White Box Testing

CS241

WHITE-BOX TESTING

- Depends on the internal structure of the program
- Also known as *structural testing*
- The criteria for selecting testcases are generally quite precise
- Three different approaches-
  - Control flow based testing
  - Data flow based testing
  - Mutation testing

Control Flow Based Testing

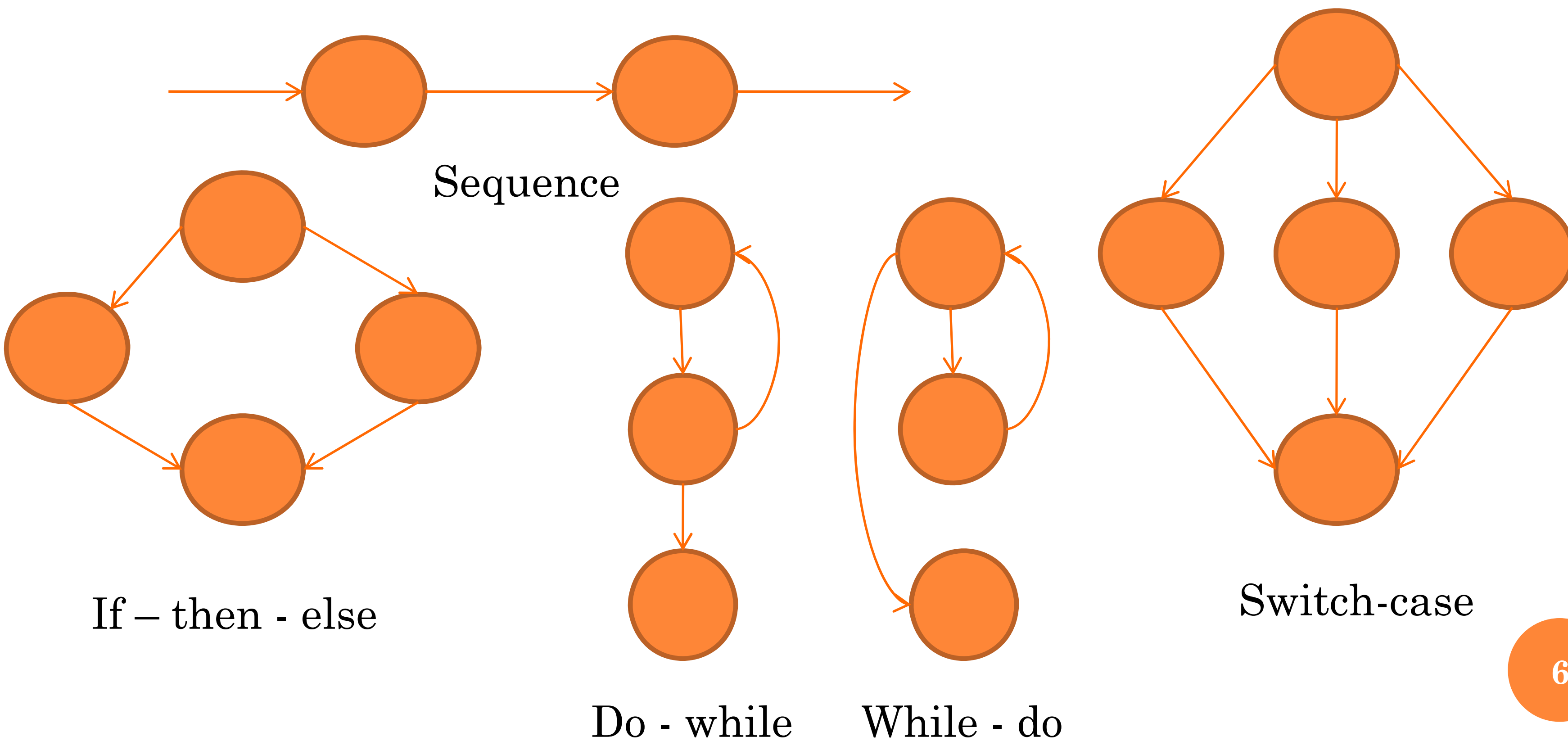
- This technique is quite popular for its simplicity and effectiveness
- The different execution paths of the program are identified
- These paths represent some sequence of statements
- The test cases are used to run those programming statements identified by different paths

Control Flow Based Testing

- Based on control flow of the program
- Initially control flow graph is constructed
- It is a directed graph consisting of a set of vertices and edges
  - **Node:** represents one or more procedural statements
  - **Edges:** represents the flow of control in a program
  - **Decision node:** a node with more than one arrow leaving is called decision node
  - **Junction node:** a node with more than one arrow entering is called junction node
  - **Regions:** areas bounded by edges and nodes are called regions

- **Start node:** the node whose first statement is the start statement of the program
- **Exit node:** the node whose last statement is an exit statement
- **A path** is a finite sequence of nodes $(n_1, n_2, \ldots, n_k)$ where k>1, such that there is an edge between the adjacent nodes
- **Complete path:** a path whose first node is the start node and the last node is the exit node

- Sequential statements having no conditions or loops can be merged in a single node



Control Flow Criteria

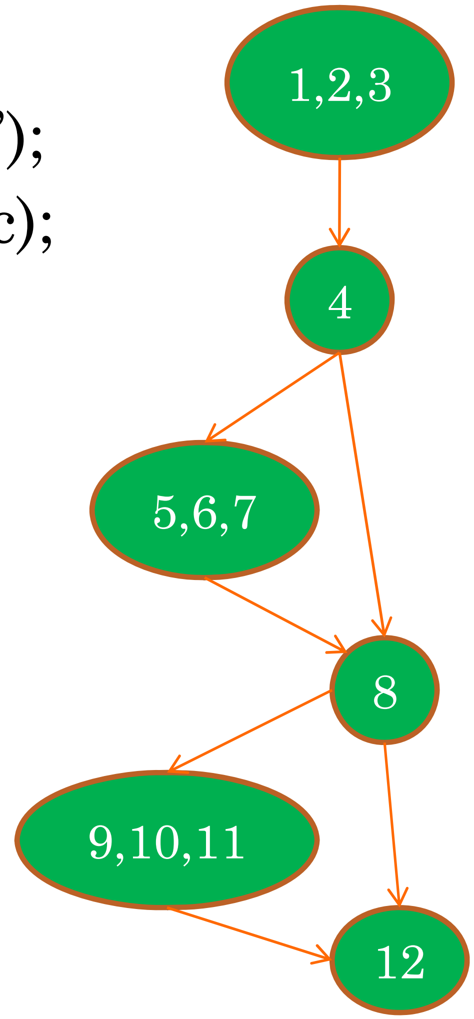
- Statement coverage
- Branch coverage
- Condition coverage
- Path coverage

Statement Coverage

- Each *statement* of the program be executed atleast once
  - Also known as *all nodes criteria*
  - 100% statement coverage

EXAMPLE

```
1.  int a,b,c,x=0,y=0;
2.  printf("Enter three numbers:");
3.  scanf("%d %d %d", &a, &b, &c);
4.  if ((a>b)&&(a>c))
5.  {
6.       x=a*a+b*b;
7.   }
8.   if(b>c)
9.   {
10.      y=a*a-b*b;
11.   }
12.   printf("x=%d, y=%d",x,y);
```



Test Cases

- What are the test cases for satisfying *statement coverage* criteria?
- If the inputs are a=9, b=8, c=7 all statements are executed

BRANCH COVERAGE

- Each *branch/edge* in the control flow graph should be executed atleast once
- Each flow of control (decision) should be evaluated to both *true* and *false*
- Testing based on branch coverage is called as branch testing
- Branch coverage implies statement coverage

- In previous example, branch corresponds to all the true conditions are evaluated
- In this case, branch corresponding to false conditions are also considered
- What are the test cases?
  - a=9, b=8, c=7
  - a=7, b=8, c=9
- These two test cases are sufficient to cover 100% branch coverage

Condition Coverage

- Better than the branch coverage
  - As branch coverage can be achieved without testing every condition
  - For a composite condition of n components, 2^n test cases are required
- In the previous example, line 4 has two conditions – (a>b) && (a>c)
- There are 4 possibilities-

Both are true	a=9,b=8,c=7
First is true, second is false	a=9,b=8,c=10
First is false, second is true	a=7,b=8, c=5
Both are false	a=7,b=8, c=9

Path Coverage

- All possible paths in the control flow graph be executed atleast once
- Also known as path testing
- If the program contains loop then the graph may contain infinite no. of possible paths
- Path coverage implies branch coverage

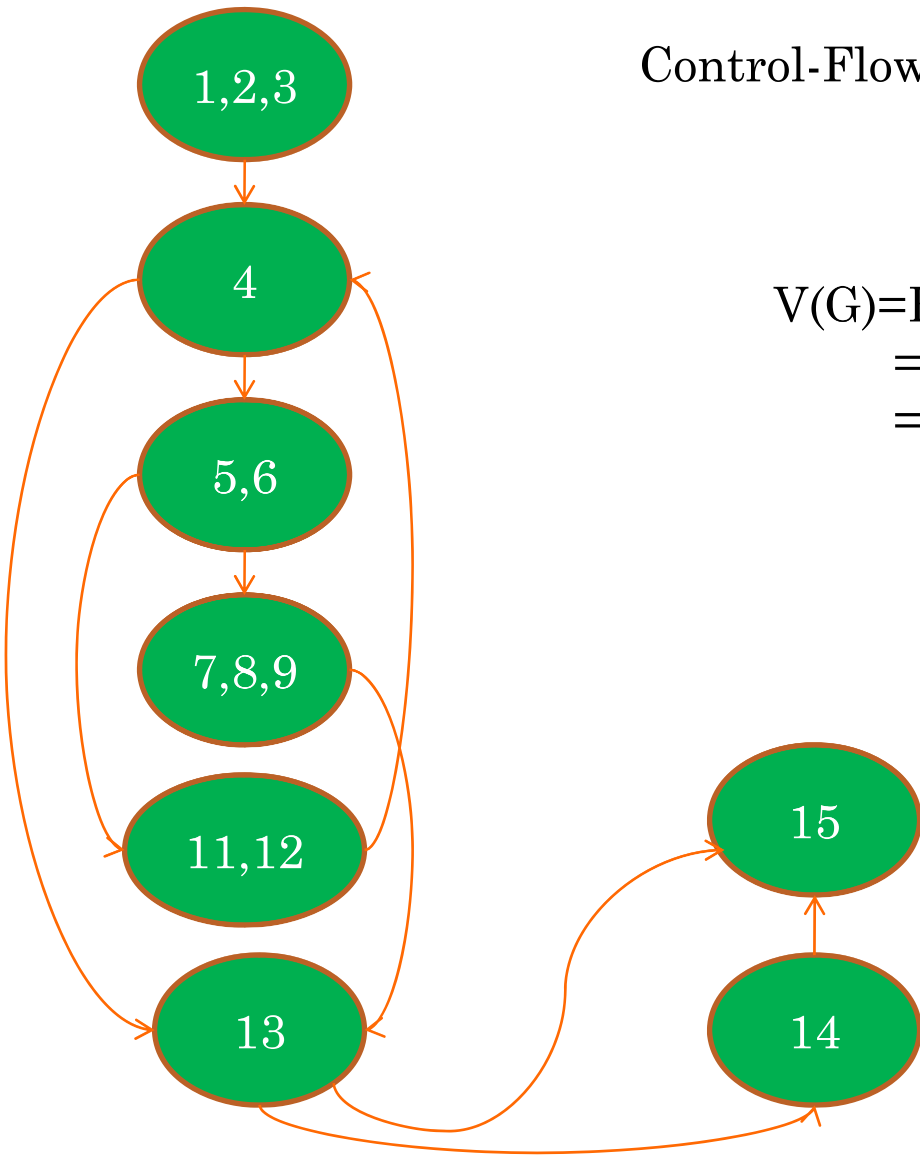
- Additional criteria required to select limited no. of paths
  - Select set of paths that ensure branch coverage condition
  - Two paths are considered to be same if they differ only in their subpaths caused due to the loops
  - Cyclomatic complexity

McCabe's Cyclomatic Metric

- Provides a practical way of determining the maximum no. of linearly independent paths
- Given a control flow graph G, cyclomatic complexity V(G) can be computed as
  1. V(G)=E-N+2 where
     - E is the no. of edges in G
     - N is the no. of nodes in G
  2. V(G)= bounded region + 1
  - Intuitively, no. of bounded regions increases with the no. of decision nodes and loops

EXAMPLE

```
printf("enter a no.");
scanf("%d", &number);
index = 2;
while (index <= number - 1)
{
    if (number % index == 0)
    {
        printf("Not a prime number");
        break;
    }
    index++;
}
if(index == number)
    printf("Prime number");
}
```



Control-Flow-Graph

V(G)=E-N+2
= 10-8+2
=4

Linearly Independent Path

- Cyclomatic complexity of a graph is equal to the maximum number of linearly independent paths in the graph
- Linearly independent path
  - A path that introduces atleast one new edge that has not been traversed before
- To cover all the edges it will never require more than the Cyclomatic Number of paths

- In control flow based testing
  - Various paths of the program are identified
  - Test cases are designed to execute those paths

```
int main()
{
        int a, b, c;
        a=b+c;
        printf("%d", a);
        return 0;
}
```

If the program is executed then an unexpected value may be printed
Here, we assume that all values are not initially assigned to zero by default

DATA FLOW BASED TESTING

- Data flow based testing
  - helps in minimizing such mistakes
- It is based on the variables, their usage and definitions (assignment)
- Here the main points of concern are
  - Statements where variables receive values (**definition**)
  - Statements where variables are used (**referenced**)

- Initially, a **def/use** graph is constructed for the program
- A statement in a node has a variable occurrences in it
- Types of variable occurrences
- **def**:
  - Represents the definition of a variable
  - The variable must be on the LHS of an assignment operation
  - Read statement such as **read** $\mathbf{x_1, \ldots, x_n}$ contains defs of $x_1,\ldots,x_n$

- **c-use**:
  - Represents computations use of a variable
  - In an assignment statement, variables on the RHS
  - Write statement such as **print** $\mathbf{x_1, \ldots, x_n}$ contains c-use of $\mathbf{x_1, \ldots, x_n}$
- **p-use**:
  - Represents predicate use
  - Occurrences of the variables in a predicate
  - Used for transfer of control

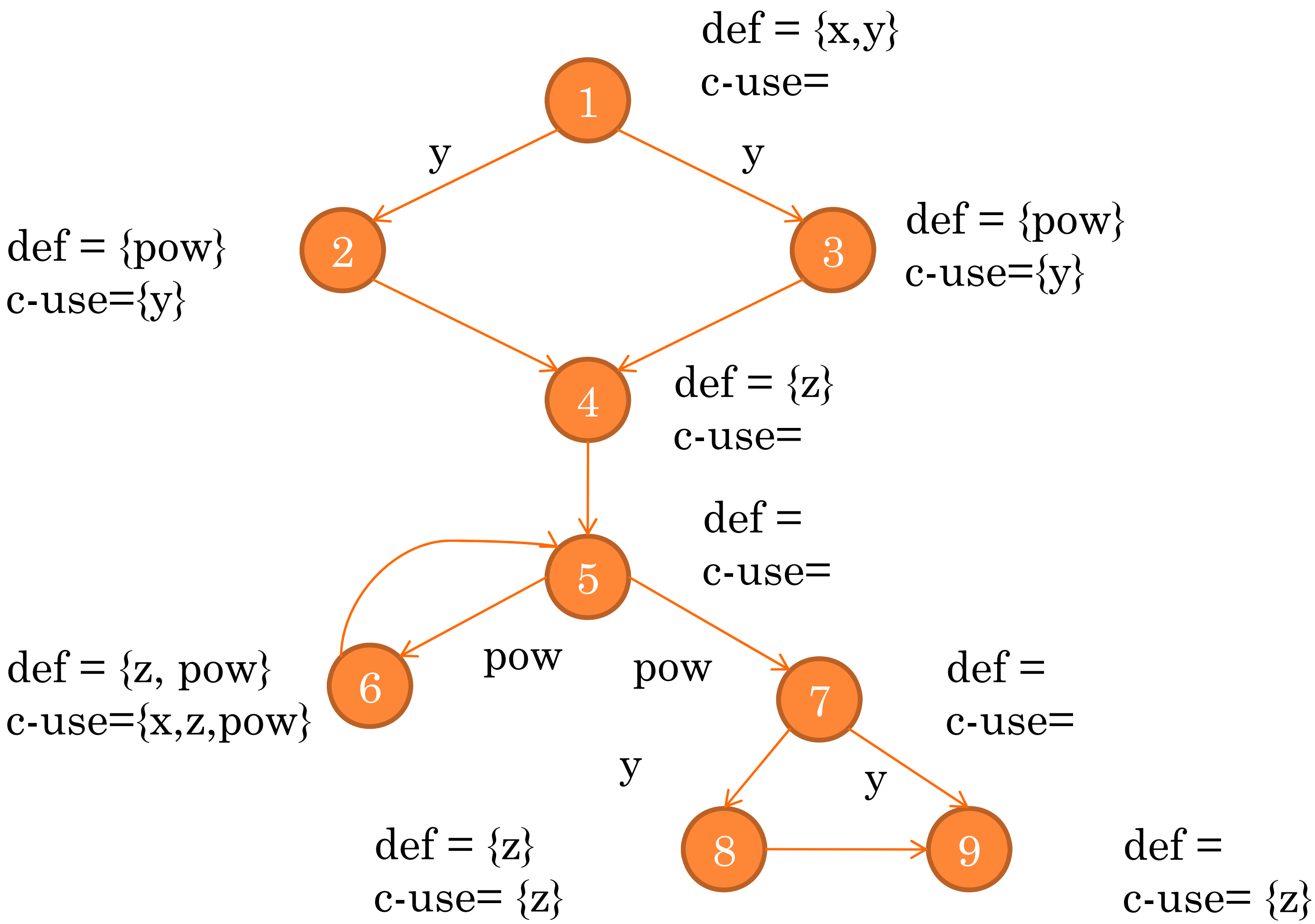
- A *def* which is only within the node in which *def* occurs is of little importance
- **global c-use**: a *c-use* of a variable x is a *global c-use,* provided there is no def of x preceding the c-use within the block in which it occurs
  - With each node *i*, all the global ***c-use*** is associated
  - The ***p-use*** is associated with the edges
  - **def-clear path** w.r.t. a variable *x* if there is no def of *x* in the nodes in the path from node *i* to *j*

- Construction of ***def/use*** graph
  - By associating sets of variables with edges and nodes in the control flow graph
  - For a node *i*, the set ***def(i)*** is the set of variables for which there is a global def in the node *i*
  - The set ***c-use(i)*** is the set of variables for which there is a global *c-use* in the node *i*
  - For any edge *(i,j)*, the set ***p-use(i,j)*** is the set of variables for which there is a *p-use* for the edge *(i,j)*

EXAMPLE

```
scanf(x,y); if (y<0)
        pow = 0 – y;
  else pow = y;
z = 1.0;
while (pow != 0)
  {z = z*x; pow = pow – 1;}
if (y < 0)
   z = 1.0/z;
print(z);
```

Def/use Graph



The set of nodes such that each node has x in its c-use

The set of edges such that each edge *x* has in its p-use

(var, node)	dcu	dpu
(x,1)	{6}	φ
(y,1)	{2,3}	{(1,2),(1,3),(7,8),(7,9)}
(pow,2)	{6}	{(5,6),(5,7)}
(pow,3)	{6}	{(5,6),(5,7)}
(z,4)	{6,8,9}	φ
(z,6)	{6,8,9}	φ
(pow,6)	{6}	{(5,6),(5,7)}
(z,8)	{9}	φ

- Now a family of test case selection criteria can be used
- Let G be a def/use graph for a program
- Let P be a set of complete paths (representing the complete execution) of G
- Now a test case selection criterion defines the contents of P

ALL-DEFS CRITERIA

- P satisfies *all-defs* criteria
  - For every node *i* in *G* and every *x* in *def(i), P* includes a def-clear path w.r.t. *x* to some member of *dcu(x,i)* or *dpu(x,i)*
  - So the criterion says that for the def of every variable, one of its uses must be included in a path
  - During this testing the definition of all the variables is tested

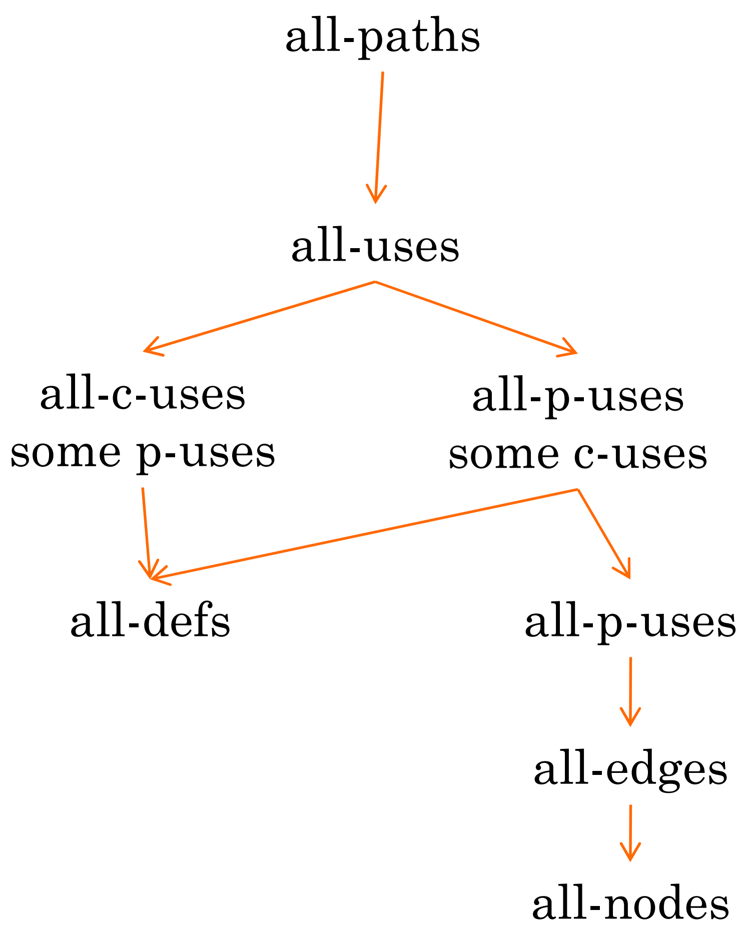
ALL-EDGES CRITERION

- P satisfies *all-edges* criterion
  - Each edge must be traversed during testing
  - E.g. (1;2;4;5;6;5;7;8;9), (1;3;4;5;7;9)
  - To execute the selected paths, the following testcases will suffice
    - (x=3, y=1) and (x=3, y=-1)
  - Can (1;2;4;5;6;5;7;9) be such a path?
    - This path not feasible

ALL-USES CRITERION

- P satisfies *all-uses* criterion
  - all *c-use* and *p-use*s must be executed
  - The previous testcases do not satisfy this criterion
  - Some path needed to use the def of node 6 (*z* and *pow*)
  - Def of *pow* in node 6 is only used by visiting node 6 again
  - For def of *x* in node *i* check the *dcu(x,i)* and *dpu(x,i)*
  - Construct the paths accordingly
  - The following paths will satisfy the criterion
    - (1;2;4;5;6;5;6;5;7;8;9), (1;3;4;5;6;5;7;9)

Relationship between different criteria



C1 → C2 represents set of test cases that satisfy criterion C1 also satisfies criterion C2

Mutation Testing

- Another structural testing technique
- In control-flow and data-flow based testing, the focus on the paths
- Mutation testing is not a path based approach
- It is often aimed at assessing or improving the adequacy of test suit
- Mutation testing can be done after initial testing is complete

- The idea behind mutation testing
  - Make some arbitrary changes in the program at a time
  - The changes are made to capture the most likely faults in some form
  - Mutants should produce a different output from the original program
  - The set of test cases is developed such that
    - It can distinguish between the original program and the mutants

Kinds of Mutation

- Value Mutation
  - Changing the values of constants or parameters
  - E.g. changing the loop bounds
- Decision Mutation
  - Modifying conditions
  - E.g. replacing > by >=
- Statement Mutation
  - Deleting, swapping statements
  - Changing operations in arithmetic expressions
  - E.g. omitting increment operator in a while loop

First order mutants

- Mutation operators that make exactly one syntactic change in the program
- Consider the expression
  - a=b*(c-d)
- Now consider a mutation operator that replaces an arithmetic operator with another one from the set {+, -, *, /}
- Then this particular mutation operator will produce 6 mutants

Example1 of Mutants

M1:

```
r=1;
for(i=1; i<=3;
++i) {
    if(a[i]>a[r])
      r=i;
}
```

M2:

```
r=1;
for(i=2; i<=3;
++i) {
    if(i>a[r])
      r=i;
}
```

P:

```
r=1;
for(i=2; i<=3;
++i) {
    if(a[i]>a[r])
      r=i;
}
```

M3:

```
r=1;
for(i=2; i<=3;
++i) {
    if(a[i]>=a[r])
      r=i;
}
```

M4:

```
r=1;
for(i=2; i<=3;
++i) {
    if(a[r]>a[r])
      r=i;
}
```

Example2 of Mutants

```
P:
int index = 0;
while(…)
{
   …;
   index++;
   if(index==10)
   break;
}
```

```
M1:
int index = 0;
while(…)
{
   …;
   index++;
   if(index>=10)
   break;
}
```

P and M1 will behave similarly for different testcases
Here, M1 is an ***equivalent mutant*** of the original program P

Steps for mutation testing

- Generate mutants M_1, M_2, …, M_N for P
- For each testcase in T, execute each mutant M_i and P
- The mutants that can be distinguished → *dead* mutants
  - Let **D** be no. of dead mutants
- The mutants that cannot be distinguished → *live* mutants
- Determine the mutants that will produce the same output as P → *equivalent* mutants
  - Let **E** be the no. of equivalent mutants
- The mutation score is **D/(N-E)**
- Add more testcases to T and continue testing until the mutation score is 1

Considering the following test data selection for example 1

	a[1]	a[2]	a[3]
TD1	1	2	3
TD2	1	2	1
TD3	3	1	2

Applying these test data to P and M1, …, M4 and obtain the value of variable r (treated as o/p)

	P	M1	M2	M3	M4	Killed Mutants
TD1	3	3	3	3	1	M4
TD2	2	2	3	2	1	M2 & M4
TD3	1	1	1	1	1	None

TD1,TD2, TD3 cannot kill M1 and M3

Can you find any other testcases that can kill M1 and M3?

Constructing test cases

- Suppose a mutant M_l is generated by changing at line l of the original program P
- Now what would be the property of the test case t to kill this mutant?
  - t should force the execution of the statement at line l
  - Once the statement at line l executes M_l and P should reach to different states
  - Finally t should be such that when M_l and P terminate, their states are different

- If no errors is detected and mutation score reaches 1
  - Then the testing is considered to be adequate by the mutation testing criteria
- The mutation score is also used to measure the Residual Defect Density (RDD) in the program
  - RDD refers to defects remaining in the code

Problem of Mutation Testing

- It is related to performance issue
- The no. of mutants that can be generated by first order mutation operation may be quite large
- These many programs have to be compiled and executed on the selected test case set
- This requires an enormous amount of effort

- For example: A program with 950 line where mutation operator can be applied
- A total of about 9,00,000 mutants may be produced
- The testing of this may take 70,000 hours of time on a Sun SPARC station
- Also the tester will have to spend time to analyze whether there are any equivalent mutants