White-Box Testing

Spring, 2023

Yi Xiang xiangyi@scut.edu.cn

Contents

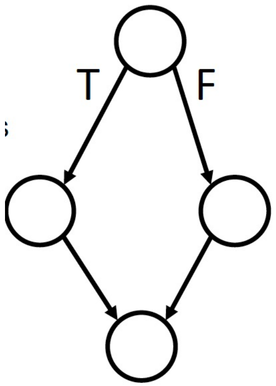
- Control Flow Testing
  - Statement Coverage
  - Decision Coverage
  - Condition Coverage
  - Decision Condition Coverage
  - Condition Combination Coverage

  - Path Coverage
  - Basis Path Testing

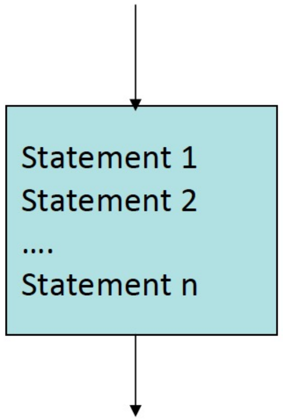
1. Control Flow Graphs (CFGs)

Being able to create a Control Flow Graphs is essential for path testing techniques.

- Directed graph G(V, E)
  - V is set of vertices
  - E is set of edges, E = V*V

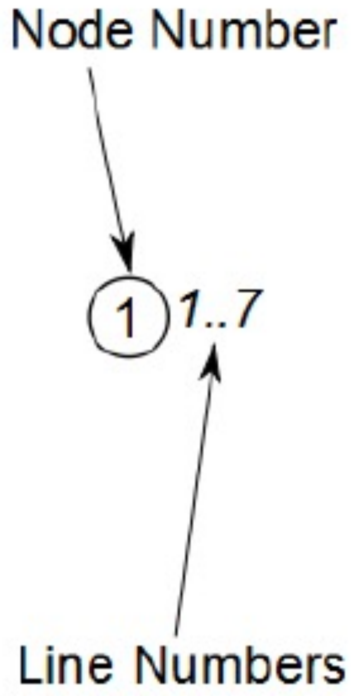


- Represent the flow of control
  - Each node represents one or more statements
  - Each edge represents a 'jump' or 'branch'
  - Two exits=a decision (True or False)



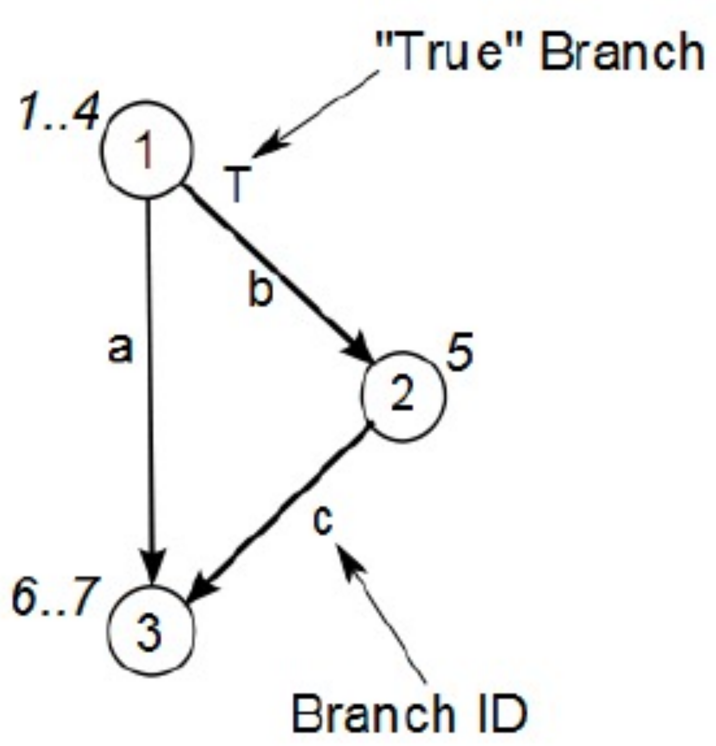
Control Flow Graphs for Sequence

```
1 int f()
2 {
3   int x,y;
4   x = 10;
5   y = x+3;
6   return y;
7 }
```



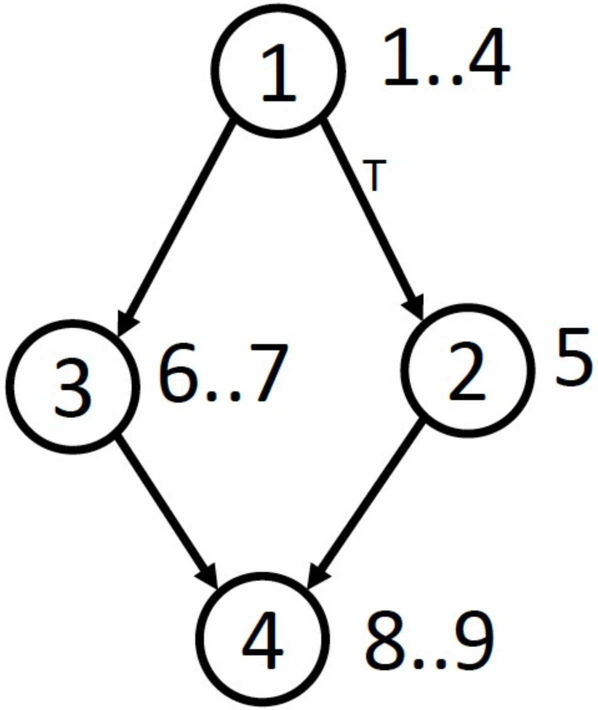
CFG for Selection (if-then)

```
int f(int a)
{
  int x=0;
  if (a>10)
    x=a;
  return x;
}
```



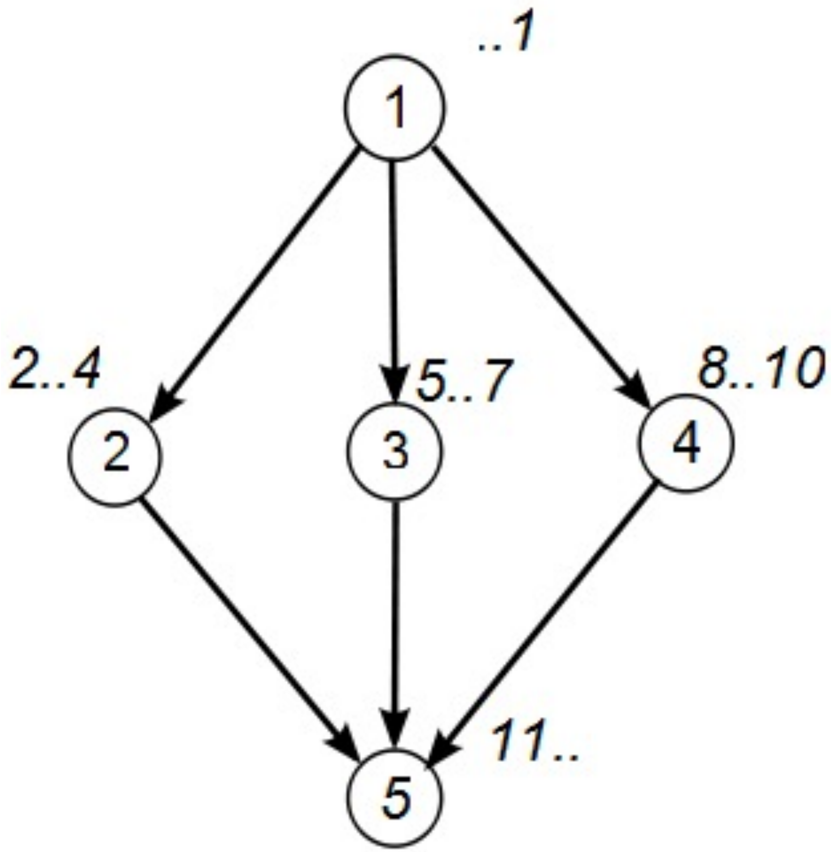
CFG for Selection (if-then-else)

```
int f(int a)
{
  int x;
  if (a>10)
    x=a;
  else
    x=10;
  return x;
}
```



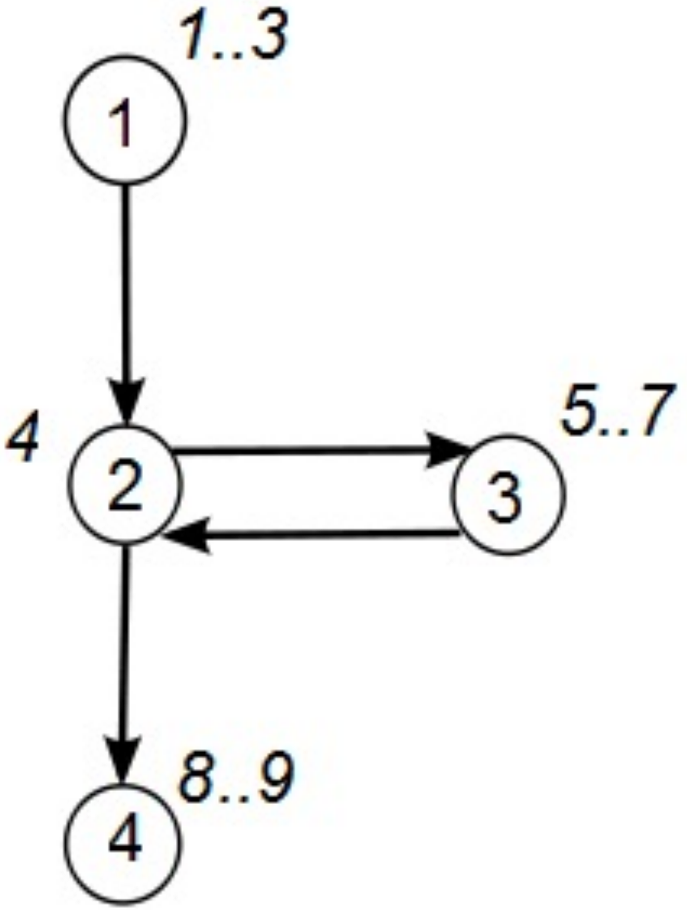
CFG for Selection (switch)

```
...
1  switch(a) {
2     case 0:
3        b=33;
4        break
5     case 1:
6        b=44;
7        break
8     default:
9        ok=false;
10       break;
11 }
...
```



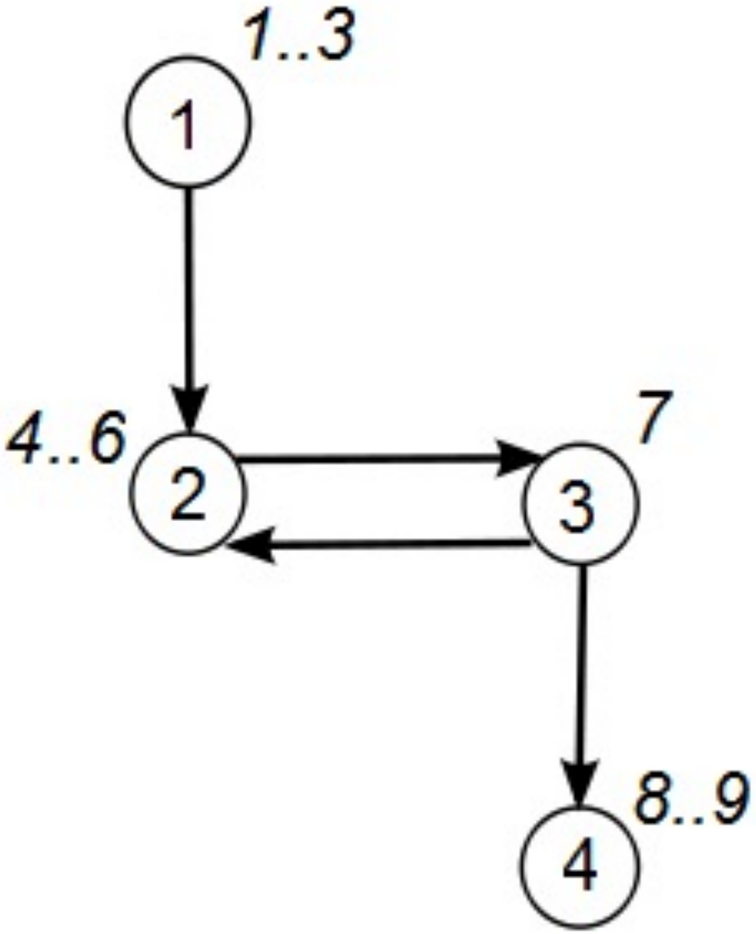
CFG for Iteration (while)

```
int f(int a)
{
  int x=0;
  while (a>0){
    x++;
    a--;
  }
  return x;
}
```



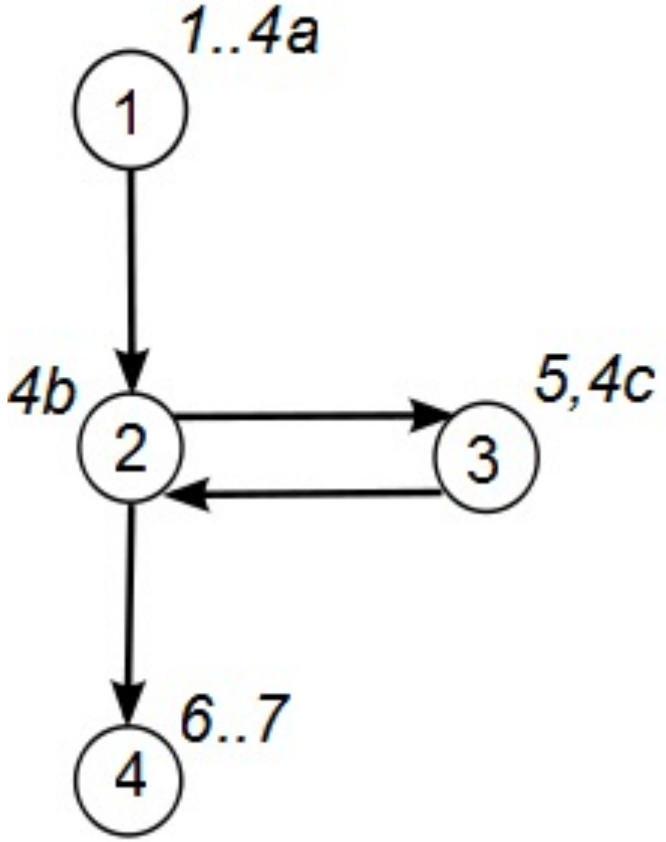
CFG for Iteration (do-while)

```
1 int f(int a)
2 {
3   int x=0;
4   do {
5     x++;
6     a--;
7   } while (a>0);
8   return x;
9 }
```



CFG for Iteration (for)

```
int f(int a)
{
    int x=1;
    for (int i=0; i<a; i++)
        x=x*2;
    return x;
}
```



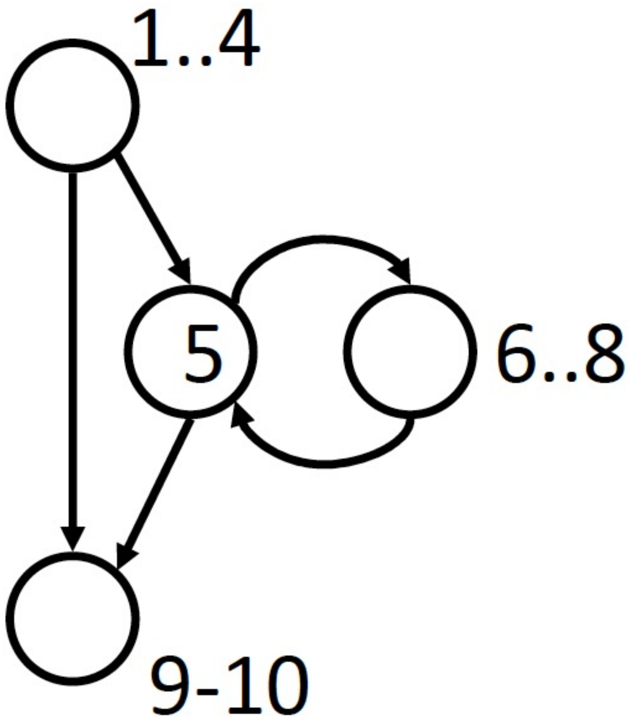
CFG Tips

- Identify all the “jump” points (decisions):
  - if, while, switch/case, for

- Start at the top of the code
- Work your way down to the next jump point
- Create a new node
- For each decision identify the destination node if (a) True and (b) false
- Connect the nodes

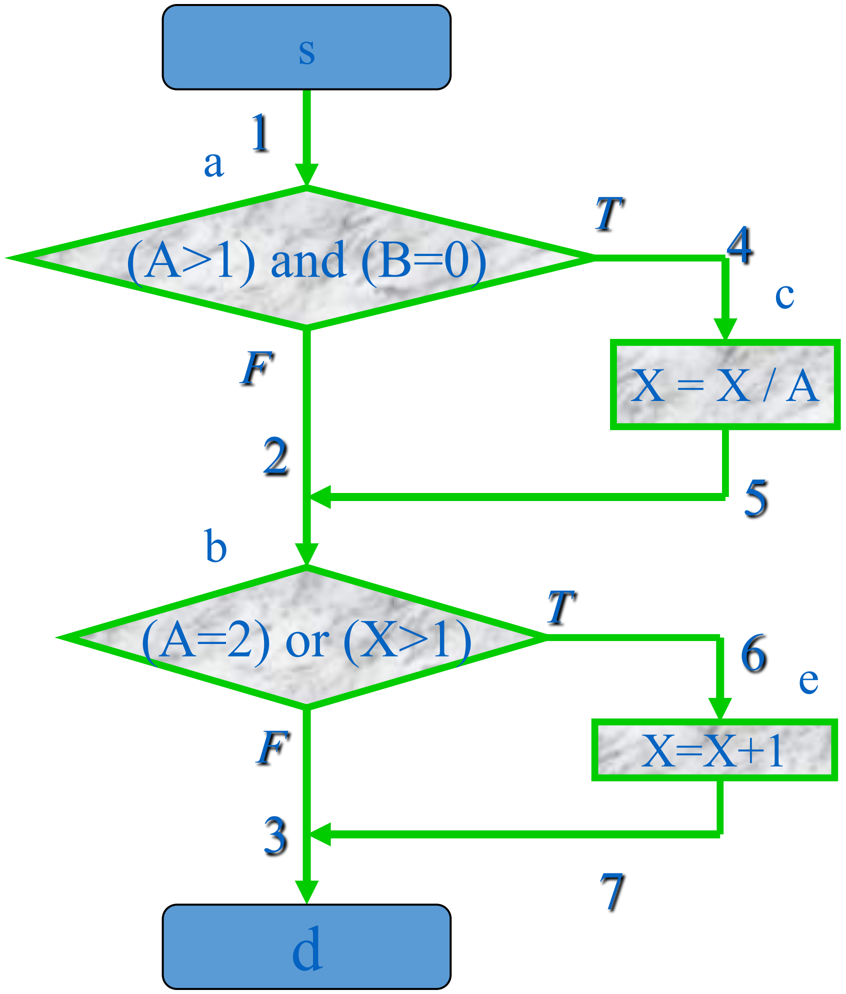
Example Code

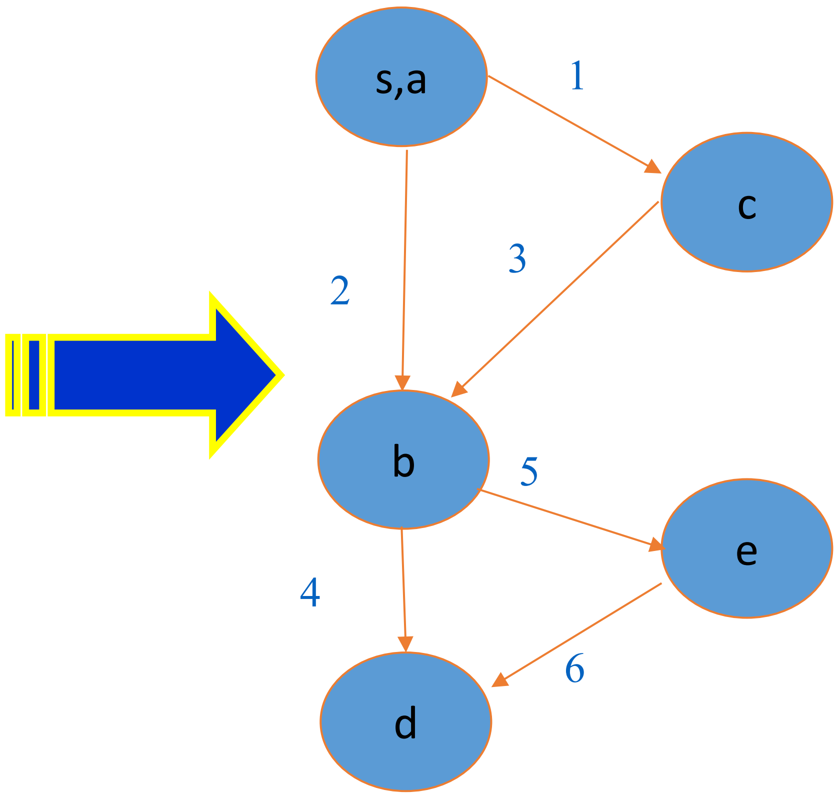
```
1  int multiply(int a,
int b)
2  {
3    int v=0;
4    if ((a>0)&&(b>0))
5      while (a>0) {
6        v=v+b;
7        a--;
8      }
9    return v;
10 }
```



Program Flow Graph vs. Control Flow Graph

- Control flow Graph is a simplified program flow graph
- Only describes the control flow of the program
- Does not show the specific operation of data and the specific conditions of branch or loop





2. Path Coverage

- Generate test data to exercise all the distinct paths in a program. This is called "path coverage"

- Path coverage causes every possible path from entry to exit of the program to be taken during test execution.
- The goal is to achieve 100% coverage of every start-to-finish path in the code.

- A path that makes i iterations through a loop is distinct from a path that makes i+1 iterations through a loop, even if the same nodes are visited in both iterations

Thus, there can be an infinite number of paths is some programs!

Path Coverage

- Need to limit the number of paths: choose equivalence classes of paths

- Two paths are considered equivalent if they differ only in the number of loop iterations, giving two classes of loops:
  - one with 0 iterations
  - one with n iterations ($n > 0$)

- Other equivalence paths can also be chosen if required

Path Expression

The CFG of a program can be described by a regular expression that uses the following operations:

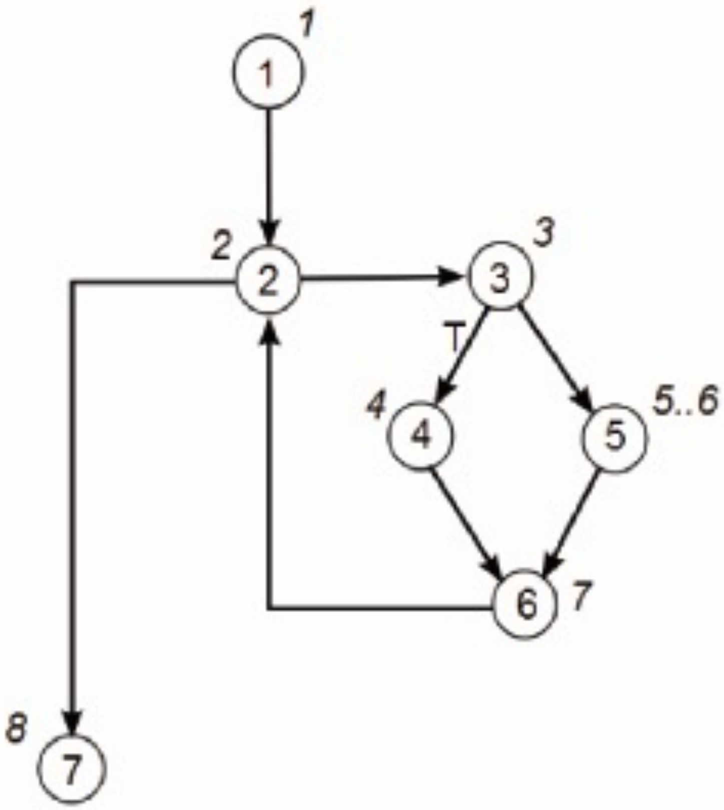
. is the concatenation of a sequence of nodes

+ is a decision in the graph (i.e. an if statement)

* is iteration (0 or more times, e.g. a while statement)

Path Expression - Example

```
1)  i=0;
2) while (i<list.length) {
3)        if (list[i]==target)
4)                match++;
5)        else
6)                mismatch++;
7)        i++;
8) }
```



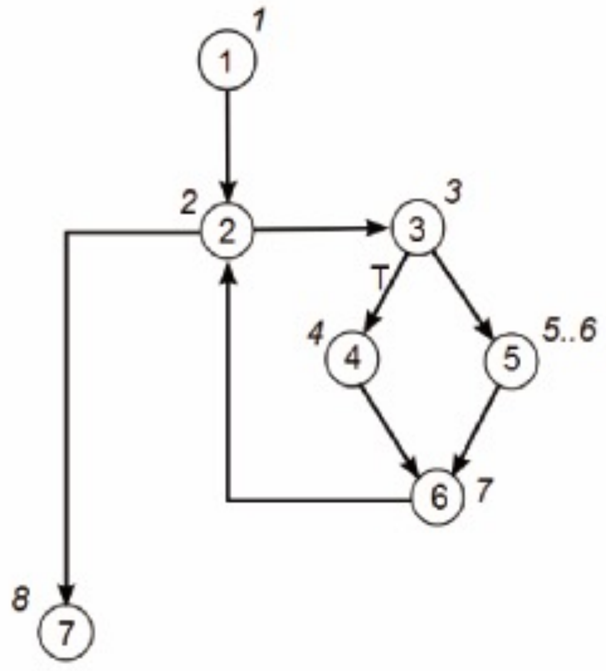
The CFG can be represented by

$$1.2.(3.(4+5).6.2)^*.7$$

Path Expression - Example

- The loop can be simplified by
  - replacing the (expression)* with a (expression+0)
  - where 0 is a null represents a loop with 0 iterations

1.2.(3.(4+5).6.2)*.7



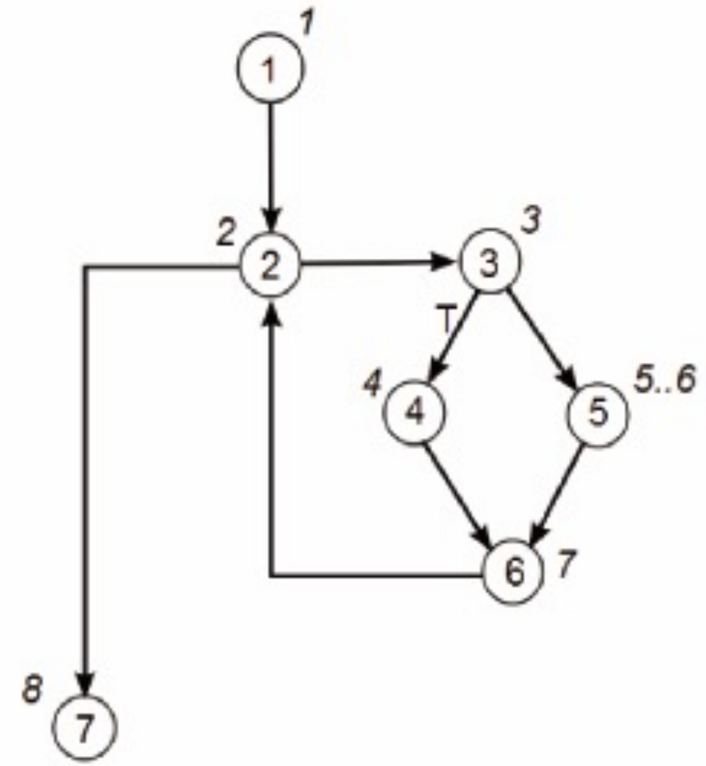
This gives:

1.2.(3.(4+5).6.2+0).7

Expanding gives the paths:

- 1-2-7
- 1-2-3-4-6-2-7
- 1-2-3-5-6-2-7

Path Coverage - Example



1.2.(3.(4+5).6.2+0).7

- Replacing each node number (including the null) by a 1
- Evaluating the expression mathematically (+ becomes addition and . becomes multiplies)

we can work out the total number of paths

$$Paths = 1.1.(1.(1+1).1.1+1).1 = 3$$

Note for "null else" statements where there is an if and no else
the expression (node +0) is used where 0 represents the "null else" decision

Path Coverage - seatsAvailable

```
1   public static boolean seatsAvailable(int freeSeats,
                                         int seatsRequired)
2   {
3      boolean rv=false;
4      if ( (freeSeats>=0) && (seatsRequired>=1)
                           && (seatsRequired<=freeSeats) )
5         rv=true
6      return rv;
7   }
```

Node 1: Lines 1–4
Node 2: Line 5
Node 3: Lines 6, 7

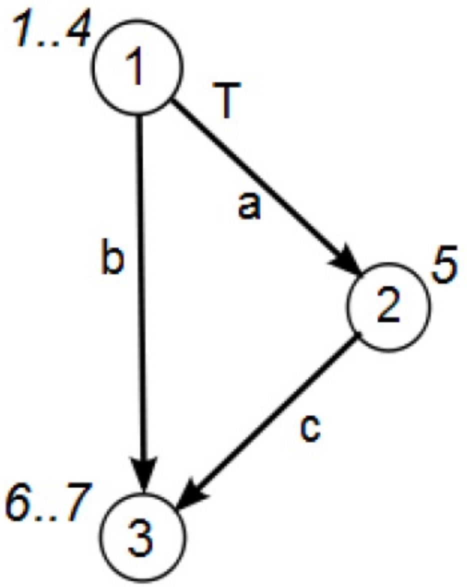


Figure 4.1: CFG for seatsAvailable()

Paths

Examining the control flow graph, two paths can be seen:

(1) 1-3
(2) 1-2-3

Path Coverage - seatsAvailable

If we wish to characterize the program using a Regular Expression we can write:

$$\boxed{1.(0+2).3}$$

Replace all values, including null, by 1 to compute the number of paths through the program.
This gives:

$$\boxed{1.(1+1).1=1.(2).1=2}$$

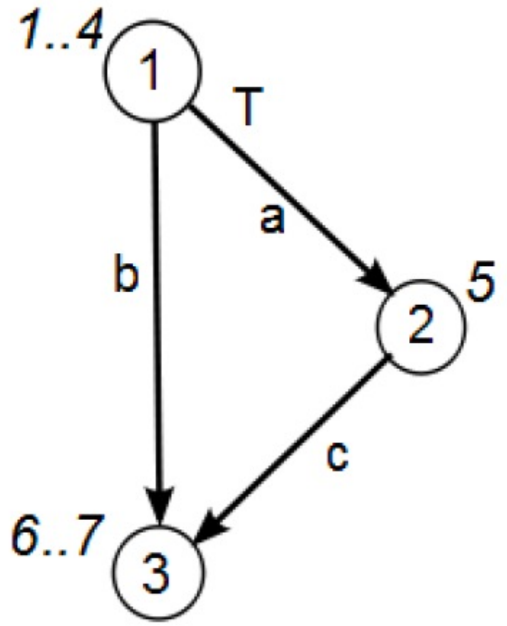


Figure 4.1: CFG for seatsAvailable()

Test Cases and Test Data - seatsAvailable

- Each Path is a Test Case
  1. Path 1
  2. Path 2

Table 4.22: Path Test Cases for seatsAvailable()

Case	Nodes	Test
1	1,3	30
2	1,2,3	31

- Test Data

Table 4.15: BC Tests for seatsAvailable()

ID	Test Cases Covered	Inputs		Expected Output
		freeSeats	seatsRequired	return value
30	a,c	50	25	true
31	b	-50	25	false

- Each path must be tested in a separate test.
- It is straightforward to create tests to cover both paths.
- In this case, the tests will be the same as for Branch testing.
- It must be noted though that this will not always be so.

Test Cases and Test Data - seatsAvailable

Compared with Condition Combination Coverage:

Table 4.20: MCC Test Cases for seatsAvailable()

Case	freeSeats≥0	seatsRequired≥1	seatsRequired≤freeSeats	Test
1	T	T	T	36
2	T	T	F	37
3	T	F	T	38
4	T	F	F	
5	F	T	T	
6	F	T	F	39
7	F	F	T	40
8	F	F	F	41

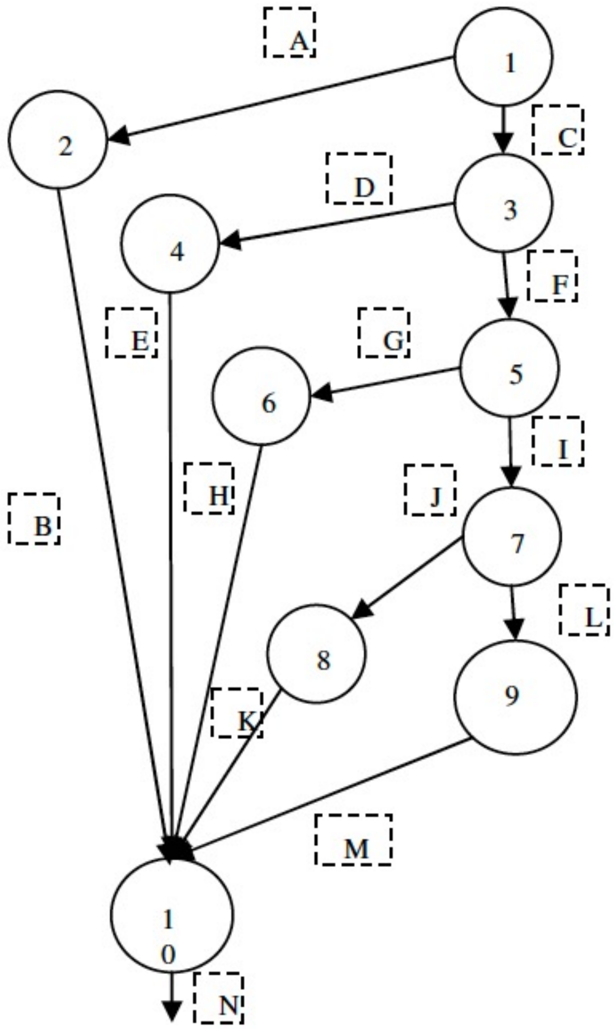
Table 4.21: MCC Tests for seatsAvailable()

ID	Test Cases Covered	Inputs		Expected Output
		freeSeats	seatsRequired	return value
36	1	50	25	true
37	2	50	75	false
38	3	50	-25	false
39	6	-50	25	false
40	7	-50	-75	false
41	8	-50	-25	false

Path Coverage does not explicitly evaluate the conditions in each decision.

Path Coverage – Program Grade

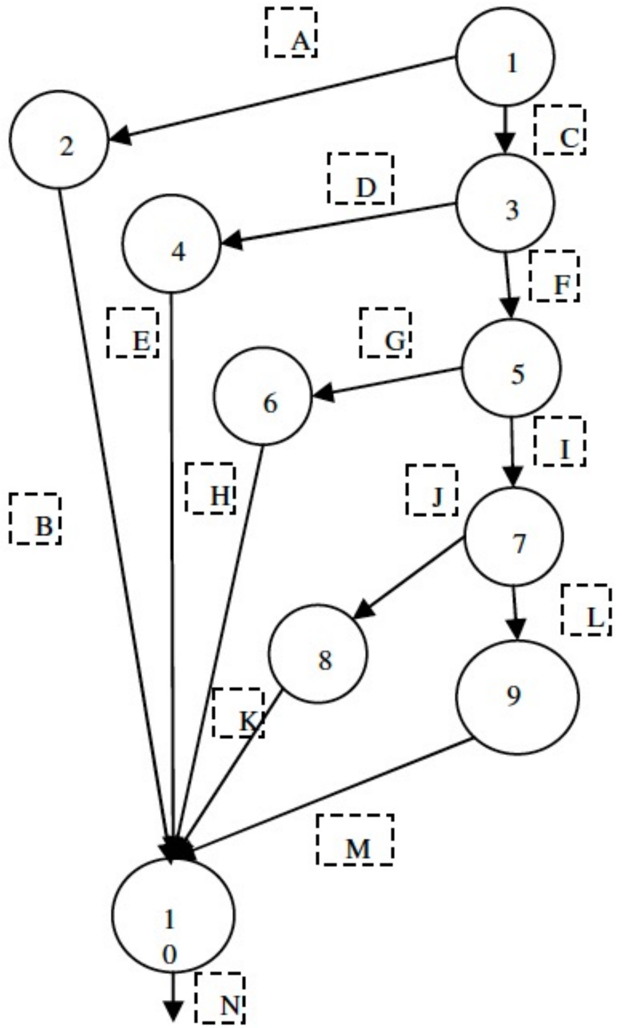
```
public static String Grade (int exam, int course) {
String result="null";
long average;
average = Math.round((exam+course)/2);
if ( (exam<0) || (exam>100) || (course<0) || (course>100)
result="Marks out of range";
else {
if ( (exam<50) || (course<50)) {
result="Fail";
}
else if (exam < 60) {
result="Pass,C";
}
else if ( average >= 70) {
result="Pass,A";
}
else {
result="Pass,B";
}
}
return result;
}
```



Path Coverage – Program Grade

Statement Coverage

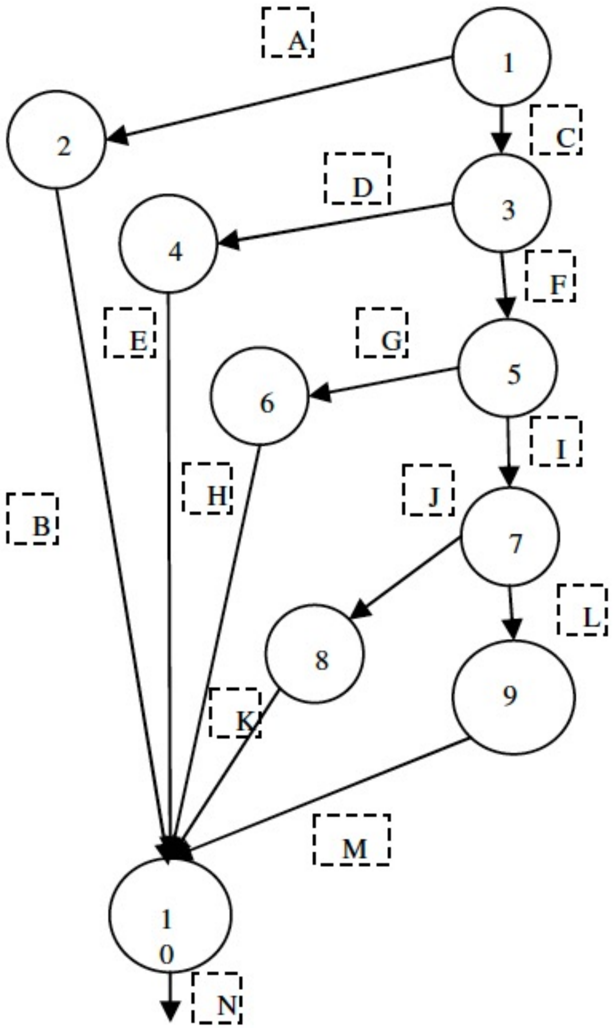
Test No.	Test Cases/Nodes Covered	Inputs		Expected Outputs
		exam	*course*	*Result*
1	1, 2, 10	-1	-1	Marks out of Range
2	1, 3, 4, 10	40	50	Fail
3	1, 3, 5, 6, 10	55	50	Pass, C
4	1, 3, 5, 7, 8, 10	90	50	Pass, A
5	1, 3, 5, 7, 9, 10	80	50	Pass, B



Path Coverage – Program Grade

Decision(Branch) Coverage

Test No.	Test Cases/Nodes Covered	Inputs		Expected Outputs
		exam	*course*	*Result*
1	A, B, N	-1	-1	Marks out of Range
2	C, D, E, N	40	50	Fail
3	C, F, G, H, N	55	50	Pass, C
4	C, F, I, J, K	90	50	Pass, A
5	C, F, I, L, M, N	80	50	Pass, B



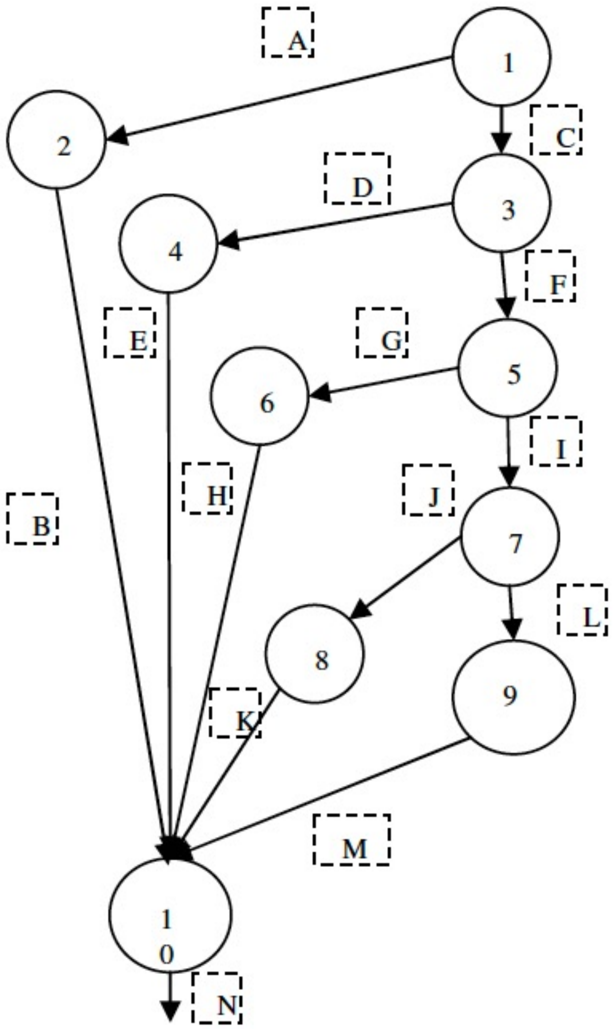
Path Coverage – Program Grade

Path Coverage

Path Expression: 1.(2+3.(4+5.(6+7.(8+9)))).10

Five paths :

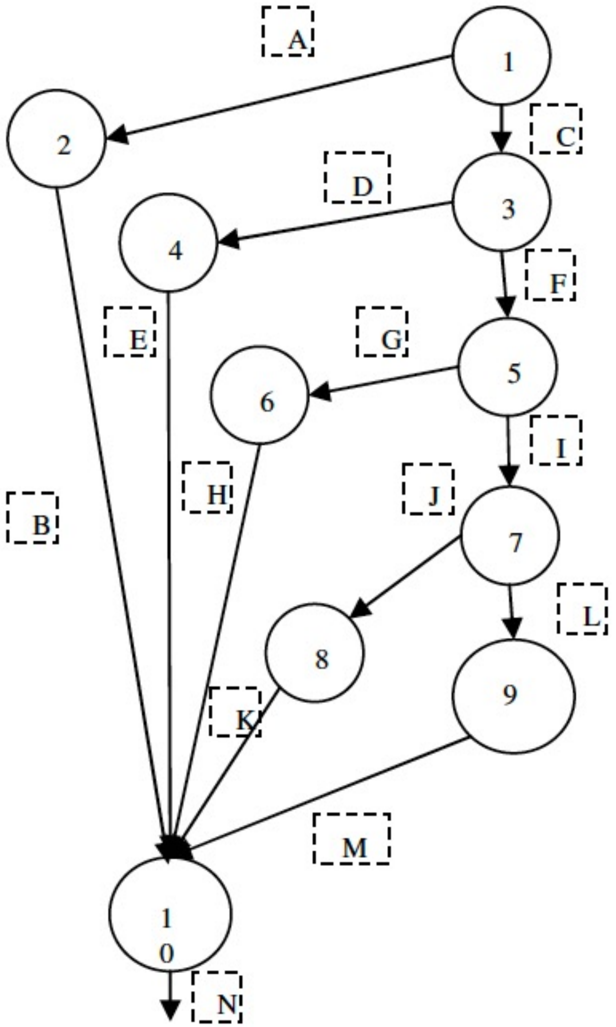
1-2-10
1-3-4-10
1-3-5-6-10
1-3-5-7-8-10
1-3-5-7-9-10



Path Coverage – Program Grade

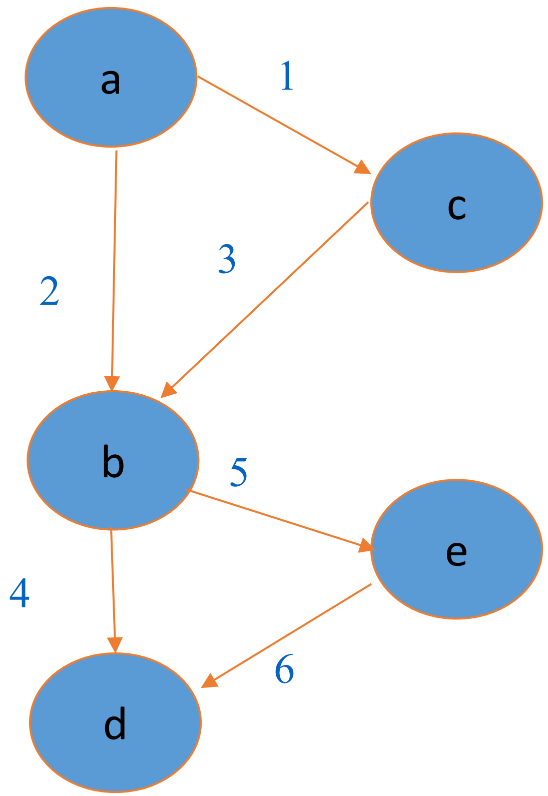
Path Coverage

Test No.	Test Cases/Paths Covered	Inputs		Expected Outputs
		exam	*course*	*Result*
1	1	-1	-1	Marks out of Range
2	2	40	50	Fail
3	3	55	50	Pass, C
4	4	90	50	Pass, A
5	5	80	50	Pass, B



Path Coverage can achieve 100% statement coverage and 100% branch coverage.

Path Coverage – EX.




```
public static float Example(float A,B,X){

    if ( A>1 &&  B==0 )
        X= X / A;
    if  (A==2 || X>1)
        X=X+1;
    return X;

}
```

Path Expression: a.(0+c).b.(0+e).d

Paths:

abd
abed
acbd
acbed

Path Coverage – EX.

Test Cases			Paths	Output
A	B	X		X
1	1	1	abd	1
1	1	2	abed	2
3	0	1	acbd	1/3
2	0	4	acbed	4/3

Compared with Condition Combination coverage:

TestCase			Path	Conditions	Condition Combination	Decisions	Expected Output
A	B	X					
2	0	4	sacbed	T1,T2,T3,T4	1, 5	TT	3
2	1	1	sabed	T1,$\overline{\text{T2}}$,T3,$\overline{\text{T4}}$	2, 6	FT	2
1	0	2	sabed	$\overline{\text{T1}}$,T2,$\overline{\text{T3}}$,T4	3, 7	FT	3
1	1	1	sabd	$\overline{\text{T1}}$,$\overline{\text{T2}}$,$\overline{\text{T3}}$,$\overline{\text{T4}}$	4, 8	FF	1

Path Coverage – Strengths/Weaknesses

- It does create combinations of paths not exercised by other methods
  - Creating and executing tests for all possible paths results in 100% statement coverage and 100% branch coverage.
- However, it can be computationally intensive if the program is complex and many paths are found.
- Also, it does not explicitly evaluate the conditions in each decision.
- If path coverage and condition combination coverage are combined, test cases with stronger fault detection ability can be designed

3. Basis Path Testing

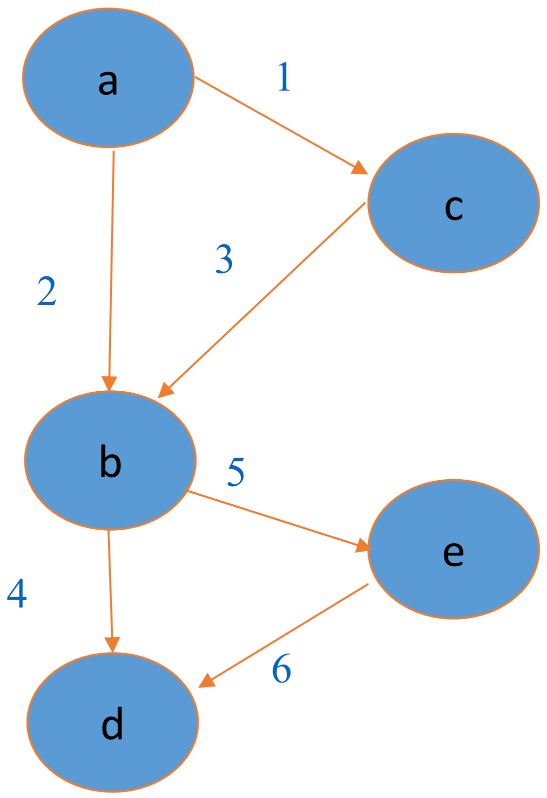
- Basis Path Testing is a White Box Testing method in which test cases are defined based on flows or logical paths that can be taken through the program.

- The objective of basis path testing is to define the number of independent paths, so the number of test cases needed can be defined explicitly to maximize test coverage.

- Basis path testing involves execution of all possible blocks in a program and achieves maximum path coverage with the least number of test cases.

Independent paths

Independent path is defined as a path from entry to exit that has at least one edge which has not been traversed before in any other paths.



Path Expression: a.(0+c).b.(0+e).d

Paths:

abd ✓
abed ✓
acbd ✓
acbed ✗

Number of the independent paths: 3

Steps for Basis Path testing

(1) Draw a control flow graph (to determine different program paths)

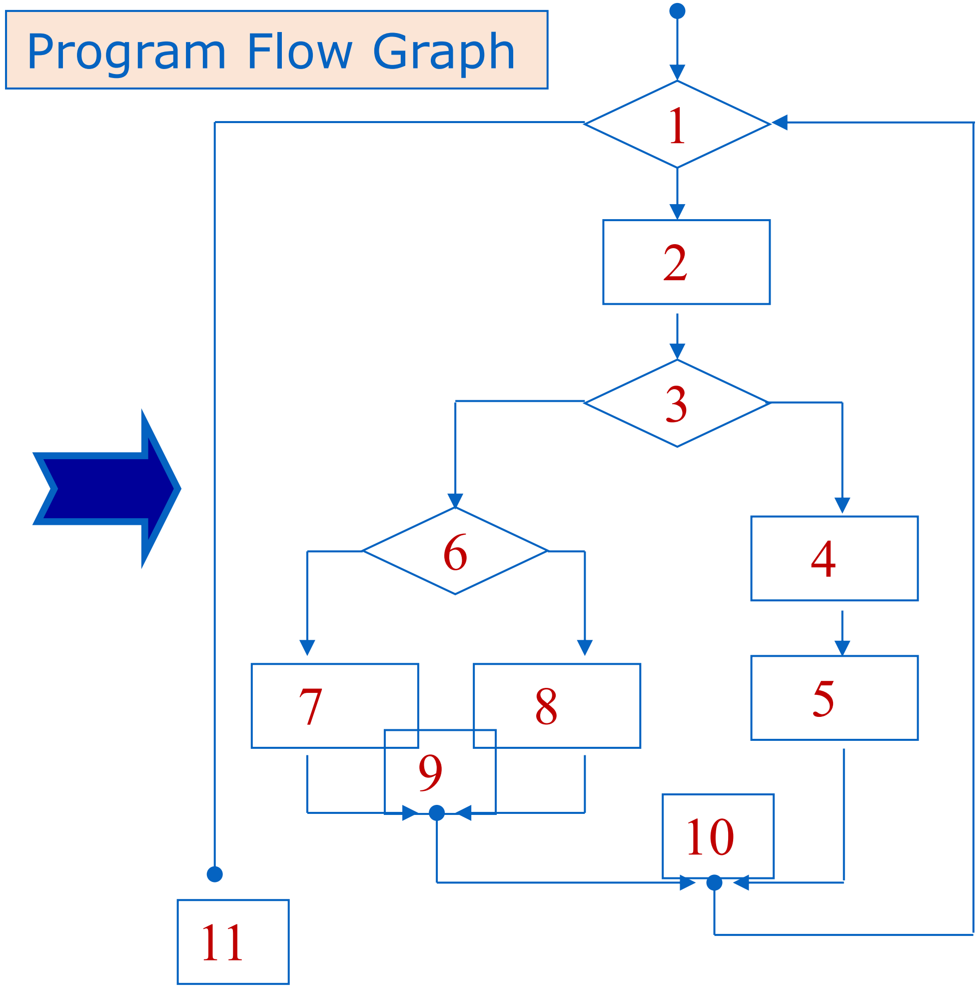
(2) Calculate Cyclomatic complexity
(metrics to determine the number of independent paths)

(3) Find a basis set of paths

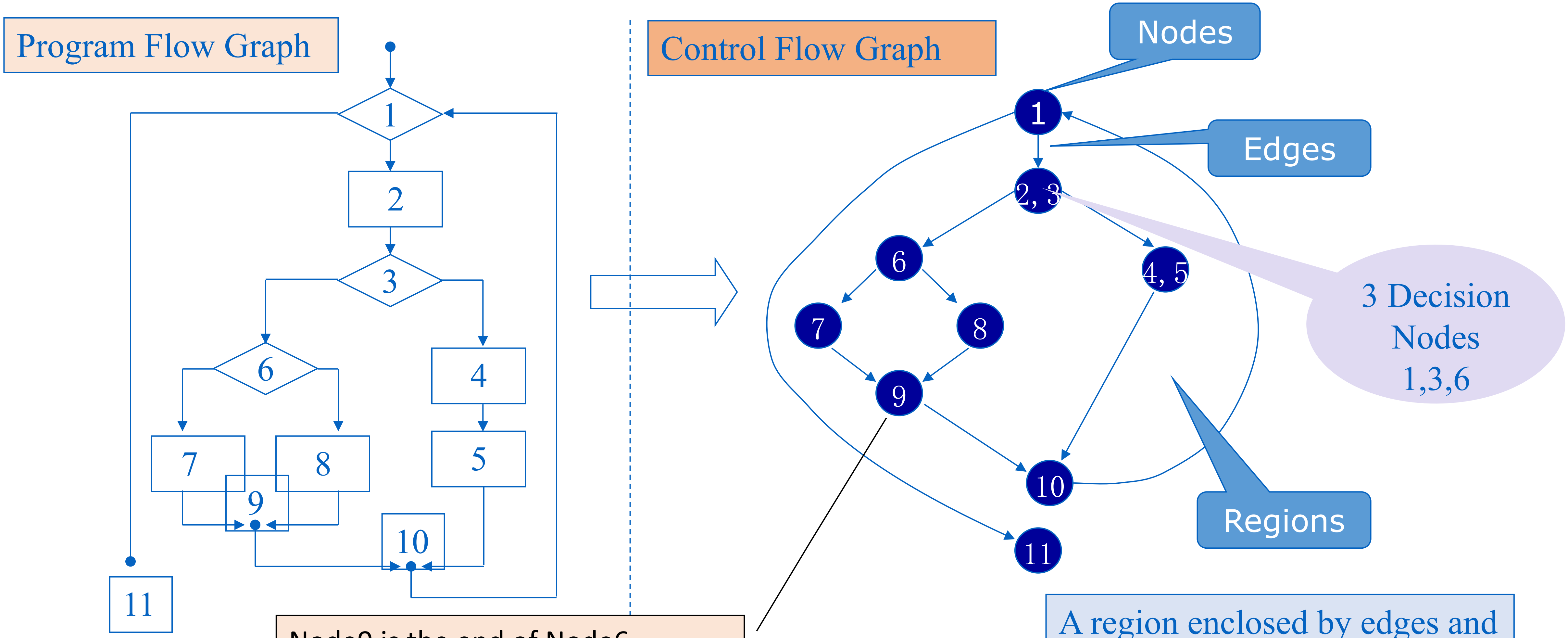
(4) Generate test cases to exercise each path

Step1 ： Draw a control flow graph

```
void Func(int nPosX, int nPosY)
{
   while (nPosX > 0)
   {
             int nSum = nPosX + nPosY;
             if (nSum > 1)
             {
                nPosX--;
              nPosY--;
             }
             else
             {
                if (nSum < -1) nPosX -= 2;
                else nPosX -= 4;
             } // end of if
   }  // end of while
}
```



Step1 : Draw a control flow graph



Node9 is the end of Node6, Node10 /Node3, Node 11/ Node 1

A region enclosed by edges and nodes(including outer region)

Step2: Calculate McCabe's Cyclomatic complexity

- Cyclomatic Complexity is a testing metric developed by Thomas J. McCabe and used for measuring the complexity of a software program.
  - It is a quantitative measure of independent paths in the source code of a software program.
- Question:
  How many paths should be found to cover the basis path set?
- Cyclomatic Complexity provides a basis for determining the upper bound of the basis path set.
  - Cyclomatic Complexity is the maximum number of independent paths
  - Note: The basis path set is not unique.

Basis Path Testing checks each linearly independent path through the program, which means **number of test cases, will be equivalent to the cyclomatic complexity of the program.**

Step2: Calculate McCabe's Cyclomatic complexity

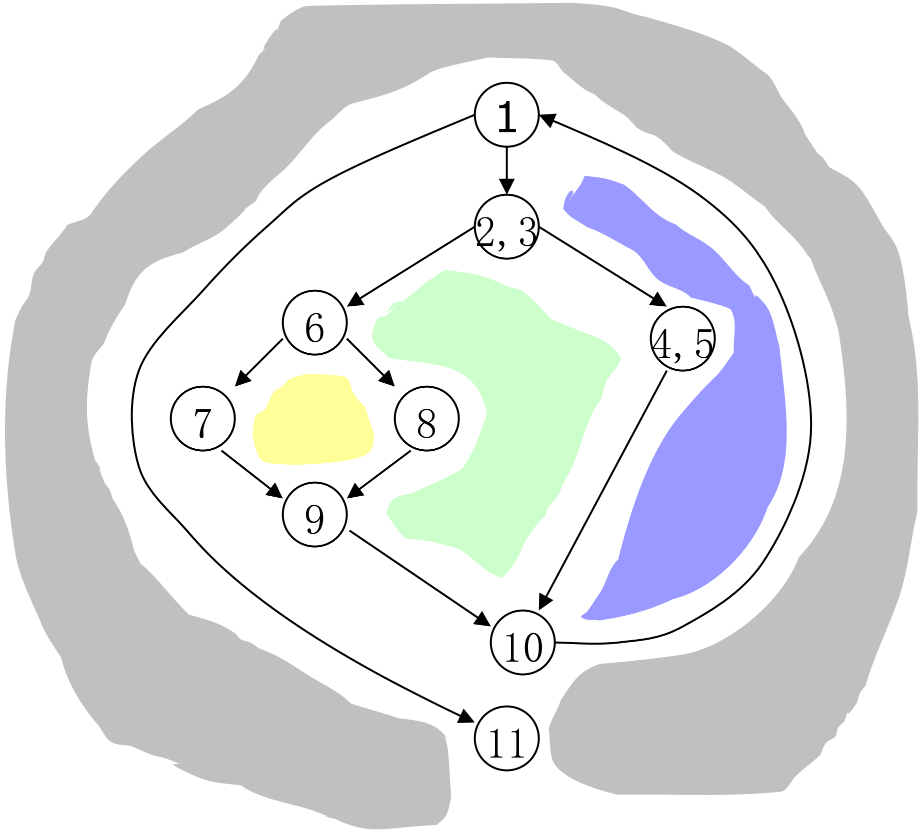
- Three methods to compute Cyclomatic Complexity V(G)

$V(G) = E - N + 2$ E = Number of edges, N =Number of Nodes

$V(G) = P + 1$ P = Number of decision nodes (node that contains condition)

$V(G) = R$ R= Number of regions

Step2: Calculate McCabe's Cyclomatic complexity



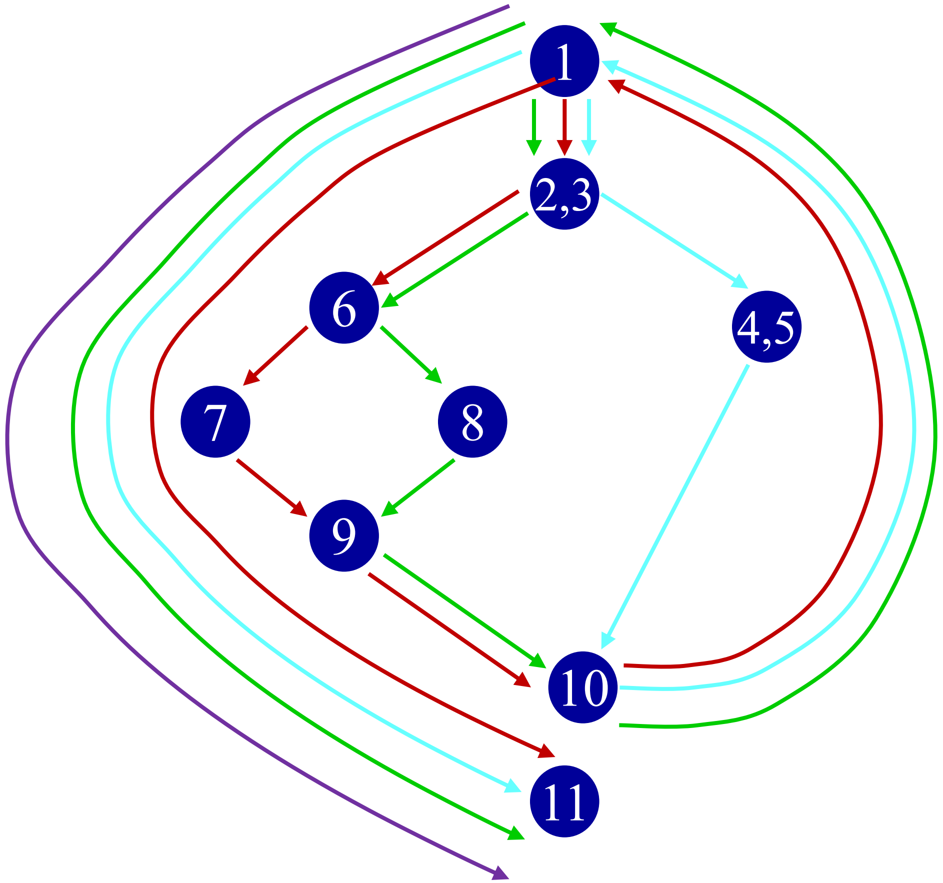
(1) V(G) =R= 4

(2) V(G) =E-N+2 = 11-9+2=4

(3) V(G) =P+1 = 3+1=4

Step3: Find Basis Path Set

Independent path:
A path that moves along at least one new edge from the beginning to the end



Path 1 : 1-11
Path 2 : 1-2-3-4-5-10-1-11
Path 3 : 1-2-3-6-8-9-10-1-11
Path 4 : 1-2-3-6-7-9-10-1-11

To traverse the above path is to execute all statements and all the branches in the program at least once.

Step4: Design Test Cases

Design test cases to ensure the execution of each path in the basis path set.

Input		Paths	Output	
nPosX	nPosY		nPosX	nPosY
-1	1	1 – 11	-1	1
1	1	1 – 2 – 3 – 4 – 5 – 10 – 1 – 11	0	0
1	-3	1 – 2 – 3 – 6 – 8 – 9 – 10 – 1 – 11	-1	-3
1	-1	1 – 2 – 3 – 6 – 7 – 9 – 10 – 1 – 11	-3	-1