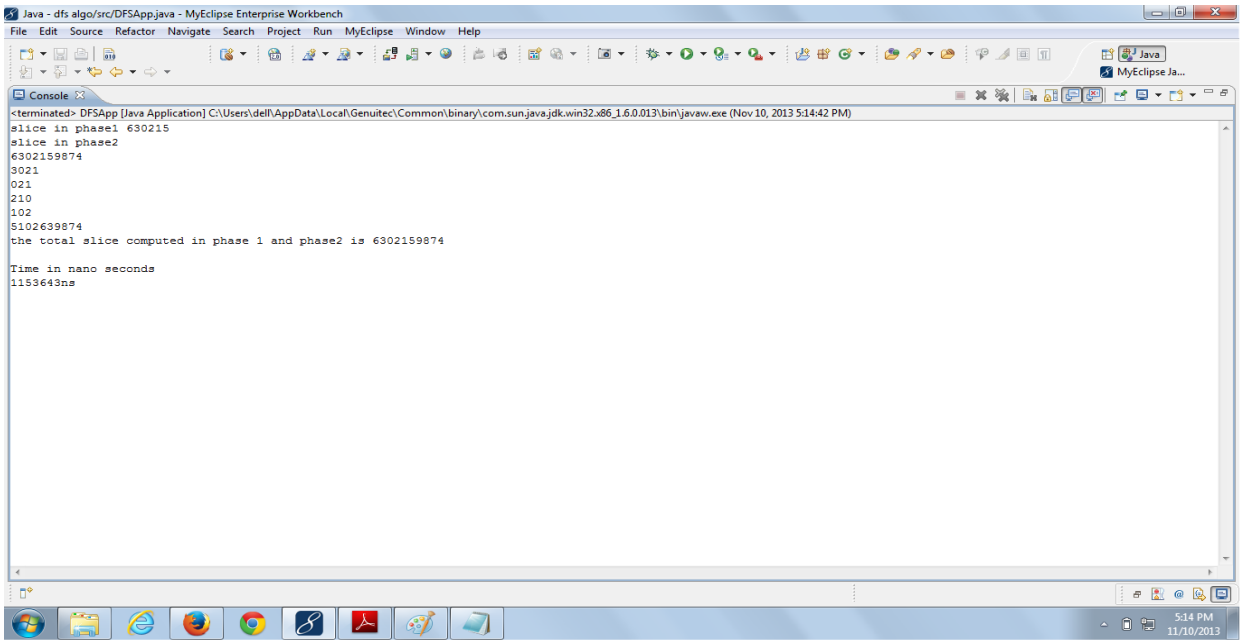


Figure 5.2

Time taken to compute the slices by Horowitz 2-phase algorithm by taking input as original graph

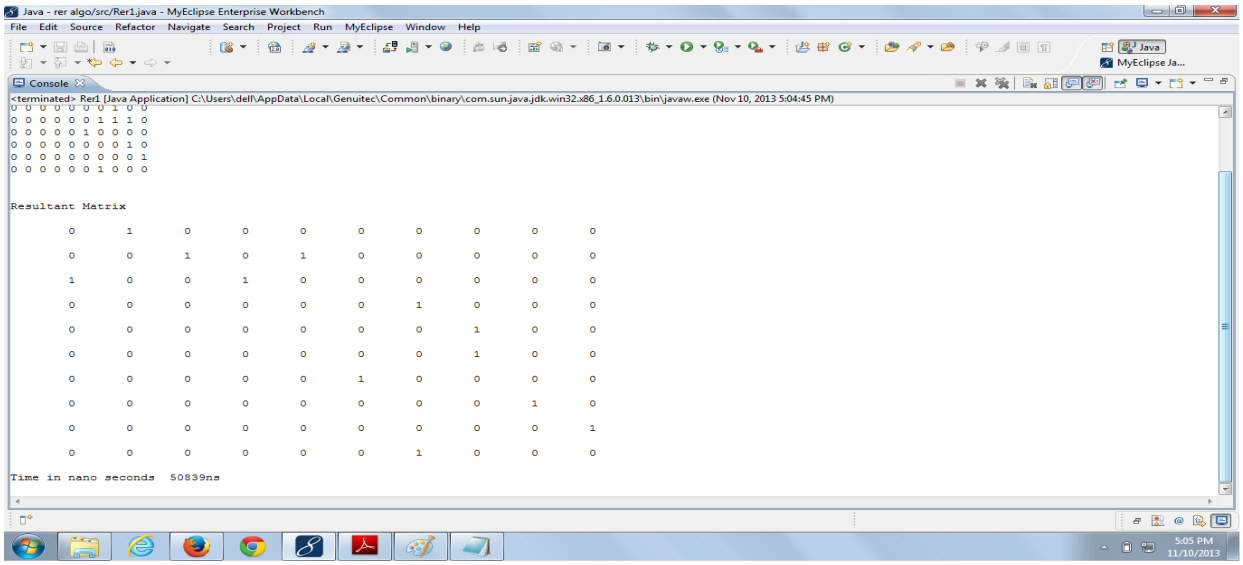


Figure 5.3

Time taken by RER Algorithm to reduce the graph

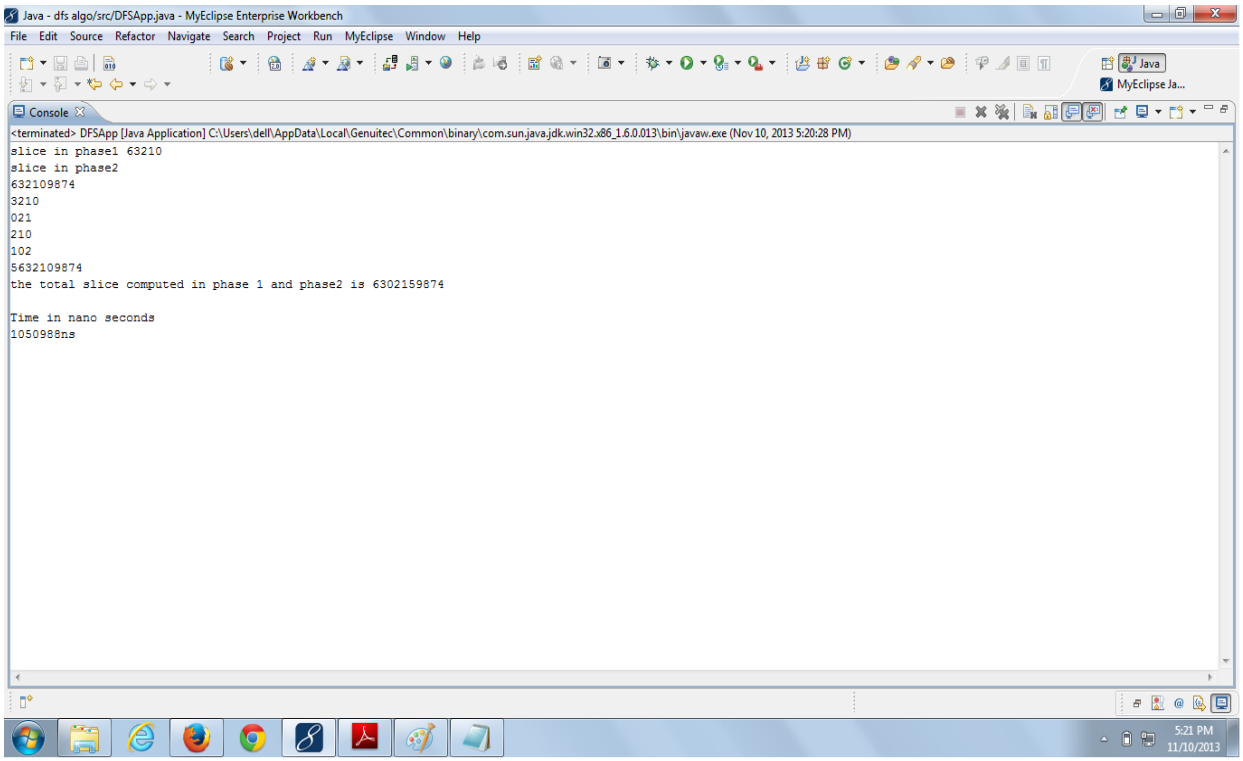
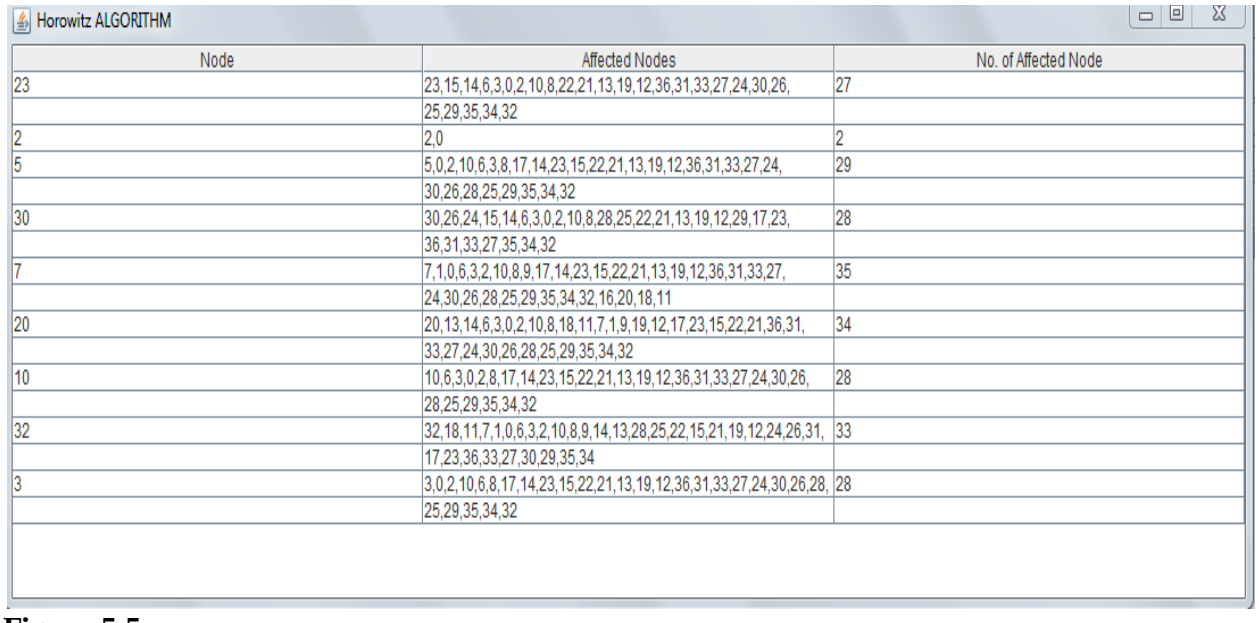


Figure 5.4

Time taken to compute the slices by Horowitz 2-phase algorithm by taking input as reduced graph

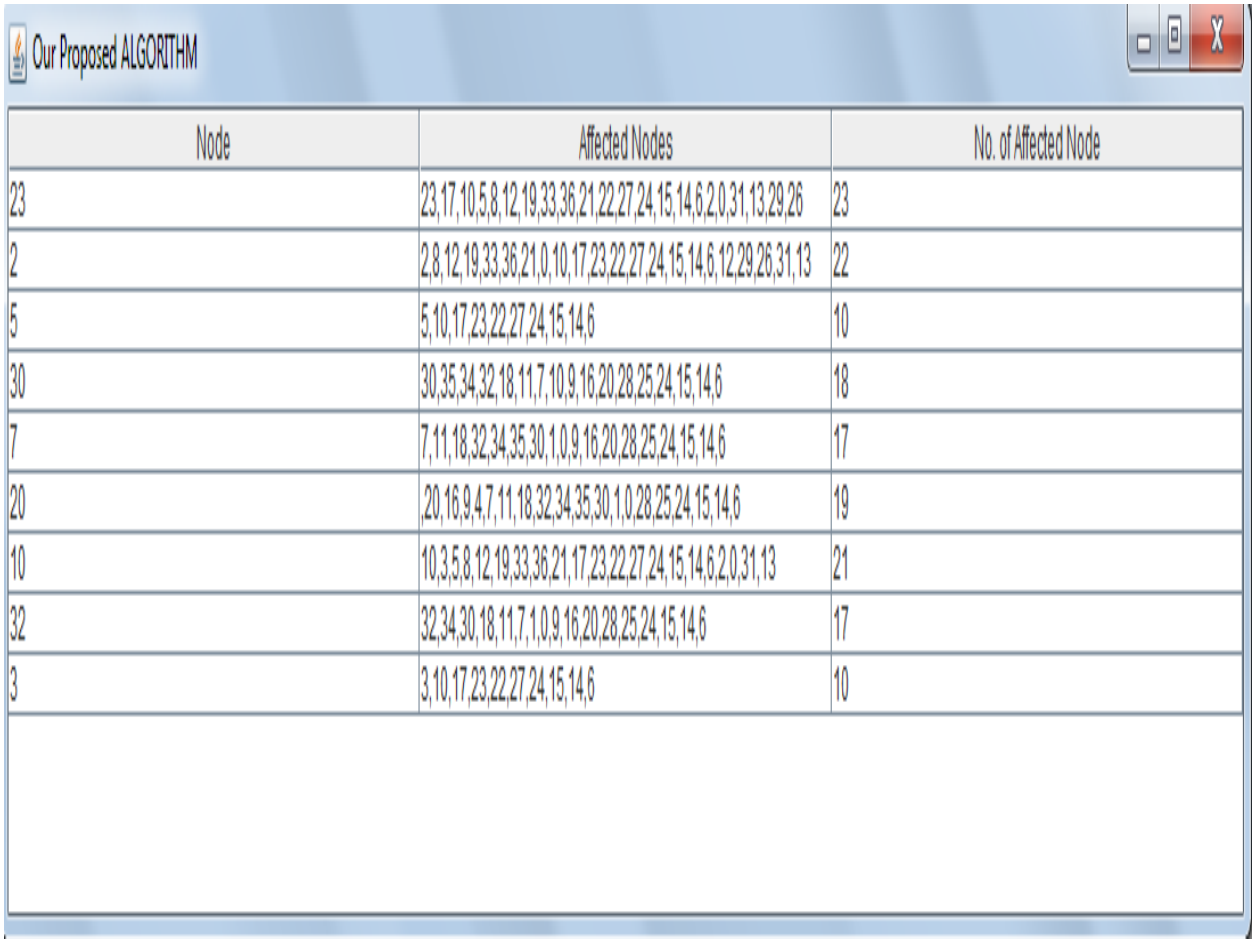


Horowitz ALGORITHM

Node	Affected Nodes	No. of Affected Node
23	23,15,14,6,3,0,2,10,8,22,21,13,19,12,36,31,33,27,24,30,26,	27
	25,29,35,34,32	
2	2,0	2
5	5,0,2,10,6,3,8,17,14,23,15,22,21,13,19,12,36,31,33,27,24,	29
	30,26,28,25,29,35,34,32	
30	30,26,24,15,14,6,3,0,2,10,8,28,25,22,21,13,19,12,29,17,23,	28
	36,31,33,27,35,34,32	
7	7,1,0,6,3,2,10,8,9,17,14,23,15,22,21,13,19,12,36,31,33,27,	35
	24,30,26,28,25,29,35,34,32,16,20,18,11	
20	20,13,14,6,3,0,2,10,8,18,11,7,1,9,19,12,17,23,15,22,21,36,31,	34
	33,27,24,30,26,28,25,29,35,34,32	
10	10,6,3,0,2,8,17,14,23,15,22,21,13,19,12,36,31,33,27,24,30,26,	28
	28,25,29,35,34,32	
32	32,18,11,7,1,0,6,3,2,10,8,9,14,13,28,25,22,15,21,19,12,24,26,31,	33
	17,23,36,33,27,30,29,35,34	
3	3,0,2,10,6,8,17,14,23,15,22,21,13,19,12,36,31,33,27,24,30,26,28,	28
	25,29,35,34,32	

Figure 5.5

Output of Horowitz Algorithm



Node	Affected Nodes	No. of Affected Node
23	23,17,10,5,8,12,19,33,36,21,22,27,24,15,14,6,2,0,31,13,29,26	23
2	2,8,12,19,33,36,21,0,10,17,23,22,27,24,15,14,6,12,29,26,31,13	22
5	5,10,17,23,22,27,24,15,14,6	10
30	30,35,34,32,18,11,7,10,9,16,20,28,25,24,15,14,6	18
7	7,11,18,32,34,35,30,1,0,9,16,20,28,25,24,15,14,6	17
20	,20,16,9,4,7,11,18,32,34,35,30,1,0,28,25,24,15,14,6	19
10	10,3,5,8,12,19,33,36,21,17,23,22,27,24,15,14,6,2,0,31,13	21
32	32,34,30,18,11,7,1,0,9,16,20,28,25,24,15,14,6	17
3	3,10,17,23,22,27,24,15,14,6	10

Figure 5.6
Output of our proposed Algorithm

Seriel Number	Node number	Number of slice from Horowitz Algorithm	Number of slice from our proposed Algorithm
1	23	27	23
2	2	2	22
3	5	29	10
4	30	28	18
5	7	35	17
6	20	34	19
7	10	28	21
8	32	33	17
9	3	28	10

Table 5.1
Comparison between Horowitz and our proposed Algorithm

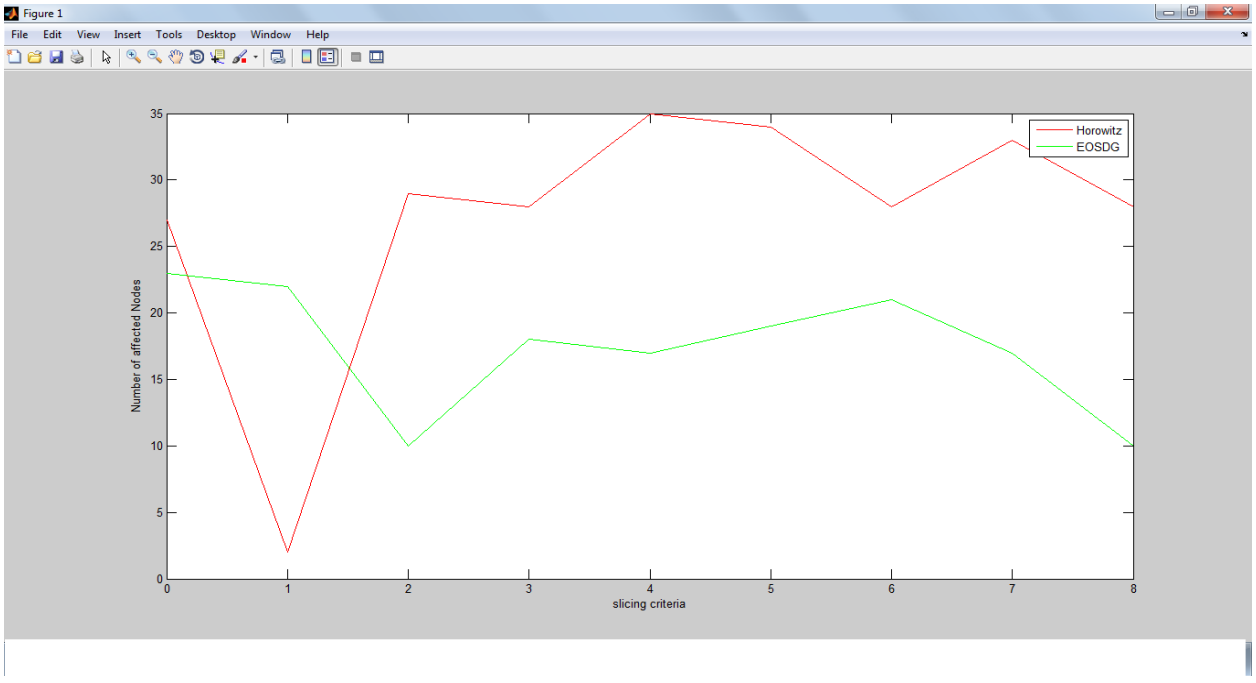


Figure 5.7
Comparison between Horowitz and our proposed Algorithm

Chapter 6

Conclusion and future work

6.1 Conclusion

We have Implemented RER Algorithm to reduce the graph and We have proposed an algorithm to compute the slices. We have implemented Horowitz 2-phase algorithm.

6.2 Future work

Our future work is to implement a new better slicing algorithm which will be able to support oops concepts like polymorphism.

References

[1]M Weiser-Program Slicing- In Proceedings of the 5th International Conference on Software, pages 439-449.San Diego, California, USA, 1981.

[2]A Krishnaswamy-Program Slicing: An Application of Object-Oriented Program Dependence Graphs. Technical Report TR94-108, Department of Computer Science, Clemson University, 1994.

[3]S Horowitz and T Reps-The use of Program Dependence Graphs in Software Engineering-in Fourteenth International Conference on Software Engineering, Melbourne, pages 392-411, 1992.

[4]J Ferrante, K J Ottenstein, and J D Warren- The Program Dependence Graph and its Use in Optimization.ACM Transactions on Programming Languages and System, 9(3):319-349, 1987.

[5] S Horowitz, T Reps, and D Binkley- Inter-procedural Slicing using Dependence Graphs. ACM Transactions on
Programming Languages and Systems, 12(1):26-60, 1990.

[6] L Larsen and M J Harrold-Slicing Object-Oriented software-In Proceedings of the 18th IEEE International Conference on Software Engineering, pages 495-505, 1996.

[7]M J Harrold and et al-Regression Test Selection for Java Software- In Proceeding of the ACM Conference on OO Programming, Systems, Languages, and Applications (OOPSLA'01), pages 312-326, 2001.

[8] D Jeffrey and N Gupta- Test Case Prioritization using Relevant Slices- In Proceedings of 30th Annual International Computer Software and Applications Conference, pages 411-420, 2006.

[9] D P Mohapatra, R Mall, and R Kumar- An Edge Marking Technique for Dynamic Slicing of Object-Oriented Programs- In 28th International Computer Software and Applications Conference (COMPSAC 2004), Design and Assessment of Trustworthy Software-Based Systems, pages 60-65, IEEE Computer Society, 2004.

[10] D P Mohapatra, R Mall, and R Kumar. An Overview of Slicing Techniques for Object-Oriented Programs.
Informatica (Slovenia), 30(2):253-277, 2006.