White Box Testing: Dynamic Testing

M Saeed Siddik

IIT, University of Dhaka

White Box Testing

- White-box testing techniques are used for testing the module during the development phase or pre-release phase.
- It also called as structural testing or glass box testing; that designs test cases based on the information derived from source code
- There may be portions in the code that are not checked when executing functional test cases, but these will be executed and tested by white-box testing
- Though test cases for black box can be designed earlier than white- box test cases, they cannot be executed until the code is produced and checked using white-box testing techniques.

Primary Goals

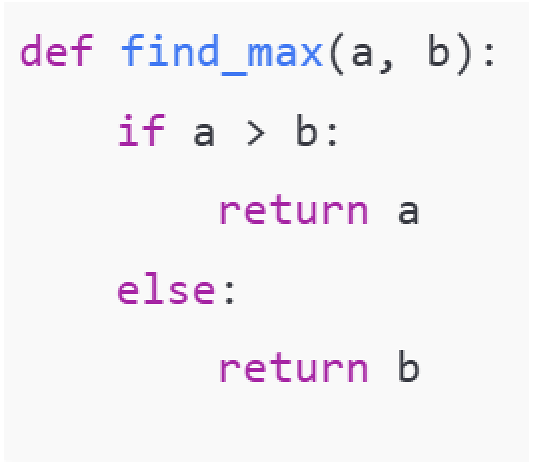
- Validate internal workflows and logic (not just output).
- Ensure code behaves as expected under all scenarios.
- Identify logical bugs, security vulnerabilities etc.

Coverage in White Box Testing

- key metric used to assess the thoroughness of testing
  - Statement Coverage
  - Branch Coverage (Decision Coverage)
  - Condition Coverage
  - Path Coverage
  - Function Coverage
  - Loop Coverage

Statement Coverage

- **Goal:** Execute every line at least once.
- **Test Case:** find_max(5, 3)
  - Executes: line 1, 2, and 3
- **Test Case:** find_max(2, 4)
  - Executes: line 1, 2, and 4

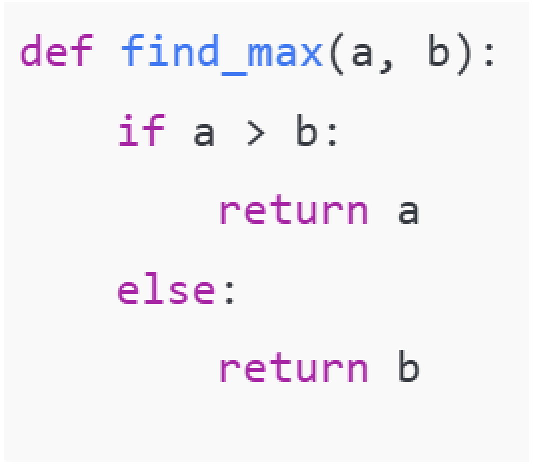


```python
def find_max(a, b):
    if a > b:
        return a
    else:
        return b
```

Branch (Decision) Coverage

- **Goal:** Test both outcomes of every decision (if a > b).
- **Test Case 1:** find_max(5, 3) → True branch (line 3)
- **Test Case 2:** find_max(2, 4) → False branch (line 4)

☐ Covers **both true and false** conditions → **100% branch coverage**.



```python
def find_max(a, b):
    if a > b:
        return a
    else:
        return b
```

Condition Coverage

```
if a > b and b > 0:
```

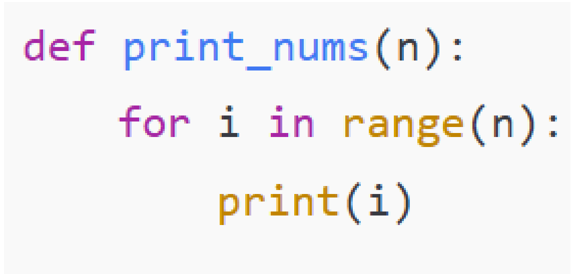
- **Goal:** Test each Boolean sub-condition (a > b, b > 0) as true and false.
- **Test Case 1:** find_max(5, 3) → both true
- **Test Case 2:** find_max(1, 4) → both false
- **Test Case 3:** find_max(5, -1) → a > b true, b > 0 false

Covers all combinations → **100% condition coverage**

Loop Coverage

- **Test Case 1:** print_nums(0) → loop executes 0 times
- **Test Case 2:** print_nums(1) → loop executes once
- **Test Case 3:** print_nums(3) → loop executes multiple times

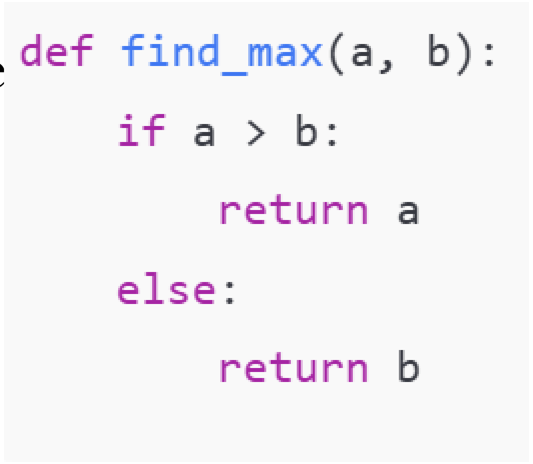
  ☐ Covers all loop behaviors.



```python
def print_nums(n):
    for i in range(n):
        print(i)
```

Path Coverage

- For the original find_max(a, b), there are 2 paths:
- Path 1: if condition true → return a
- Path 2: if condition false → return b
  ☐ Running both test cases from branch coverage achieves **100% path coverage**.



```python
def find_max(a, b):
  if a > b:
    return a
  else:
    return b
```

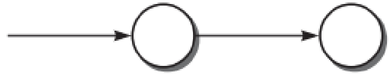
Basis path testing

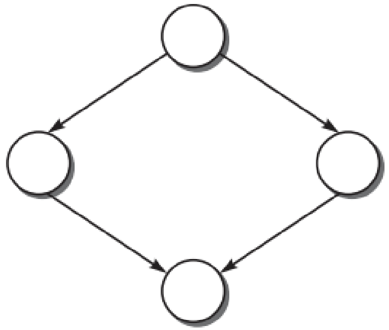
- based on the control structure of the program
- a flow graph is prepared and all the possible paths can be covered
- useful for detecting more errors
- Choose enough paths in a program such that maximum logic coverage is achieved

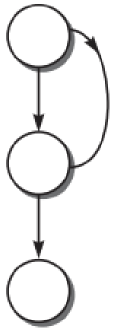
Control Flow Graph

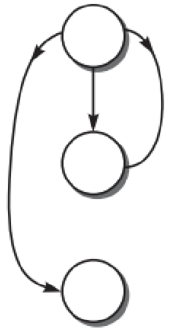
- *Node* It represents one or more procedural statements. The nodes are denoted by a circle. These are numbered or labeled.
- *Edges or links* They represent the flow of control in a program. This is denoted by an arrow on the edge. An edge must terminate at a node.
- *Decision node* A node with more than one arrow leaving it is called a decision node.
- *Junction node* A node with more than one arrow entering it is called a junction.
- *Regions* Areas bounded by edges and nodes are called regions. When counting the regions, the area outside the graph is also considered a region.

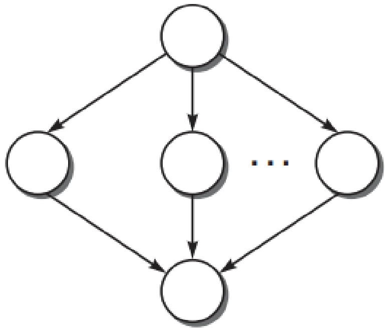
Decision-to-Decision graph or *DD graph*


(a) Sequence


(b) If-Then-Else


(c) Do-While


(d) While-Do


(e) Switch-Case

Path Testing Terminology

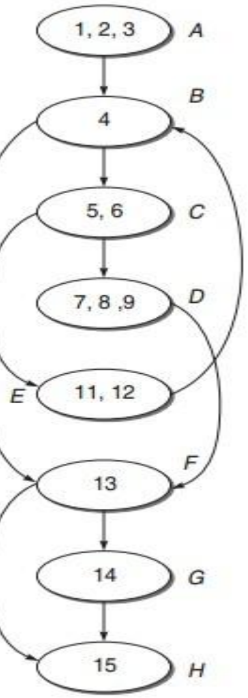
- ***Path*** A path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same, junction, decision, or exit.
- ***Segment*** Paths consist of segments. The smallest segment is a link, that is, a single process that lies between two nodes
- ***Length of a path*** The length of a path is measured by the number of links in it or the number of nodes traverse
- ***Independent path*** An independent path is any path through the graph that introduces at least one new set of processing statements or new conditions.

Control Flow Graph

- A Control Flow Graph is a visual representation of all possible paths that might be traversed through a program during its execution.
- **Nodes** represent basic blocks, sequences of instructions with no branches except at the entry and exit. There also have Entry and Exit Block.
- **Edges** represent control flow, the possible transitions between blocks based on conditions, loops, or jumps.

Example of a Control Flow Graph

```
main()
{
     int number, index;
1.   printf("Enter a number");
2.   scanf("%d, &number);
3.   index = 2;
4.   while(index <= number - 1)
5.   {
6.        if (number % index == 0)
7.        {
8.             printf("Not a prime number");
9.             break;
10.       }
11.       index++;
12.       }
13.       if(index == number)
14.            printf("Prime number");
15.  } //end main
```

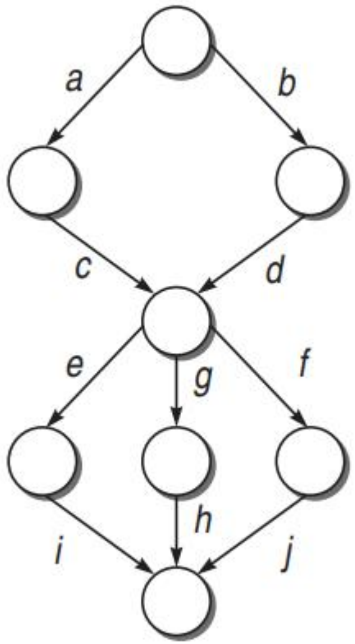


Cyclomatic Complexity

- McCabe has given a measure for the logical complexity of a program by considering the number of paths in the control flow graph of the program
- if they contain at least one cycle, the number of paths is infinite, and we consider only independent paths

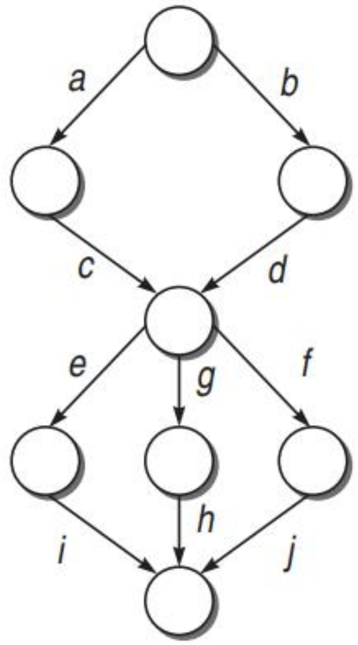
Calculating Cyclomatic Complexity

- How many paths are there?



Calculating Cyclomatic Complexity

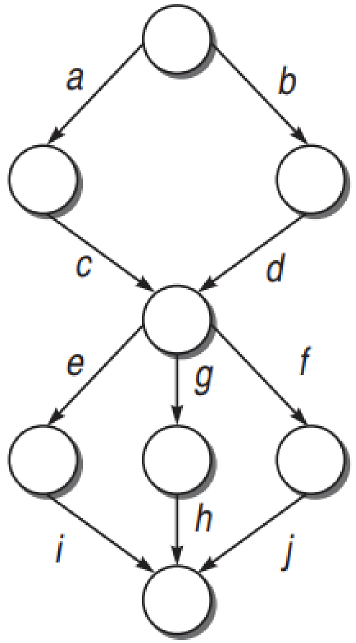
- How many paths are there?
- there are six possible paths:
  acei, acgh, acfh, bdei, bdgh, bdfj
- only four paths are independent, as the other two are always a linear combination



Independent Paths

- In graph theory, it can be demonstrated that in a **strongly connected graph** (in which each node can be reached from any other node), the number of independent paths is given by

$$V(G) = e - n + 1$$

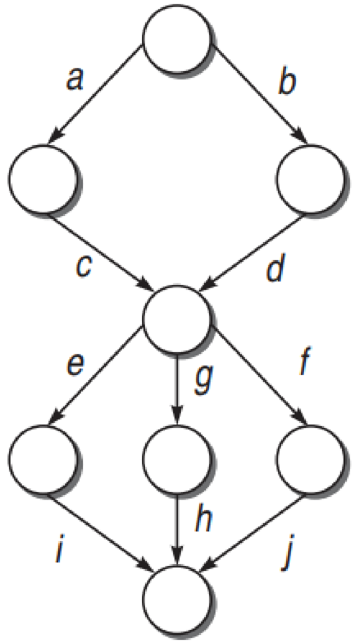


Independent Paths

- In graph theory, it can be demonstrated that in a strongly connected graph (in which each node can be reached from any other node), the number of independent paths is given by

$$V(G) = e - n + 1$$

- to make the graph strongly connected, we add an arc from the last node to the first node of the graph

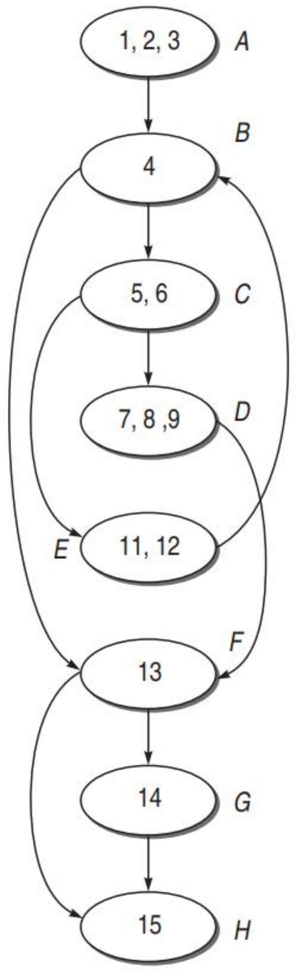


Guidelines for Basis Path Testing

- Draw the flow graph using the code provided for which we have to write test cases.
- Determine the cyclomatic complexity of the flow graph.
- Cyclomatic complexity provides the number of independent paths.
- Determine a basis set of independent paths through the program control structure. The basis set is in fact the base for designing the test cases. Based on every independent path, choose the data such that this path is executed

Draw the DD graph for the program

```
main()
{
     int number, index;
1.   printf(“Enter a number”);
2.   scanf(“%d, &number);
3.   index = 2;
4.   while(index <= number – 1)
5.   {
6.        if (number % index == 0)
7.        {
8.             printf(“Not a prime number”);
9.             break;
10.       }
11.       index++;
12.       }
13.       if(index == number)
14.            printf(“Prime number”);
15. } //end main
```



```
main()
{
     int number, index;
1.   printf(“Enter a number”);
2.   scanf(“%d, &number);
3.   index = 2;
4.   while(index <= number – 1)
5.   {
6.        if (number % index == 0)
7.        {
8.             printf(“Not a prime number”);
9.             break;
10.       }
11.       index++;
12.       }
13.       if(index == number)
14.            printf(“Prime number”);
15. } //end main
```

Calculating Cyclomatic Complexity

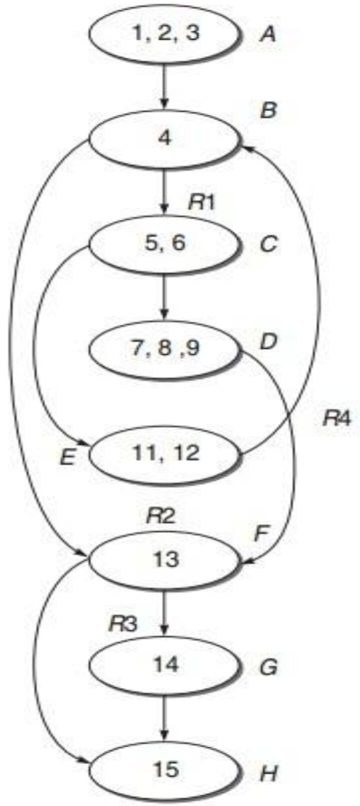
- 1. $\boldsymbol{V(G) = e - n + 2p}$
  where e is number of edges, n is the number of nodes in the graph, and p is number of components in the whole graph.
- 2. $\boldsymbol{V(G) = d + p}$
  where d is the number of decision nodes in the graph.
- 3. $\boldsymbol{V(G)}$ = **number of regions** in the graph

Cyclomatic Complexity

$$\begin{aligned} V(G) &= e - n + 2 * p \\ &= 10 - 8 + 2 \\ &= 4 \end{aligned}$$

$$\begin{aligned} V(G) &= \text{Number of predicate nodes} + 1 \\ &= 3 \text{ (Nodes } B, C, \text{ and } F) + 1 \\ &= 4 \end{aligned}$$

$$\begin{aligned} V(G) &= \text{Number of regions} \\ &= 4(R1, R2, R3, R4) \end{aligned}$$



Independent Path & Test Cases

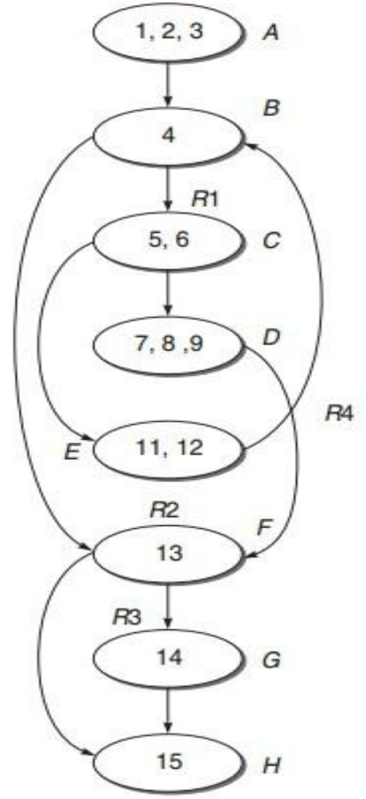
(i) A-B-F-H

(ii) A-B-F-G-H

(iii) A-B-C-E-B-F-G-H

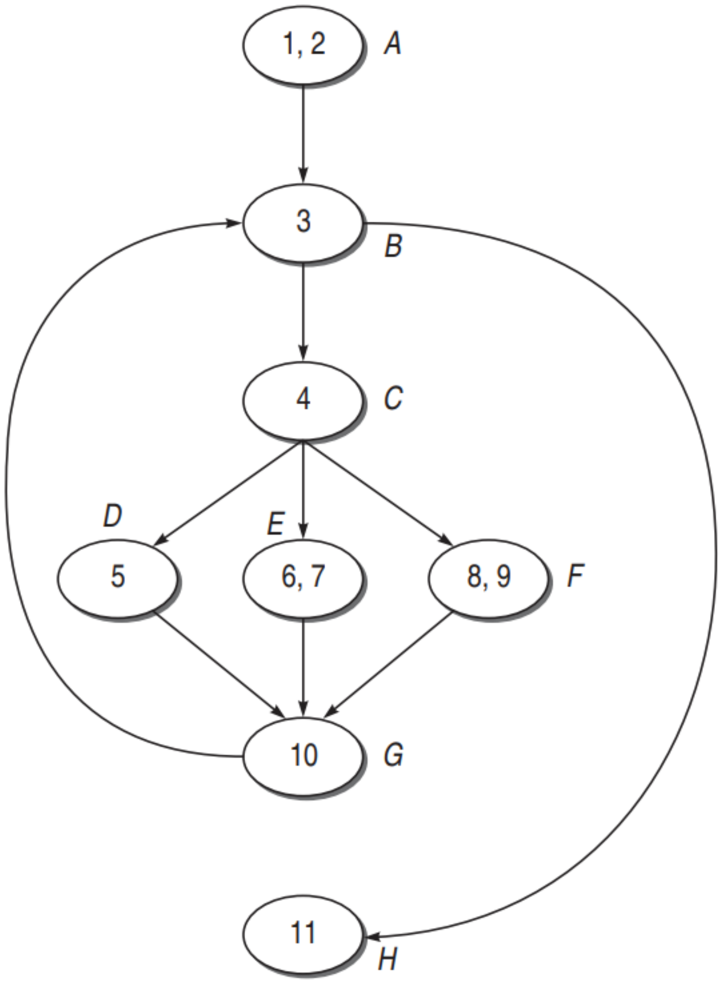
(iv) A-B-C-D-F-H

Test case ID	Input num	Expected result	Independent paths covered by test case
1	1	No output is displayed	A-B-F-H
2	2	Prime number	A-B-F-G-H
3	4	Not a prime number	A-B-C-D-F-H
4	3	Prime number	A-B-C-E-B-F-G-H



Draw the DD graph for the program

```
        main()
        {
            char string [80];
            int index;
1.          printf(“Enter the string for checking its characters”);
2.          scanf(“%s”, string);
3.          for(index = 0; string[index] != ‘\0’; ++index)   {
4.              if((string[index] >= ‘0’ && (string[index] <=‘9’
5.                          printf(“%c is a digit”, string[index]);
6.              else if ((string[index] >= ‘A’ && string[index] <‘Z’))  ||
                        ((string[index] >= ‘a’ && (string[index] <‘z’)))
7.                      printf(“%c is an alphabet”, string[index]);
8.              else
9.                      printf(“%c is a special character”, string[index]);
10.             }
11.     }
```



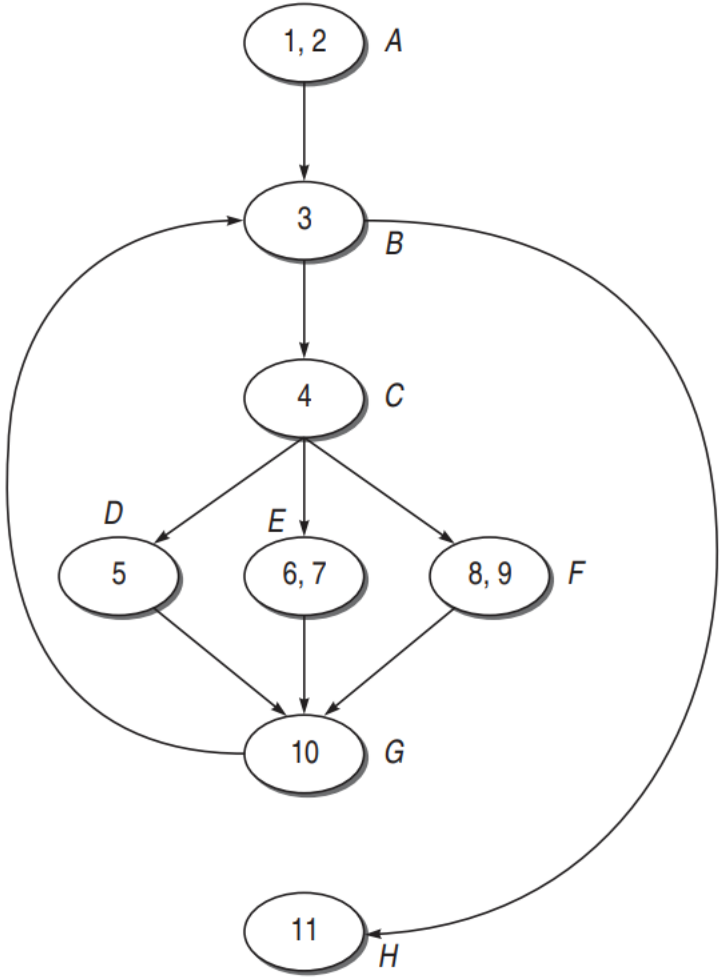
```
    main()
    {
        char string [80];
        int index;
1.      printf("Enter the string for checking its characters");
2.      scanf("%s", string);
3.      for(index = 0; string[index] != '\0'; ++index)   {
4.          if((string[index] >= '0' && (string[index] <='9'
5.                      printf("%c is a digit", string[index]);
6.          else if ((string[index] >= 'A' && string[index] <'Z'))  ||
                    ((string[index] >= 'a' && (string[index] <'z')))
7.                  printf("%c is an alphabet", string[index]);
8.          else
9.                  printf("%c is a special character", string[index]);
10.         }
11. }
```

Cyclomatic complexity

(i) $V(G) = e - n + 2 * P$
$= 10 - 8 + 2$
$= 4$

(ii) $V(G) =$ Number of predicate nodes $+ 1$
$= 3$ (Nodes B, C) $+ 1$
$= 4$

(iii) $V(G) =$ Number of regions
$= 4$



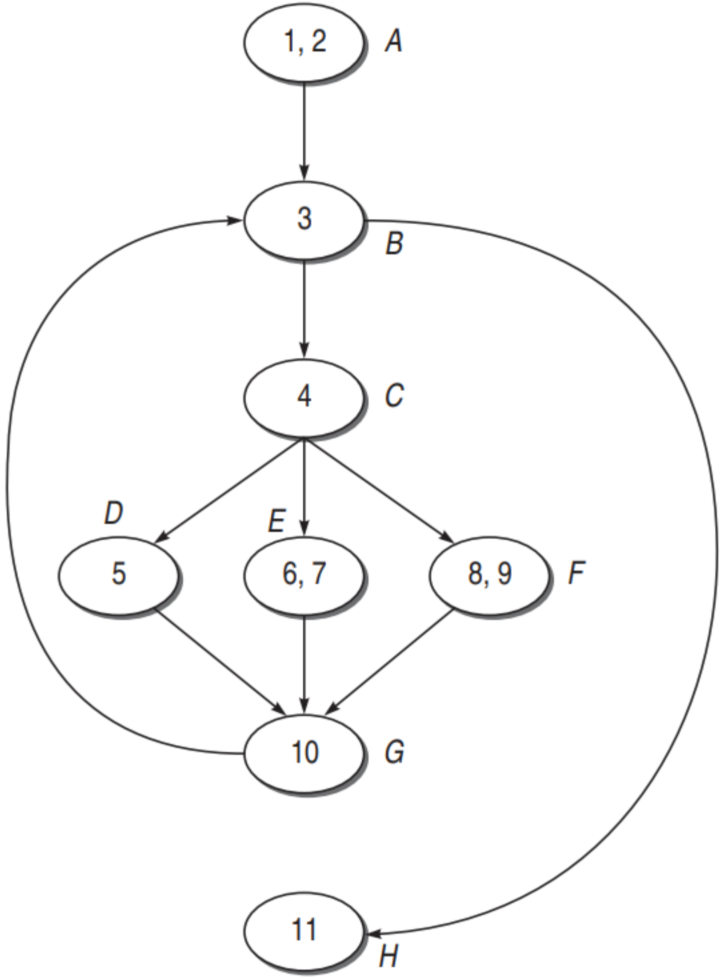
Independent paths

(i) *A-B-H*

(ii) *A-B-C-D-G-B-H*

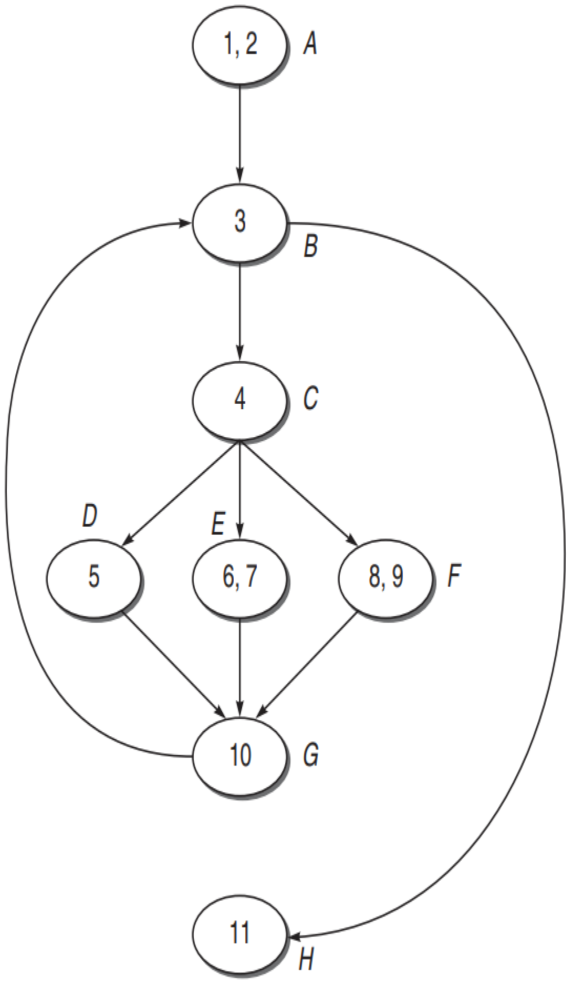
(iii) *A-B-C-E-G-B-H*

(iv) *A-B-C-F-G-B-H*



Independent paths

Test Case ID	Input Line	Expected Output	Independent paths by Test case
1	0987	0 is a digit 9 is a digit 8 is a digit 7 is a digit	A-B-C-D-G-B-H A-B-H
2	AzxG	A is a alphabet z is a alphabet x is a alphabet G is a alphabet	A-B-C-E-G-B-H A-B-H
3	@#	@ is a special character # is a special character	A-B-C-F- G-B-H A-B-H



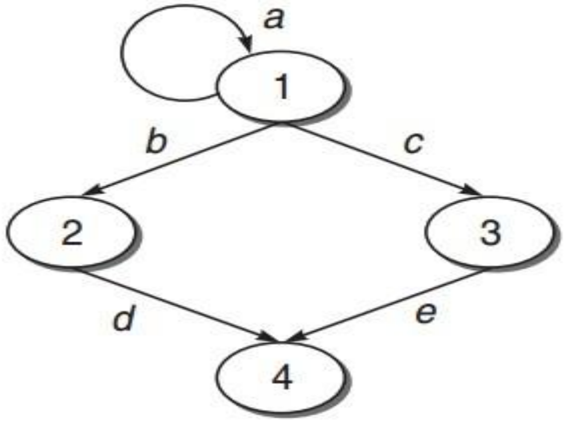
Applications of Path Testing

- *Thorough testing*
  Cyclomatic complexity along with basis path analysis employs more comprehensive scrutiny of code structure and control flow, providing a far superior coverage technique
- *Unit testing*
  Since each decision outcome is tested independently, path testing uncovers these errors in unit testing
- *Integration testing*
  Since modules in a program may call other modules or be called by some other module, there may be chances of interface errors during calling of the modules.
- *Maintenance testing*
  If you have earlier prepared a unit test suite, it should be run on the modified software or a selected path testing can be done as a part of regression testing.

Graph Matrix

- Graph matrix, a data structure, is the solution which can assist in developing a tool for automation of path tracing.
- Graph theorems can be proved easily with the help of graph matrices are very useful for understanding the testing theory.
- Each cell in the matrix can be a direct connection or link between one node to another node.

Graph Matrix

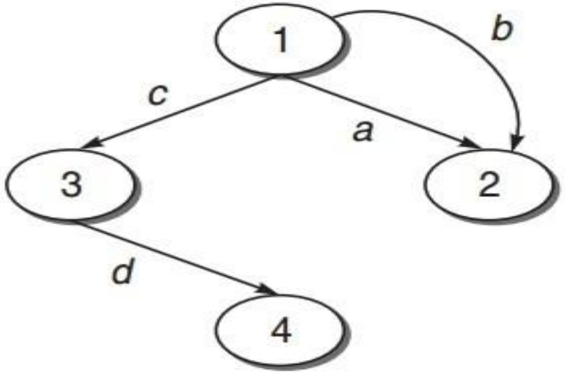


The flow graph

	1	2	3	4
1	a	b	c	
2				d
3				e
4				

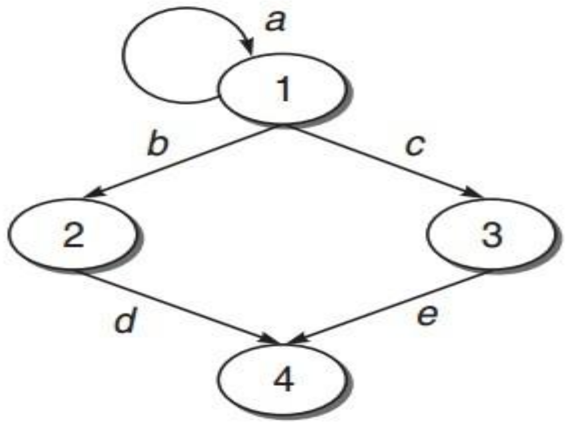
The graph matrix

Graph Matrix



	1	2	3	4
1		a+b	c	
2				
3				d
4				

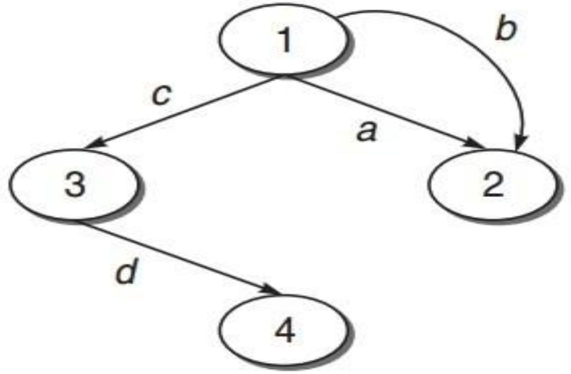
Connection Matrix



	1	2	3	4
1	a	b	c	
2				d
3				e
4				

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

Connection Matrix



	1	2	3	4
1		a+b	c	
2				
3				d
4				

	1	2	3	4
1		1	1	
2				
3				1
4				

Use of Connection Matrix in Finding Cyclomatic Complexity

Step 1: For each row, count the number of 1s and write it in front of that row.

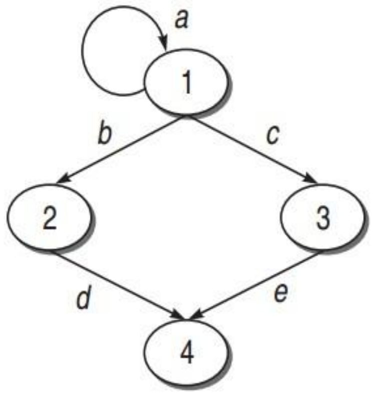
Step 2: Subtract 1 from that count. Ignore the blank rows, if any.

Step 3: Add the fi nal count of each row.

Step 4: Add 1 to the sum calculated in Step 3.

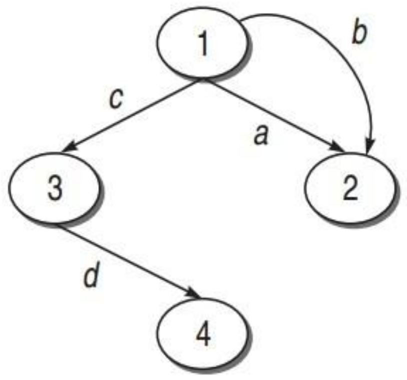
Step 5: The final sum in Step 4 is the cyclomatic number of the graph.

Use Of Connection Matrix In Finding Cyclomatic Complexity Number



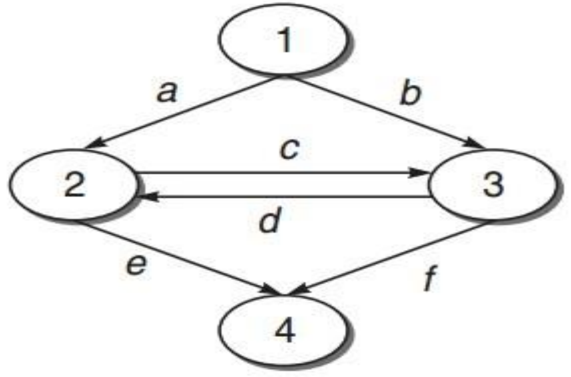
	1	2	3	4	
1	1	1	1		**3 – 1 = 2**
2				1	**1 – 1 = 0**
3				1	**1 – 1 = 0**
4					
	Cyclomatic number = 2+1 = 3				

Use Of Connection Matrix In Finding Cyclomatic Complexity Number



	1	2	3	4	
1		1	1		**2 – 1 = 1**
2					
3				1	**1 – 1 = 0**
4					
	Cyclomatic number = 1+1 = 2				

Use of Graph Matrix to Find K-Link Paths

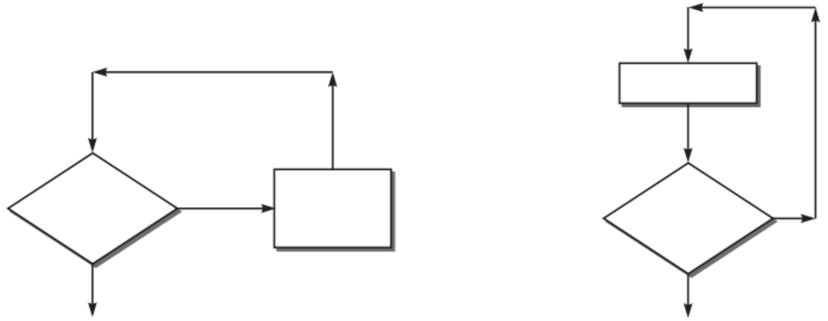


$$\begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & bd & ac & ae + bf \\ 0 & cd & 0 & cf \\ 0 & 0 & dc & de \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

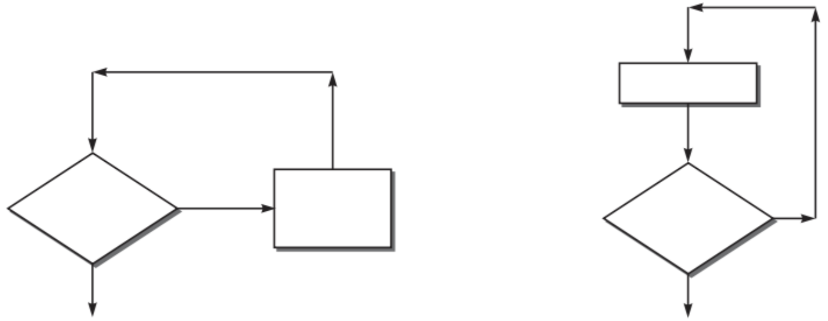
Consider the graph matrix in example above and find 2-link paths for each node.

Loop Testing (Simple Loop)



- Check whether you can bypass the loop or not. If the test case for bypassing the loop is executed and, still you enter inside the loop, it means there is a bug.
- Check whether the loop control variable is negative.
- Write one test case that executes the statements inside the loop.

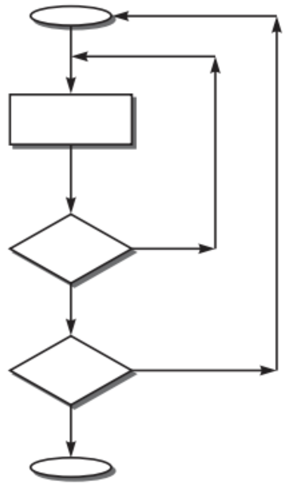
Loop Testing (Simple Loop)



- Write test cases for checking the boundary values of the maximum and minimum number of iterations defined (say min and max) in the loop. It means we should test for min, min+1, min–1, max–1, max, and max+1 number of iterations through the loop.
- Write test cases for a typical number of iterations through the loop

Loop Testing (Nested Loop)

- Adopt the approach of simple tests to test the nested loops
- Start with the innermost loops while holding outer loops to their minimum values
- Continue this outward in this manner until all loops have been covered

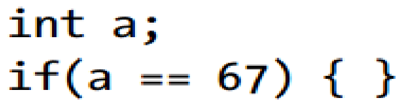


Data Flow Testing

data flow testing gives a chance to look out for inappropriate data definition, its use in predicates, computations, and termination.

For example, if an out-of-scope data is being used in a computation, then it is a bug.

There may be several patterns like this which indicate data anomalies.



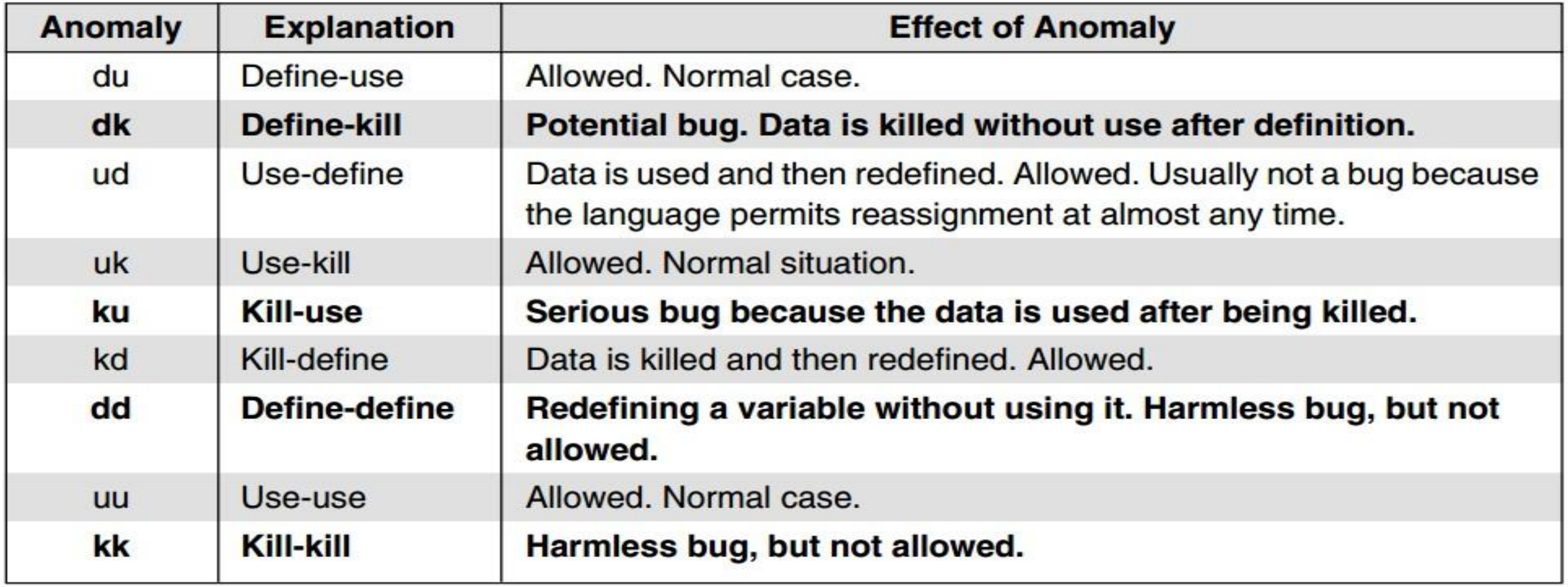
```
int a;
if(a == 67) { }
```

Potential Bug

State of A Data Object

- **Defined (d)**: A data object is called defined when it is initialized, i.e. when it is on the left side of an assignment statement. Defined state can also be used to mean that a file has been opened, a dynamically allocated object has been allocated, something is pushed onto the stack, a record written, and so on.
- **Killed/Undefined/Released (k)**: When the data has been reinitialized or the scope of a loop control variable finishes, i.e. exiting the loop or memory is released dynamically or a file has been closed.
- **Usage (u)**: When the data object is on the right side of assignment or used as a control variable in a loop, or in an expression used to evaluate the control flow of a case statement, or as a pointer to an object, etc. In general, we say that the usage is either computational use (c-use) or predicate use (p-use).

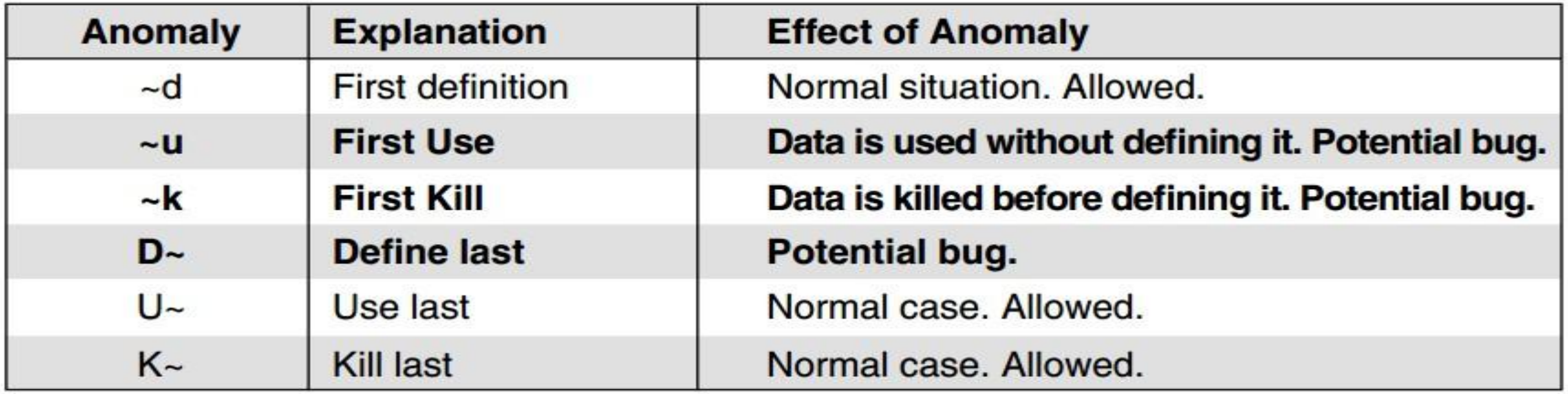
Data-Flow Anomalies



Anomaly	Explanation	Effect of Anomaly
du	Define-use	Allowed. Normal case.
dk	**Define-kill**	**Potential bug. Data is killed without use after definition.**
ud	Use-define	Data is used and then redefined. Allowed. Usually not a bug because the language permits reassignment at almost any time.
uk	Use-kill	Allowed. Normal situation.
ku	**Kill-use**	**Serious bug because the data is used after being killed.**
kd	Kill-define	Data is killed and then redefined. Allowed.
dd	**Define-define**	**Redefining a variable without using it. Harmless bug, but not allowed.**
uu	Use-use	Allowed. Normal case.
kk	**Kill-kill**	**Harmless bug, but not allowed.**

Data-Flow Anomalies

- ~x: indicates all prior actions are not of interest to x.
- x~ : indicates all post actions are not of interest to x.



Anomaly	Explanation	Effect of Anomaly
~d	First definition	Normal situation. Allowed.
~u	**First Use**	**Data is used without defining it. Potential bug.**
~k	**First Kill**	**Data is killed before defining it. Potential bug.**
D~	**Define last**	**Potential bug.**
U~	Use last	Normal case. Allowed.
K~	Kill last	Normal case. Allowed.

Data Flow Testing

- Consider the program given below for calculating the gross salary of an employee in an organization.
- If his basic salary is less than BDT 1500, then House Rent Allowance (HRA) = 10% of basic salary and Direct Allowance (DA) = 90% of the basic.
- If his/her salary is either equal to or above BDT 1500, then HRA = BDT 500 and DA = 98% of the basic salary.
- Calculate his gross salary

Data Flow Testing

- Consider the program given below for calculating the gross salary of an employee in an organization.
- If his basic salary is less than BDT 1500, then House Rent Allowance (HRA) = 10% of basic salary and Direct Allowance (DA) = 90% of the basic.
- If his/her salary is either equal to or above BDT 1500, then HRA = BDT 500 and DA = 98% of the basic salary.
- Calculate his gross salary

```
main()
{
    float bs, gs, da, hra = 0;
    printf("Enter basic salary");
    scanf("%f", &bs);
    if(bs < 1500)
    {
        hra = bs * 10/100;
        da = bs * 90/100;
    }
    else
    {
        hra = 500;
        da = bs * 98/100;
    }
    gs = bs + hra + da;
    printf("Gross Salary = Rs. %f", gs);
}
```

Data Flow Testing

For variable ‘bs’, the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	3	Normal case. Allowed
du	3-4	Normal case. Allowed
uu	4-6, 6-7, 7-12, 12-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

For variable ‘gs’, the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	14	Normal case. Allowed
du	14-15	Normal case. Allowed
uk	15-16	Normal case. Allowed
K~	16	Normal case. Allowed

```
      main()
      {
1.        float bs, gs, da, hra = 0;
2.        printf(“Enter basic salary”);
3.        scanf(“%f”, &bs);
4.        if(bs < 1500)
5.        {
6.            hra = bs * 10/100;
7.            da = bs * 90/100;
8.        }
9.        else
10.       {
11.           hra = 500;
12.           da = bs * 98/100;
13.       }
14.       gs = bs + hra + da;
15.       printf(“Gross Salary = Rs. %f”, gs);
16.   }
```

Data Flow Testing

For variable 'da', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	7	Normal case. Allowed
du	7-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

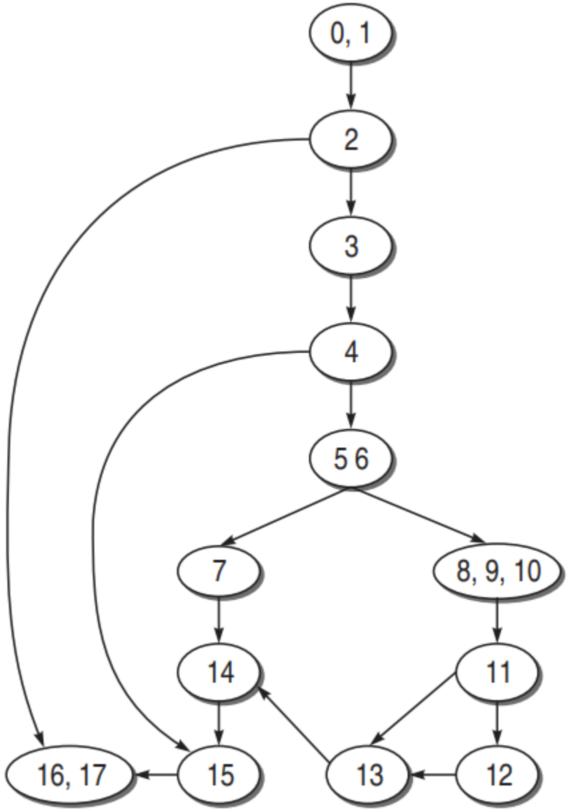
For variable 'hra', the define-use-kill patterns are given below.

Pattern	Line Number	Explanation
~d	1	Normal case. Allowed
dd	1-6 or 1-11	Double definition. Not allowed. Harmless bug.
du	6-14 or 11-14	Normal case. Allowed
uk	14-16	Normal case. Allowed
K~	16	Normal case. Allowed

```
      main()
      {
1.        float bs, gs, da, hra = 0;
2.        printf("Enter basic salary");
3.        scanf("%f", &bs);
4.        if(bs < 1500)
5.        {
6.            hra = bs * 10/100;
7.            da = bs * 90/100;
8.        }
9.        else
10.       {
11.           hra = 500;
12.           da = bs * 98/100;
13.       }
14.       gs = bs + hra + da;
15.       printf("Gross Salary = Rs. %f", gs);
16.   }
```

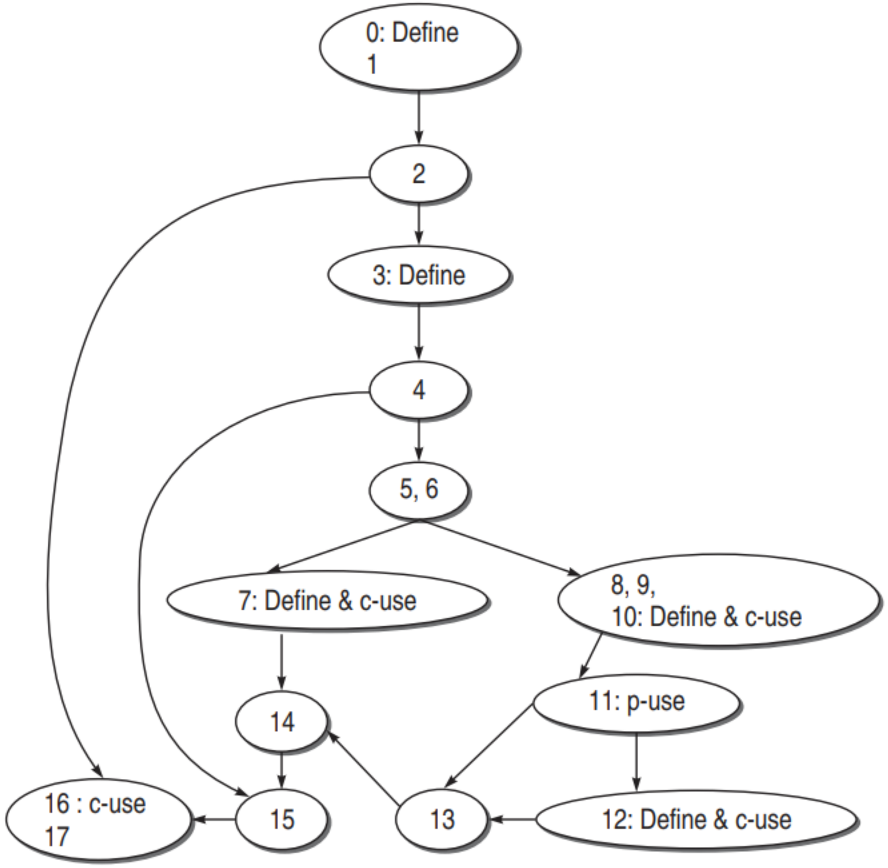
Dynamic data flow testing

- *All-du Paths (ADUP)* It states that every du-path from every definition of every variable to every use of that definition should be exercised under some test
- *All-uses (AU)* This states that for every use of the variable, there is a path from the definition of that variable (nearest to the use in backward direction) to the use.
- *All-p-uses/Some-c-uses (APU + C)* This strategy states that for every variable and every definition of that variable, include at least one dc-path from the definition to every predicate use
- *All-c-uses/Some-p-uses (ACU + P)* This strategy states that for every variable and every definition of that variable, include at least one dc-path from the definition
- *All-Definition (AD)* It states that every definition of every variable should be covered by at least one use of that variable, be that a computational use or a predicate use



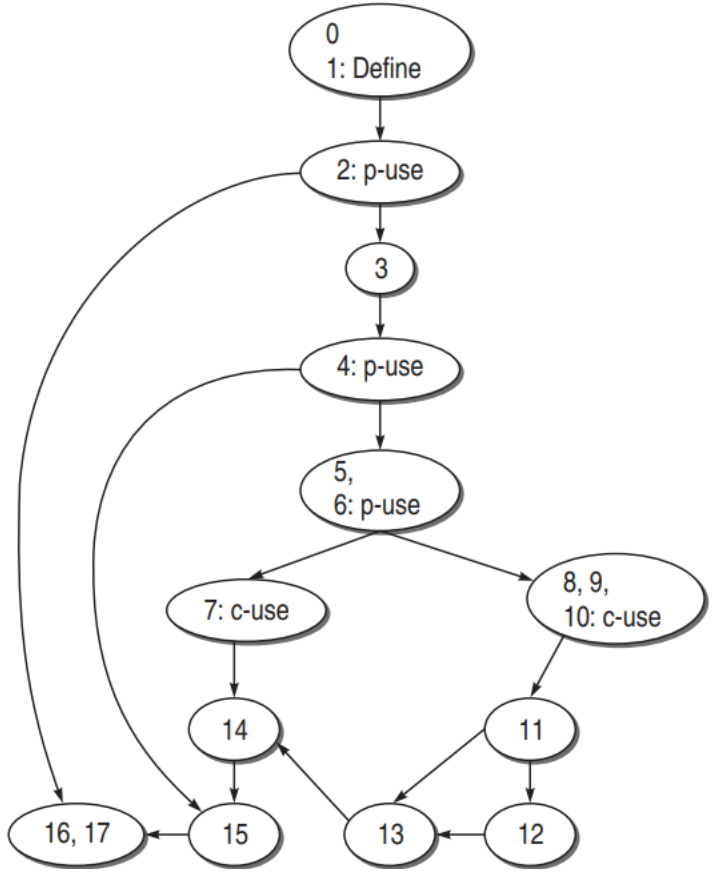
```
     main()
     {
     int work;
 0.  double payment =0;
 1.  scanf("%d", work);
 2.  if (work > 0) {
 3.      payment = 40;
 4.  if (work > 20)
 5.  {
 6.      if(work <= 30)
 7.          payment = payment + (work - 25) * 0.5;
 8.      else
 9.      {
10.          payment = payment + 50 + (work -30) * 0.1;
11.              if (payment >= 3000)
12.                  payment = payment * 0.9;
13.      }
14.  }
15.  }
16.  printf("Final payment", payment);
```

Data flow graph for 'payment'



```
    main()
    {
    int work;
0.  double payment =0;
1.  scanf("%d", work);
2.  if (work > 0) {
3.      payment = 40;
4.  if (work > 20)
5.  {
6.      if(work <= 30)
7.          payment = payment + (work - 25) * 0.5;
8.      else
9.      {
10.         payment = payment + 50 + (work -30) * 0.1;
11.             if (payment >= 3000)
12.                 payment = payment * 0.9;
13.     }
14. }
15. }
16. printf("Final payment", payment);
```

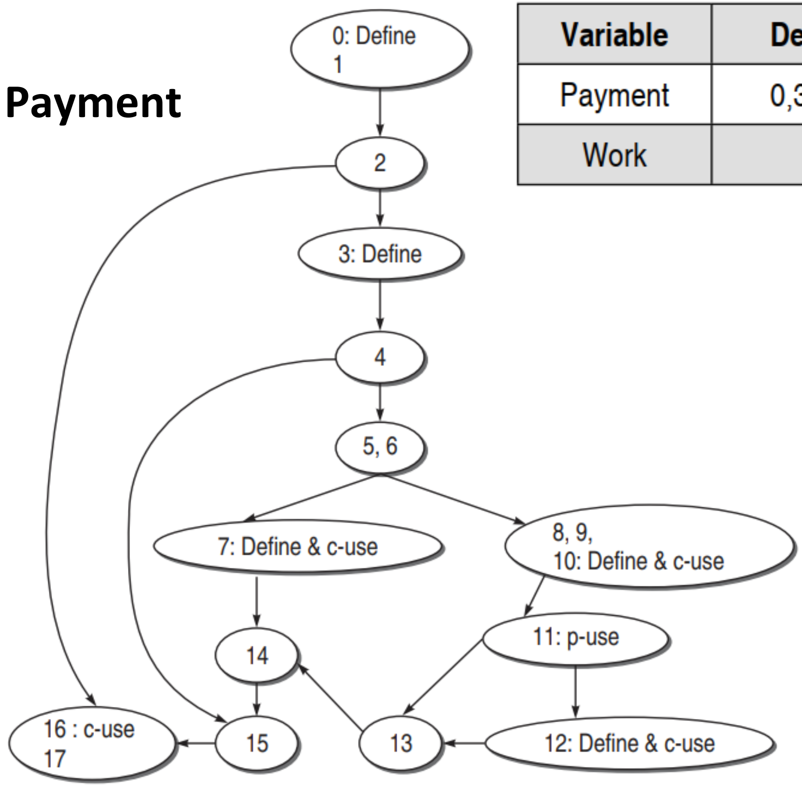
Data flow graph for variable ‘work’



```
     main()
     {
     int work;
 0.  double payment =0;
 1.  scanf(“%d”, work);
 2.  if (work > 0) {
 3.      payment = 40;
 4.  if (work > 20)
 5.  {
 6.      if(work <= 30)
 7.          payment = payment + (work - 25) * 0.5;
 8.      else
 9.      {
10.          payment = payment + 50 + (work -30) * 0.1;
11.              if (payment >= 3000)
12.                  payment = payment * 0.9;
13.      }
14.  }
15.  }
16.  printf(“Final payment”, payment);
```

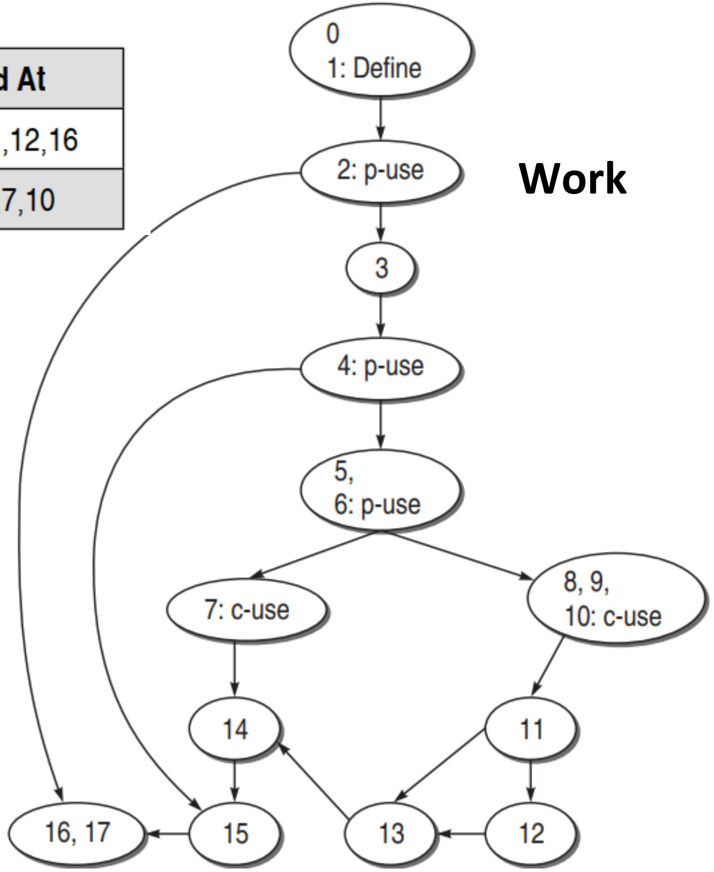
List of all the variables

Payment

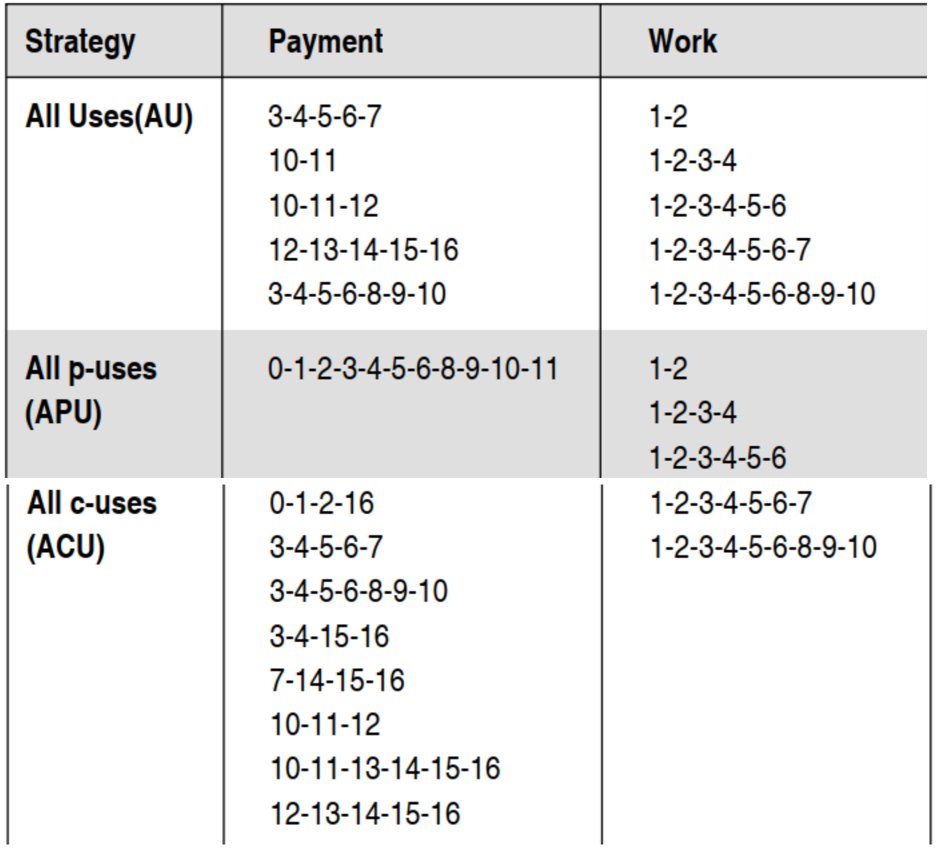


Variable	Defined At	Used At
Payment	0,3,7,10,12	7,10,11,12,16
Work	1	2,4,6,7,10

Work

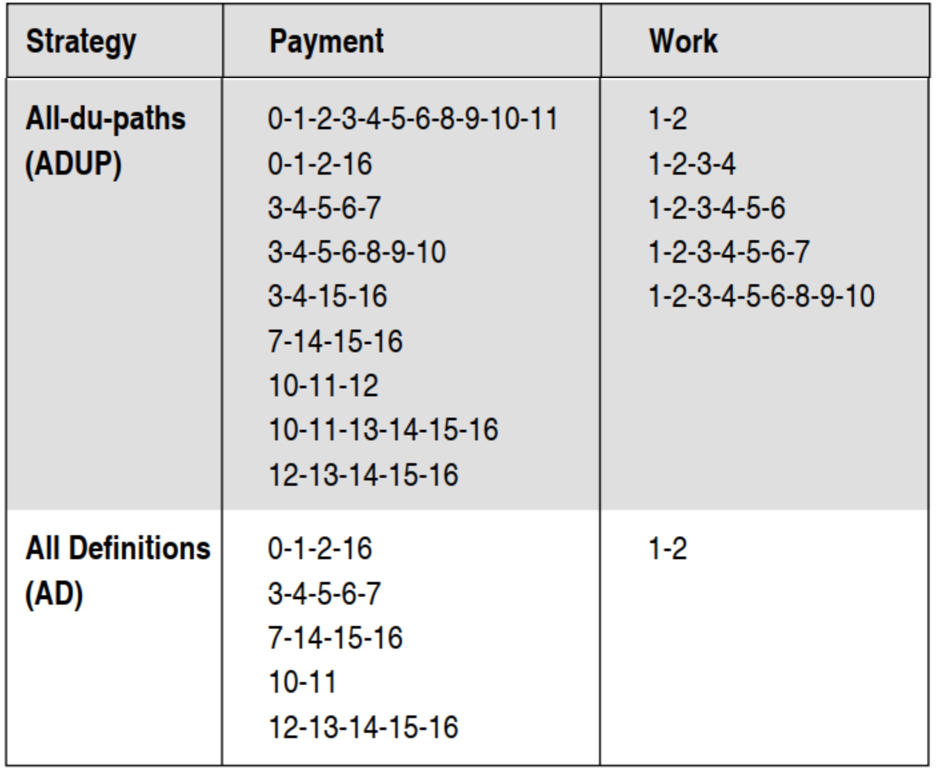


Data flow testing paths



Strategy	Payment	Work
All Uses(AU)	3-4-5-6-7 10-11 10-11-12 12-13-14-15-16 3-4-5-6-8-9-10	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All p-uses (APU)	0-1-2-3-4-5-6-8-9-10-11	1-2 1-2-3-4 1-2-3-4-5-6
All c-uses (ACU)	0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16	1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10

Data flow testing paths



Strategy	Payment	Work
All-du-paths (ADUP)	0-1-2-3-4-5-6-8-9-10-11 0-1-2-16 3-4-5-6-7 3-4-5-6-8-9-10 3-4-15-16 7-14-15-16 10-11-12 10-11-13-14-15-16 12-13-14-15-16	1-2 1-2-3-4 1-2-3-4-5-6 1-2-3-4-5-6-7 1-2-3-4-5-6-8-9-10
All Definitions (AD)	0-1-2-16 3-4-5-6-7 7-14-15-16 10-11 12-13-14-15-16	1-2

Mutation Testing

- Mutation testing is the process of mutating some segment of code (putting some error in the code) and then, testing this mutated code with some test data.
- During mutation testing, faults are introduced into a program by creating many versions of the program, each of which contains one fault
- Test data are used to execute these faulty programs with the goal of causing each faulty program to fail. Faulty programs are called *mutants* of the original program and a mutant is said to be *killed* when a test case causes it to fail.

Secondary mutants

- During secondary mutants, multiple levels of mutation are applied on the initial program.
- In this case, it is very difficult to identify the initial program from its mutants

Consider the program *P* shown below.

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

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;
}
```

M3:

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

M4:

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

Secondary mutants

Let us consider the following test data selection:

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

We apply these test data to mutants, M1, M2, M3, and M4.

	P	M1	M2	M3	M4	Killed Mutants
TD1	3	3	3	3	1	M4
TD2	2	2	3	2	1	M2 and M4
TD3	1	1	1	1	1	none

Primary mutants

- When the mutants are single modifications of the initial program using some operators as shown above, they are called *primary mutants*.

```
if (a > b)
    x = x + y;
else
    x = y;
printf("%d", x);
```

```
M1: x = x - y;
M2: x = x / y;
M3: x =  x + 1;
M4: printf("%d", y);
```

Test Data	x	y	Initial Program Result	Mutant Result
TD1	2	2	4	0 (M1)
TD2(x and y # 0)	4	3	7	1.4 (M2)
TD3 (y #1)	3	2	5	4 (M3)
TD4(y #0)	5	2	7	2 (M4)

Mutation testing process

- Construct the mutants of a test program.
- Add test cases to the mutation system and check the output of the program on each test case to see if it is correct.
- If the output is incorrect, a fault has been found and the program must be modified and the process restarted.
- If the output is correct, that test case is executed against each live mutant.
- If the output of a mutant differs from that of the original program on the same test case, the mutant is assumed to be incorrect and is killed.

Mutation testing process

- After each test case has been executed against each live mutant, each
  remaining mutant falls into one of the following two categories
  - One, the mutant is functionally equivalent to the original program.
  - Two, the mutant is killable, but the set of test cases is insufficient to kill it.
- The mutation score for a set of test data is the percentage of non-equivalent mutants killed by that data. If the mutation score is 100%, then the test data is called *mutation-adequate*

Thank You

If you have any question regarding this lecture, please email me
saeed[dot]siddik[at]iit.du.ac.bd