SOFTWARE TESTING

White Box Testing Strategies

Dr. ARIVUSELVAN.K

Associate Professor – (SCORE)

VIT University

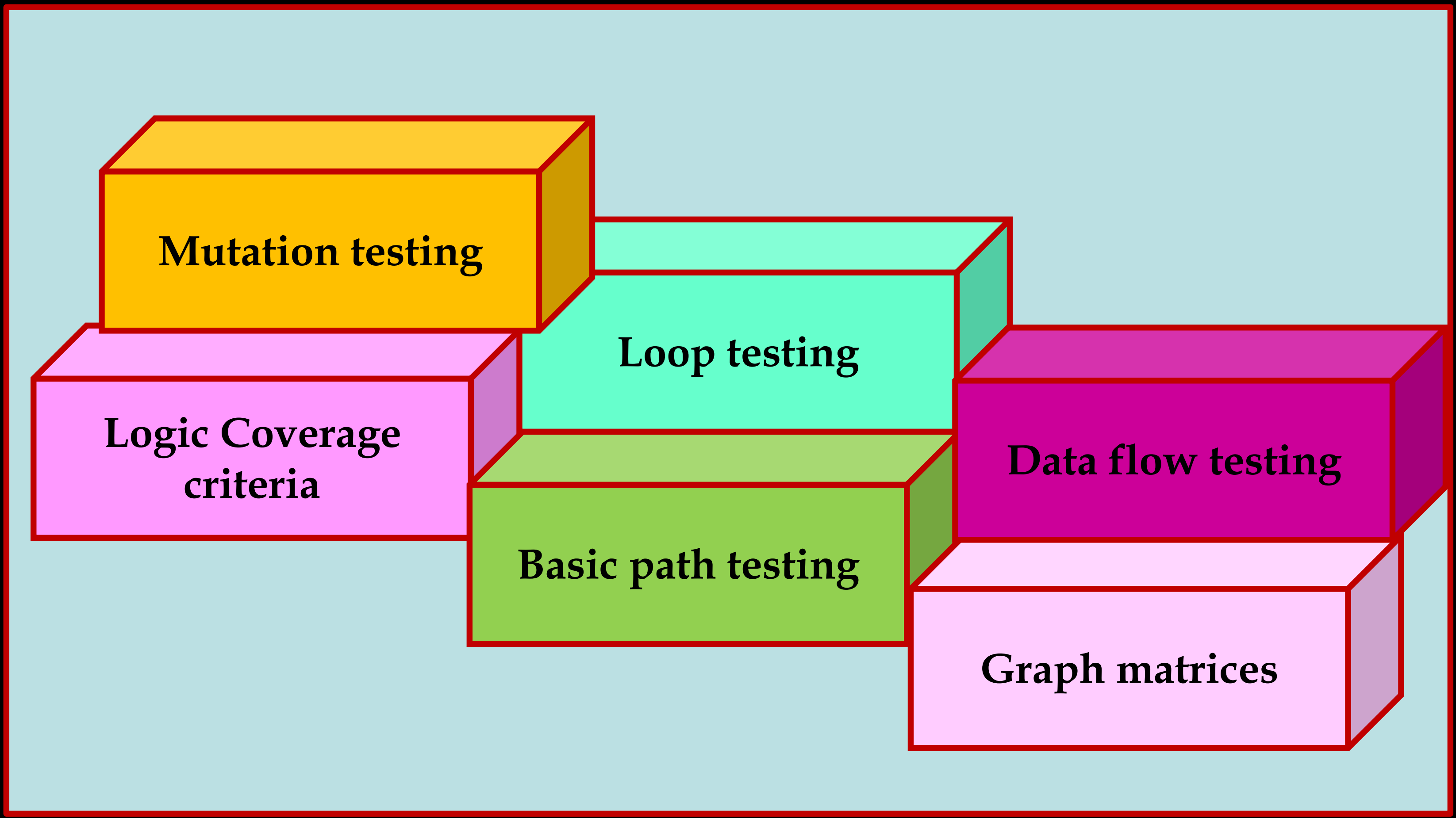
White Box Testing

White-box testing ensures that the internal parts of the software are adequately tested.

The entire design, structure, and code of the software have to be studied.

It is also known as glass-box testing (or) structural testing.

WHITE-BOX TESTING STRATEGIES



Logic Coverage Criteria

The **basic forms of logic coverage** are listed below:

(1) Statement Coverage:

It is assumed that **if all the statements** of the **module** are **executed once**, **every bug** will be **notified**.

Consider the following **code segment**:

```
scanf ("%d", &x);
scanf ("%d", &y);
while (x != y)
{
      if (x > y)
            x = x - y;
      else
            y = y - x;
 }
 printf ("x = ", x);
 printf ("y = ", y);
```

If we want to **cover every statement** in the above code, then the following **test cases** must be designed:

Test case 1: x = y = n, where n is any number

Test case 2: x = n, y = m, where n and m are different numbers.

Test case 3: x > y

Test case 4: x < y

```
scanf ("%d", &x);
scanf ("%d", &y);
while (x != y)
{
        if (x > y)
                x = x - y;
        else
                y = y - x;
 }
 printf ("x = ", x);
 printf ("y = ", y);
```

Test case 1 just skips the while loop and all loop statements are not executed.

Test case 2, the loop is also executed. However, every statement inside the loop is not executed.

If we execute only test case 3 and 4, then conditions and paths in test case 1 will never be tested and errors will go undetected.

(2) Decision (or) Branch Coverage:`

Each **branch direction** must be **traversed at least once** on **all possible outcomes** (True or False).

In the sample code shown in Figure , while and if statements have two outcomes: (True and False.)

So test cases must be designed such that both outcomes for while and if statements are tested.

The test cases are designed as:

Test case 1: x = y

Test case 2: x != y

Test case 3: x < y

Test case 4: x > y

```
while (x != y)
{
        if (x > y)
                x = x - y;
        else
                y = y - x;
 }
```

(3) Condition Coverage:

Each condition in a decision takes on all possible outcomes at least once.

For example, consider the following statement:

$$while\ ((I \leq 5)\ \&\&\ (J < COUNT))$$

In this loop statement, two conditions are there.

Test cases should be designed such that both the conditions are tested for True and False outcomes.

Test case 1: $I \leq 5$, $J < COUNT$

Test case 2: $I < 5$, $J > COUNT$

BASIS PATH TESTING

It is a white-box testing technique based on the control structure of a program.

Using this structure, a control flow graph is prepared and the various possible paths present in the graph are executed as a part of testing.

To design test cases using this technique, four steps are followed :

(1) Construct the Control Flow Graph

(2) Compute the Cyclomatic Complexity of the Graph

(3) Identify the Independent Paths

(4) Design Test cases from Independent Paths

(1) CONTROL FLOW GRAPH:

It is a graphical representation of control structure of a program.

Control Flow graphs can be prepared as a directed graph.

A directed graph (V, E) consists of a set of vertices V and a set of edges E.

Following **notations** are used for a flow graph:

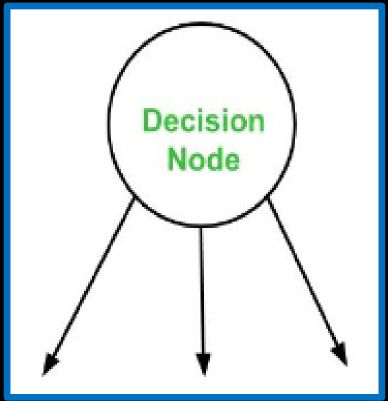
(1) Node : It represents one or more procedural statements. The nodes are denoted by a circle. These are numbered or labeled.

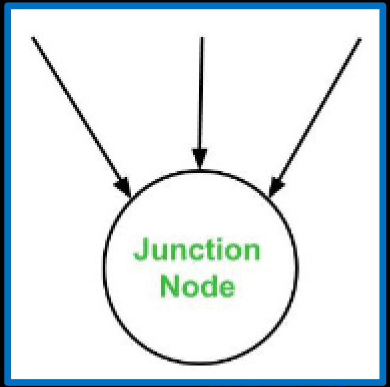
(2) Edges or links: They represent the flow of control in a program. This is denoted by an arrow on the edge.

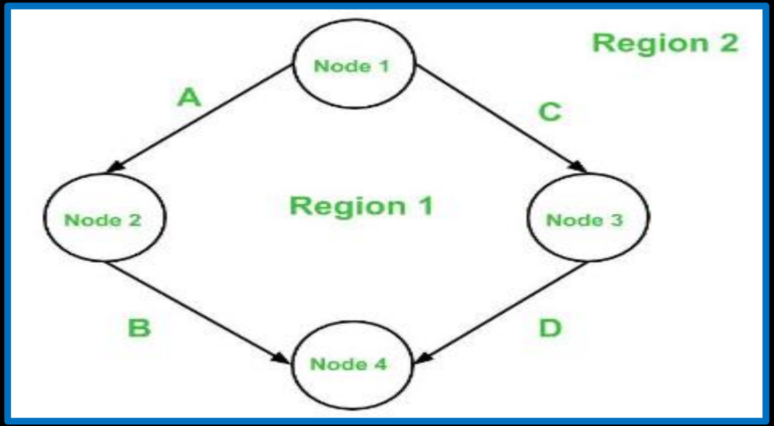
(3) Decision node : A node with more than one arrow leaving.

(4) Junction node: A node with more than one arrow entering .

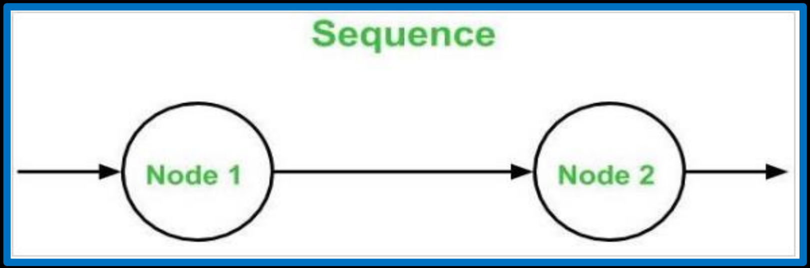
(5) Regions: Areas bounded by edges and nodes. (**NOTE:** When counting the regions, the area outside the graph is also considered a region.)

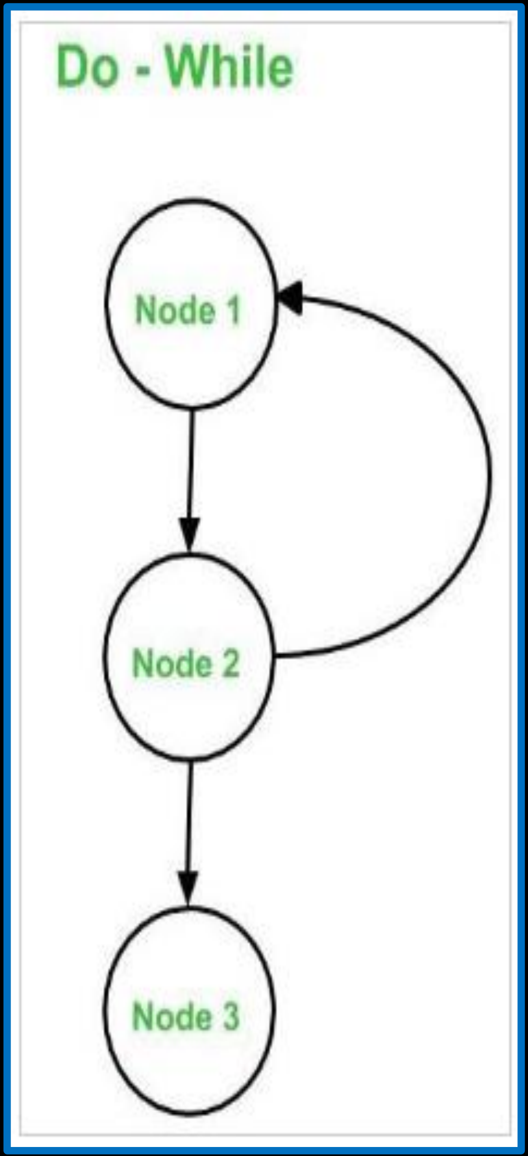


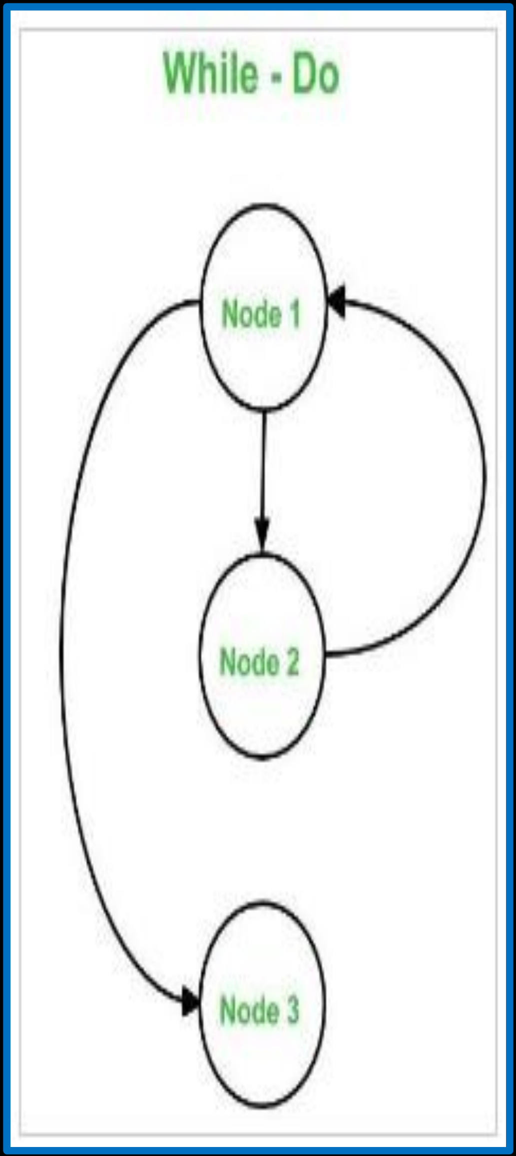




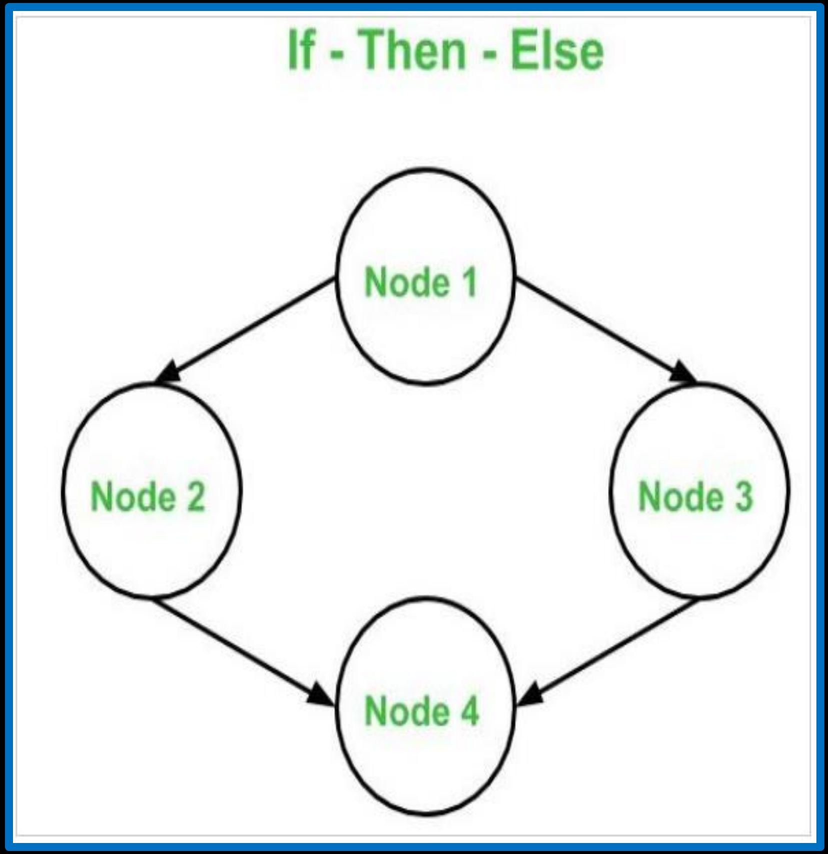
FLOW GRAPH NOTATIONS:

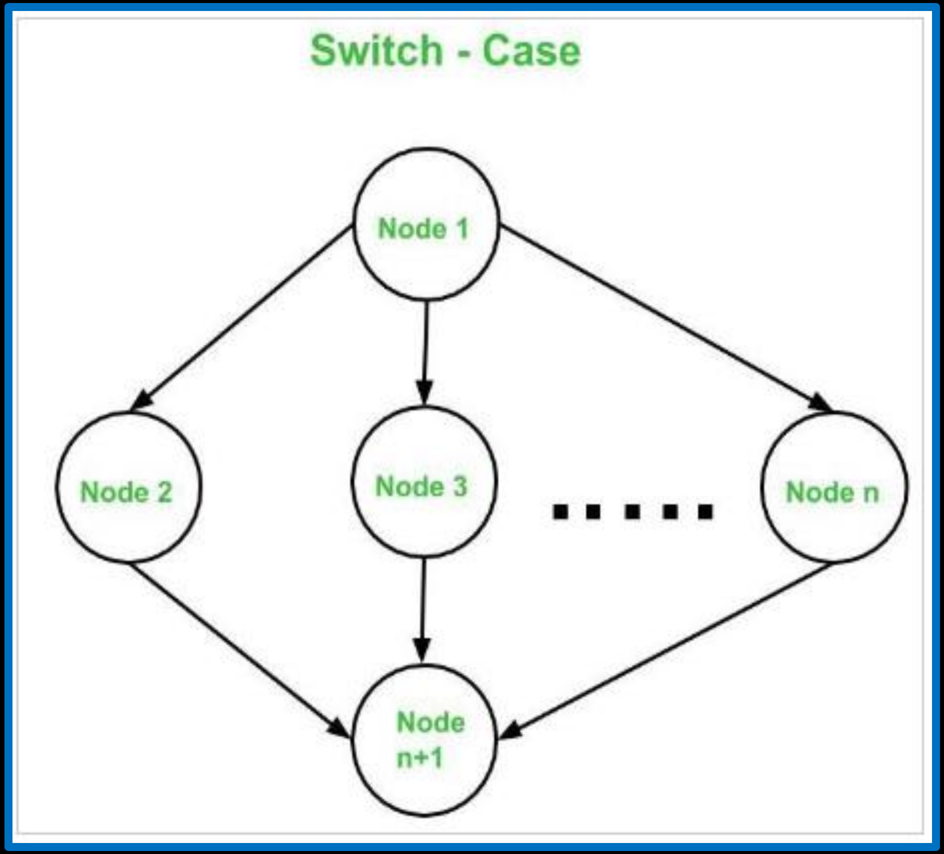






FLOW GRAPH NOTATIONS:





(2) CYCLOMATIC COMPLEXITY :

It is a software metric that measures the logical complexity of the program code.

This metric was developed by Thomas J. McCabe in 1976 .

Cyclomatic Complexity measures the number of linearly independent paths through the program code.

Cyclomatic complexity **indicates several information** about the program code as shown below:

Cyclomatic Complexity	Meaning
1 – 10	• Structured and Well Written Code • High Testability • Less Cost and Effort
10 – 20	• Complex Code • Medium Testability • Medium Cost and Effort
20 – 40	• Very Complex Code • Low Testability • High Cost and Effort
> 40	• Highly Complex Code • Not at all Testable • Very High Cost and Effort

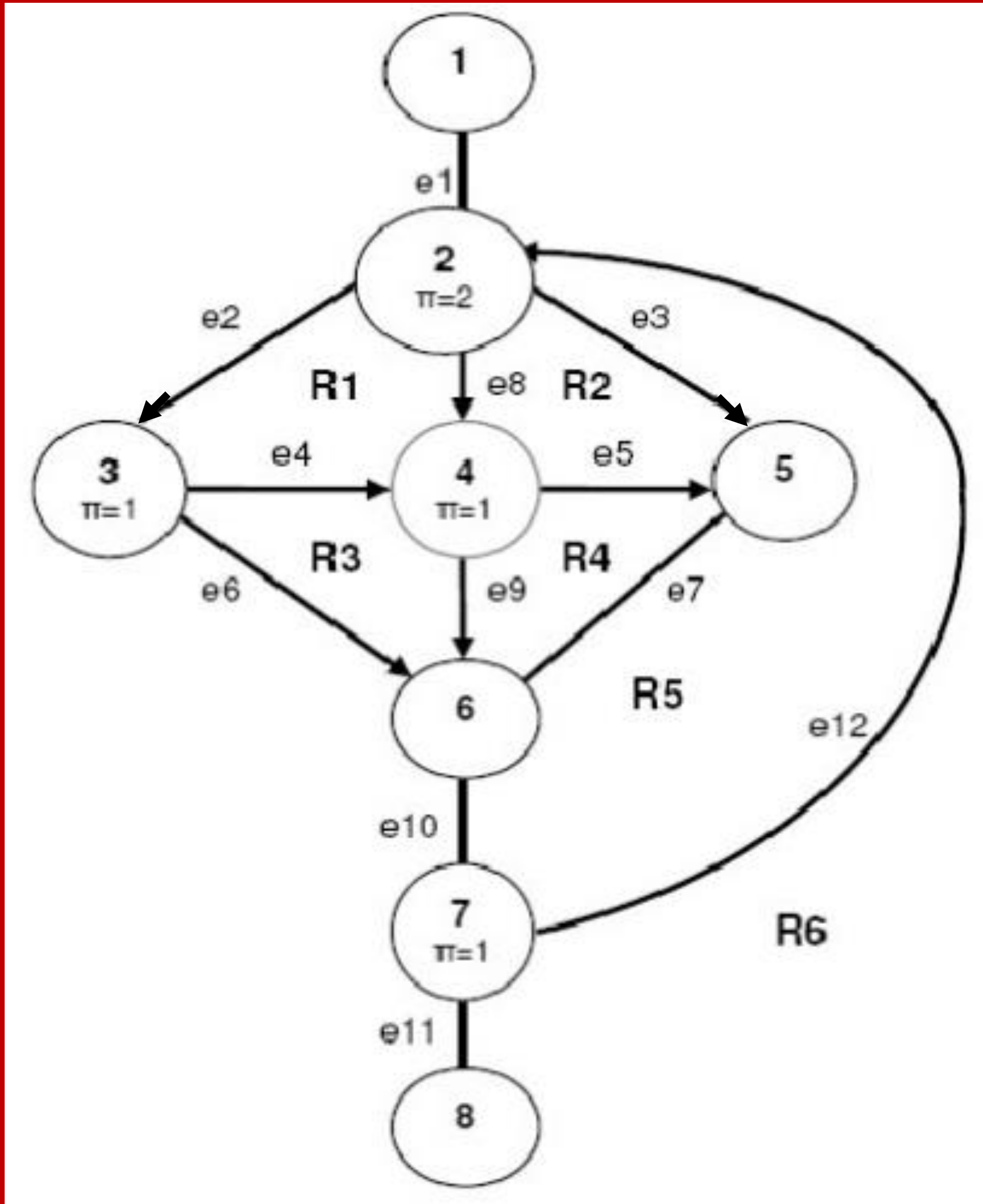
Cyclomatic complexity number can be derived through any of the following three Methods:

(1) $V(G) = e - n + 2*p$; where e is number of edges, n is the number of nodes in the graph and p is Connected components.

V(G) is the maximum number of independent paths in the graph

(2) $V(G) = \pi + 1$; where π is the number of predicate nodes in the graph. [NOTE: Predicate nodes are the conditional nodes. They give rise to two branches in the control flow graph.]

(3) V(G) = number of regions in the graph.



Edges and Nodes Method:

$v = e - n + 2$

$e = 12, n = 8$

$v = 12 - 8 + 2 = 6$

Predicate Method:

$v = \Sigma\pi + 1$

$\Sigma\pi = 5$, sum of predicates

$v = 5 + 1 = 6$

Region (Topological) Method:

$v = \Sigma R$, sum of regions R

$\Sigma R = 6$

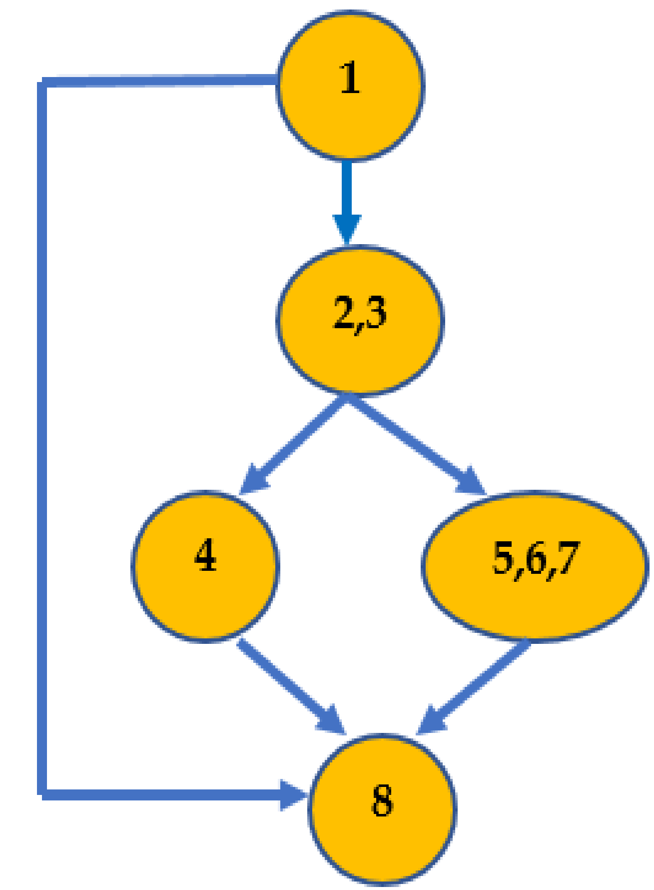
$v = 6$

EXAMPLE - 1

Consider the following program segment:

```
while (a! =b)
{
If (a > b)
a = a – b;
else
b = b – a;
}
return a;
```

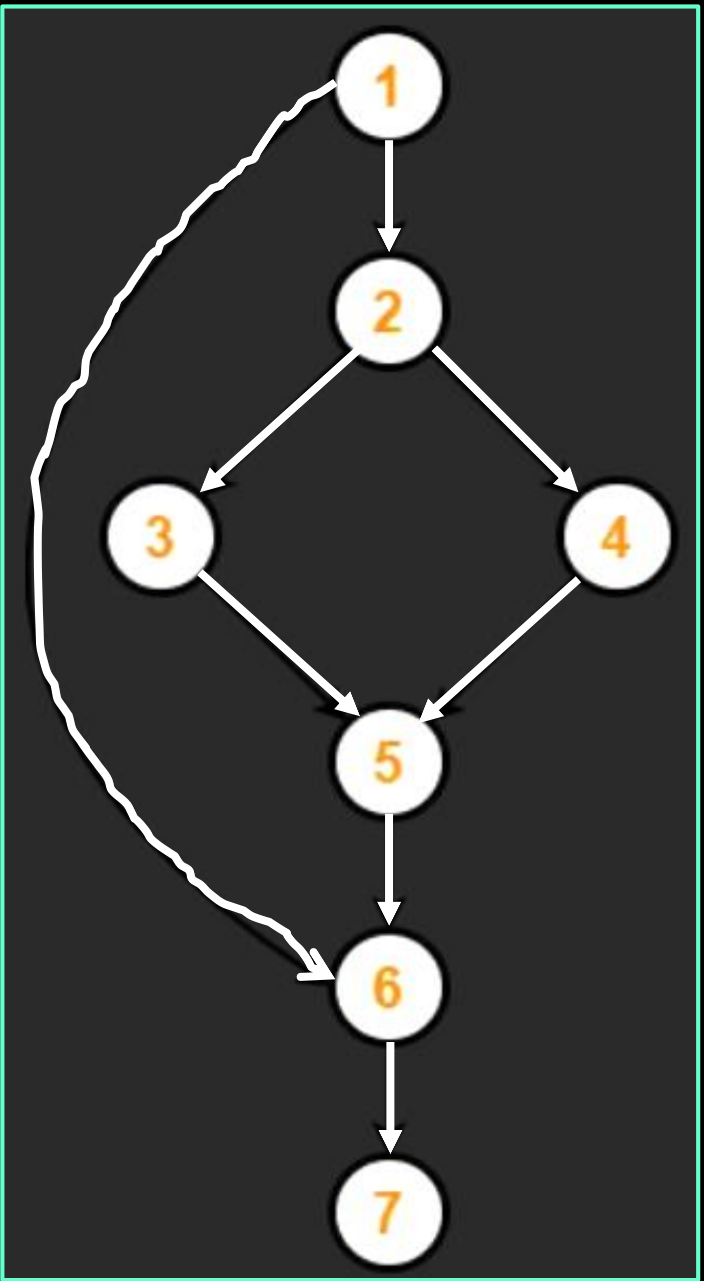
```
1. while (a! =b)
2. {
3. If (a > b)
4. a = a – b;
5. else
6. b = b – a;
7. }
8. return a;
```



EXAMPLE – 1.1

Consider the following program segment:

```
IF A = 354
   THEN IF B > C
         THEN A = B
         ELSE A = C
   END IF
END IF
PRINT A
```



Using the above control flow graph, the cyclomatic complexity may be calculated as:

Method-01:

Cyclomatic Complexity

= Total number of closed regions in the control flow graph + 1

= 2 + 1

= 3

Method-02:

Cyclomatic Complexity

= E – N + 2

= 8 – 7 + 2

= 3

Method-03:

Cyclomatic Complexity

= P + 1

= 2 + 1

= 3

EXAMPLE - 2

Consider the following program segment:

```
main()
{
    int number, index;
    printf("Enter a number");
    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");
} //end main
```

BASIS PATH TESTING

(a) Draw the DD (Decision –To – Decision) graph for the program.

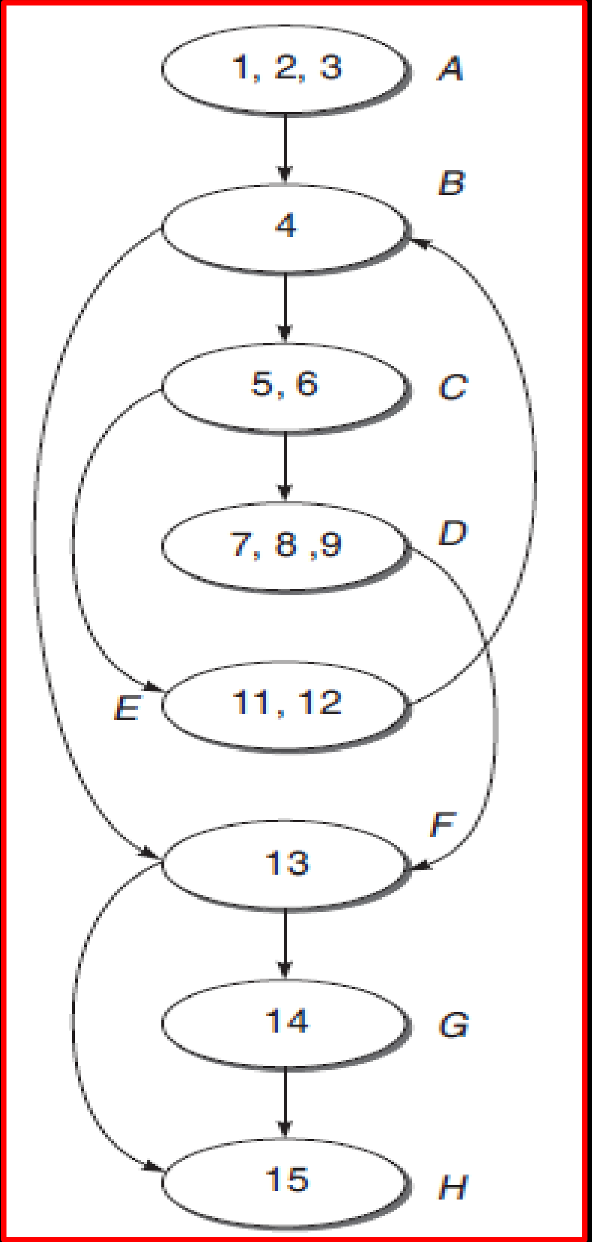
(b) Calculate the cyclomatic complexity of the program using all the methods.

(c) List all independent paths.

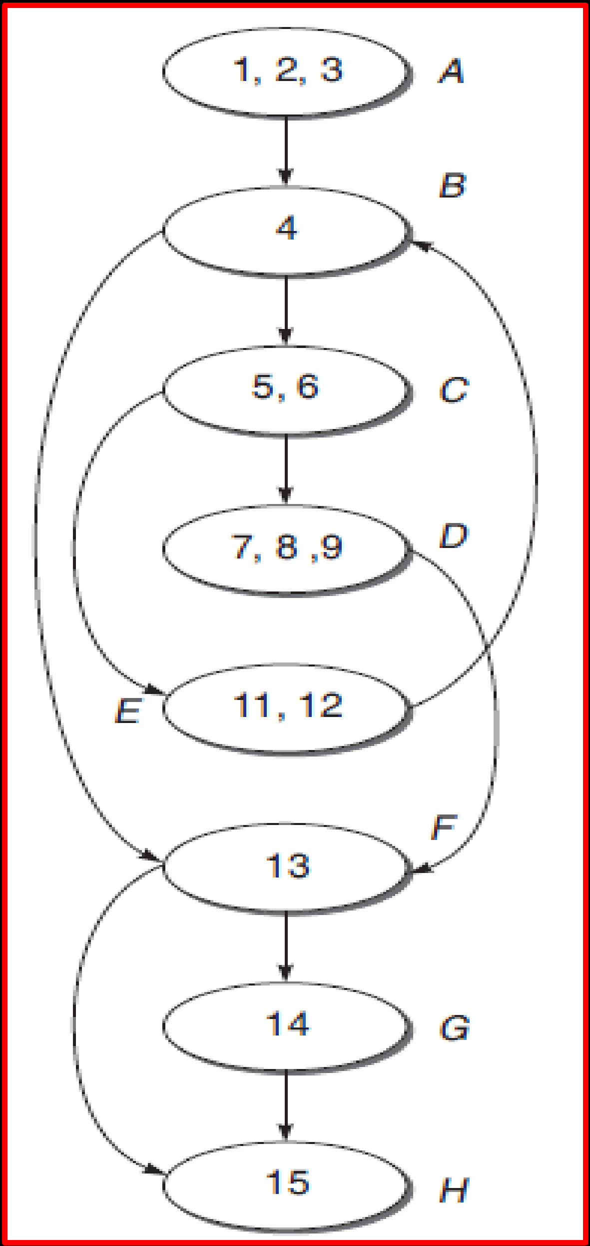
(d) Design test cases from independent paths.

EXAMPLE - 2

```
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
```



DD graph



Put the sequential statements in one node. For example, statements 1, 2, and 3 have been put inside one node.

Put the edges between the nodes according to their flow of execution.

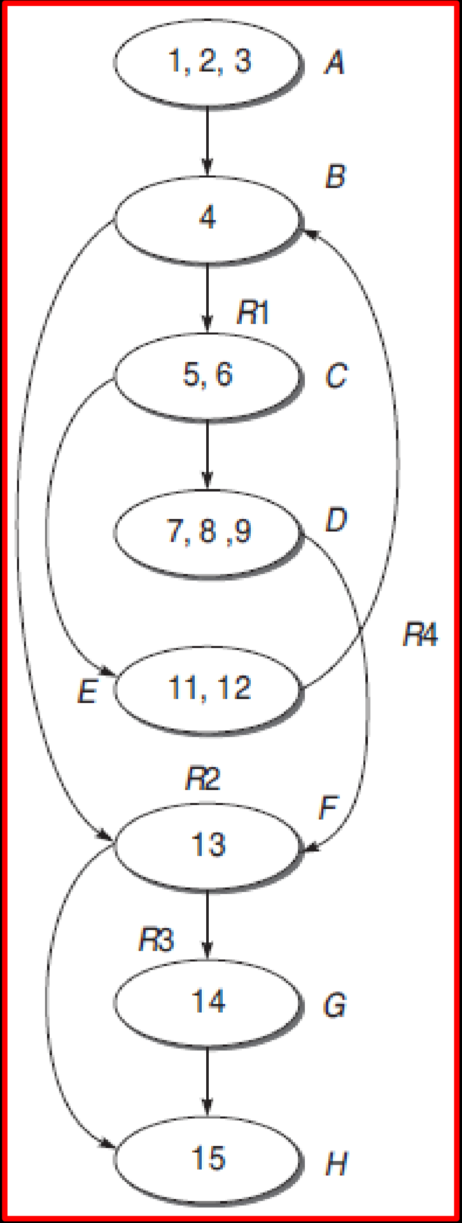
Put alphabetical numbering on each node like A, B, etc.

Cyclomatic complexity

(i) V(G) = e – n + 2 * p
= 10 – 8 + 2*1
= 4

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

(iii) V (G) = Number of regions
= 4 (R1, R2, R3, R4)



Independent paths

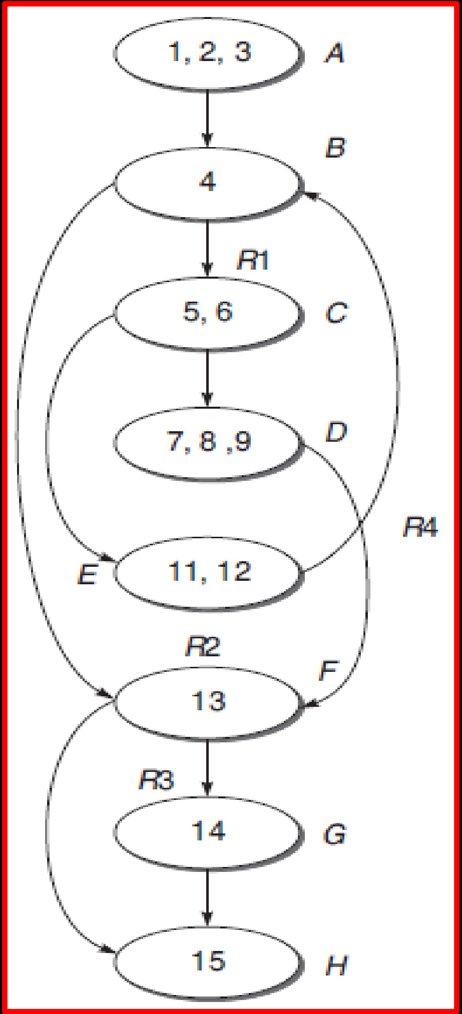
Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

(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 design

Test case design from the list of independent paths:

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

EXAMPLE - 3

Consider the following program that reads in a string and then checks the type of each character.

```
    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. }
```

BASIS PATH TESTING

(a) Draw the DD (Decision –To – Decision) graph for the program.

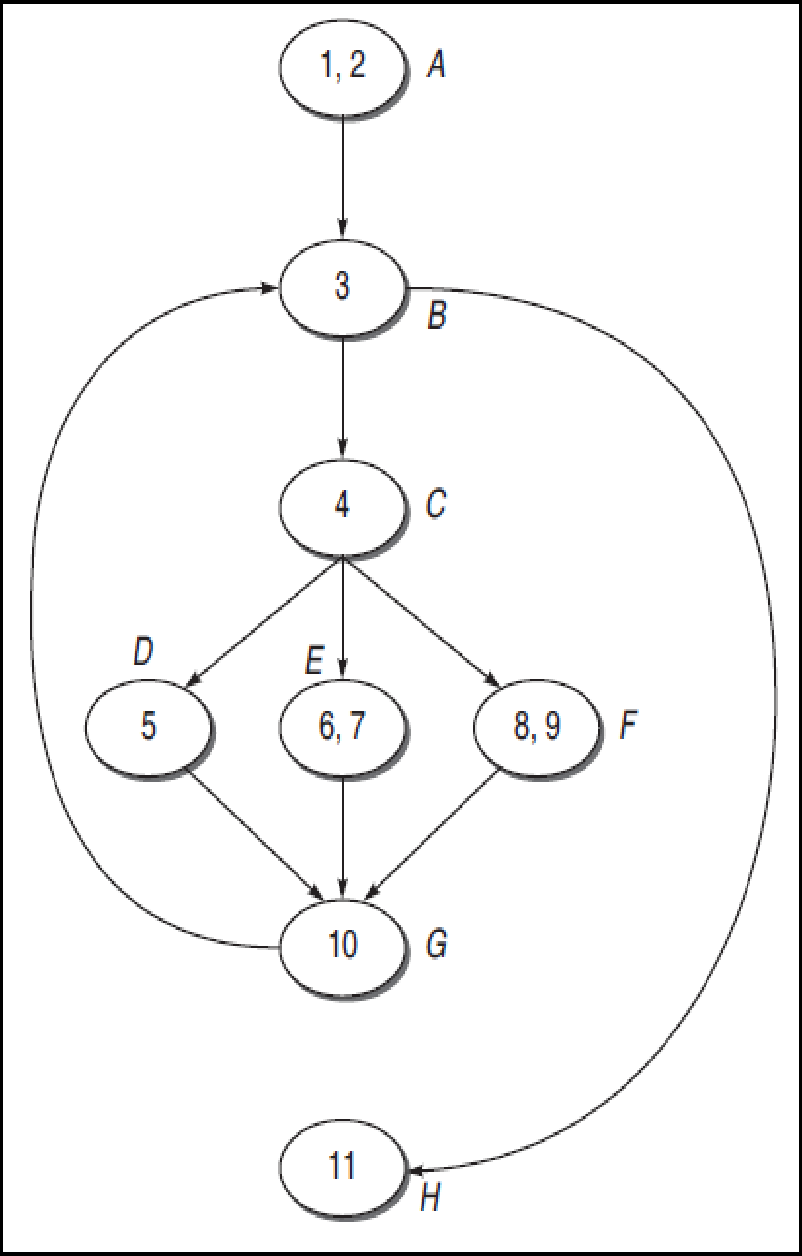
(b) Calculate the cyclomatic complexity of the program using all the methods.

(c) List all independent paths.

(d) Design test cases from independent paths.

DD graph

```
    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) $V(G) = e - n + 2 * p$
$= 10 - 8 + 2*1$
$= 4$

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

Node C is a multiple IF-THEN-ELSE, so for finding out the number of predicate nodes for this case, follow the following formula:

Number of predicated nodes = Number of links out of main node − 1
= 3 – 1 = 2 (For node C)

(iii) V (G) = Number of regions
= 4 (R1, R2, R3, R4)

Independent paths

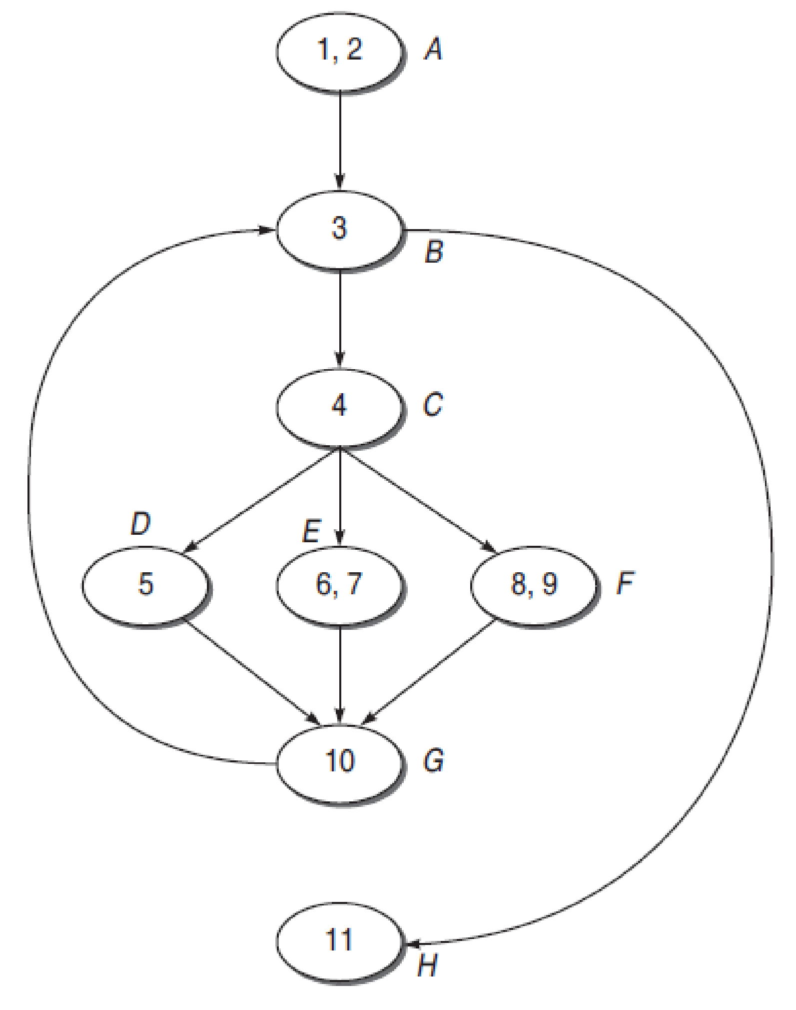
Since the cyclomatic complexity of the graph is 4, there will be 4 independent paths in the graph as shown below:

(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



Test case design

Test case design from the list of independent paths:

Test Case ID	Input Line	Expected Output	Independent paths covered 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

EXAMPLE - 4

Consider the following program:

```
main()
{
    char chr;
    printf (“Enter the special character\n”);
    scanf (%c”, &chr);
    if (chr != 48) && (chr != 49) && (chr != 50) && (chr != 51) &&
       (chr != 52) && (chr != 53) && (chr != 54) && (chr != 55) &&
       (chr != 56) && (chr != 57)
    {
        switch(chr)
       {
       Case ‘*’: printf(“It is a special character”);
       break;
       Case ‘#’: printf(“It is a special character”);
       break;
       Case ‘@’: printf(“It is a special character”);
       break;
       Case ‘!’:  printf(“It is a special character”);
       break;
       Case ‘%’: printf(“It is a special character”);
       break;
       default : printf(“You have not entered a special character”);
       break;
       }// end of switch
    } // end of If
    else
       printf(“You have not entered a character”);
} // end of main()
```

BASIS PATH TESTING

(a) Draw the DD (Decision –To – Decision) graph for the program.

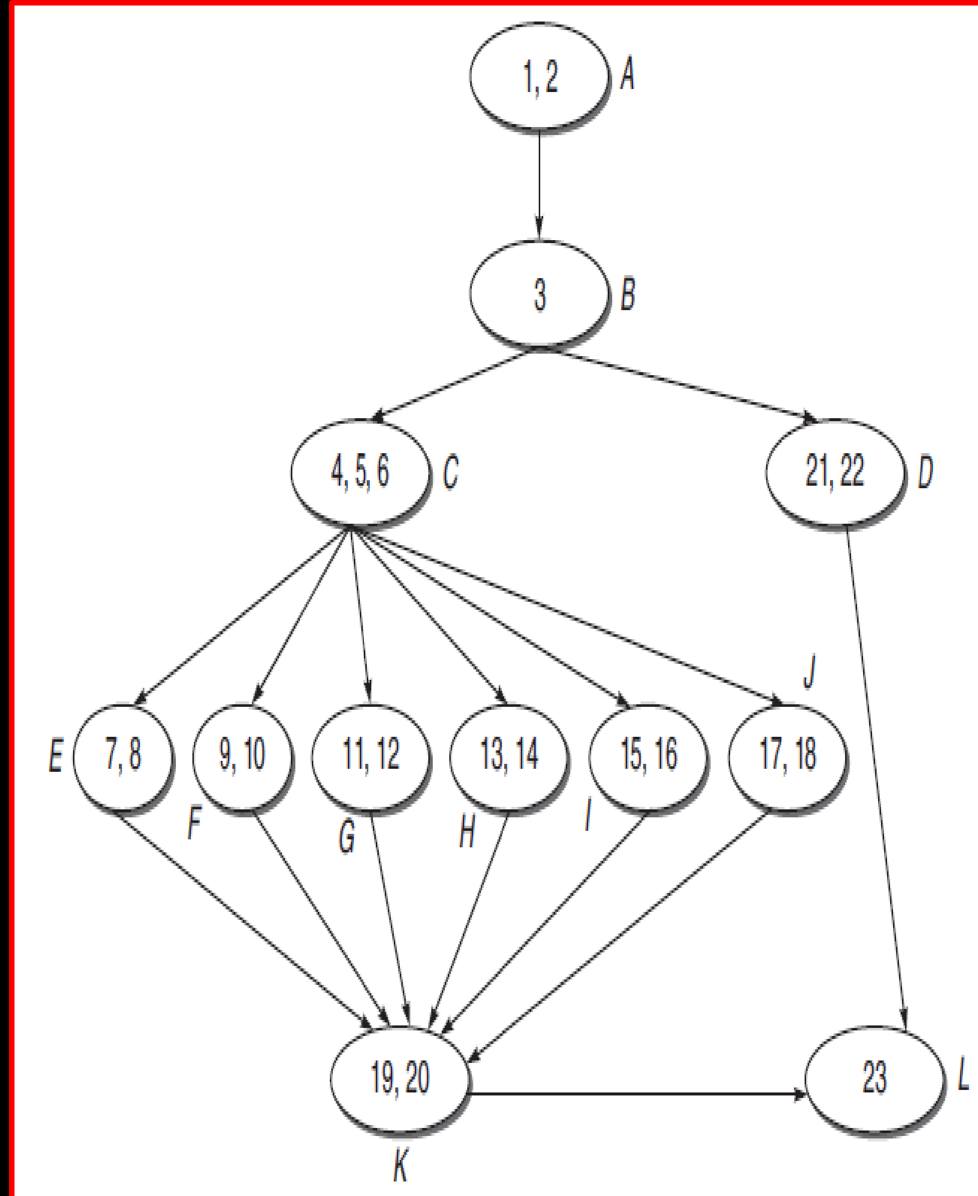
(b) Calculate the cyclomatic complexity of the program using all the methods.

(c) List all independent paths.

(d) Design test cases from independent paths.

DD graph

```
    main()
    {
        char chr;
1.      printf (“Enter the special character\n”);
2.      scanf (%c”, &chr);
3.      if (chr != 48) && (chr != 49) && (chr != 50) && (chr != 51) &&
           (chr != 52) && (chr != 53) && (chr != 54) && (chr != 55) &&
           (chr != 56) && (chr != 57)
4.      {
5.          switch(chr)
6.          {
7.          Case ‘*’: printf(“It is a special character”);
8.          break;
9.          Case ‘#’: printf(“It is a special character”);
10.         break;
11.         Case ‘@’: printf(“It is a special character”);
12.         break;
13.         Case ‘!’:  printf(“It is a special character”);
14.         break;
15.         Case ‘%’: printf(“It is a special character”);
16.         break;
17.         default : printf(“You have not entered a special character”);
18.         break;
19.         }// end of switch
20.     } // end of If
21.     else
22.         printf(“You have not entered a character”);
23. } // end of main()
```



Cyclomatic complexity

(i) V(G) = e – n + 2 * p
= 17 – 12 + 2*1
= 7

(ii) V(G) = Number of predicate nodes + 1
= 2 (Nodes B, C) + 1
= 7

Node C is a switch-case, so for finding out the number of predicate nodes for this case, follow the following formula:
Number of predicated nodes = Number of links out of main node –1
= 6 – 1 = 5 (For node C)

(iii) V (G) = Number of regions
= 7

Independent paths

Since the cyclomatic complexity of the graph is 7, there will be 7 independent paths in the graph as shown below:

(i) A-B-D-L

(ii) A-B-C-E-K-L

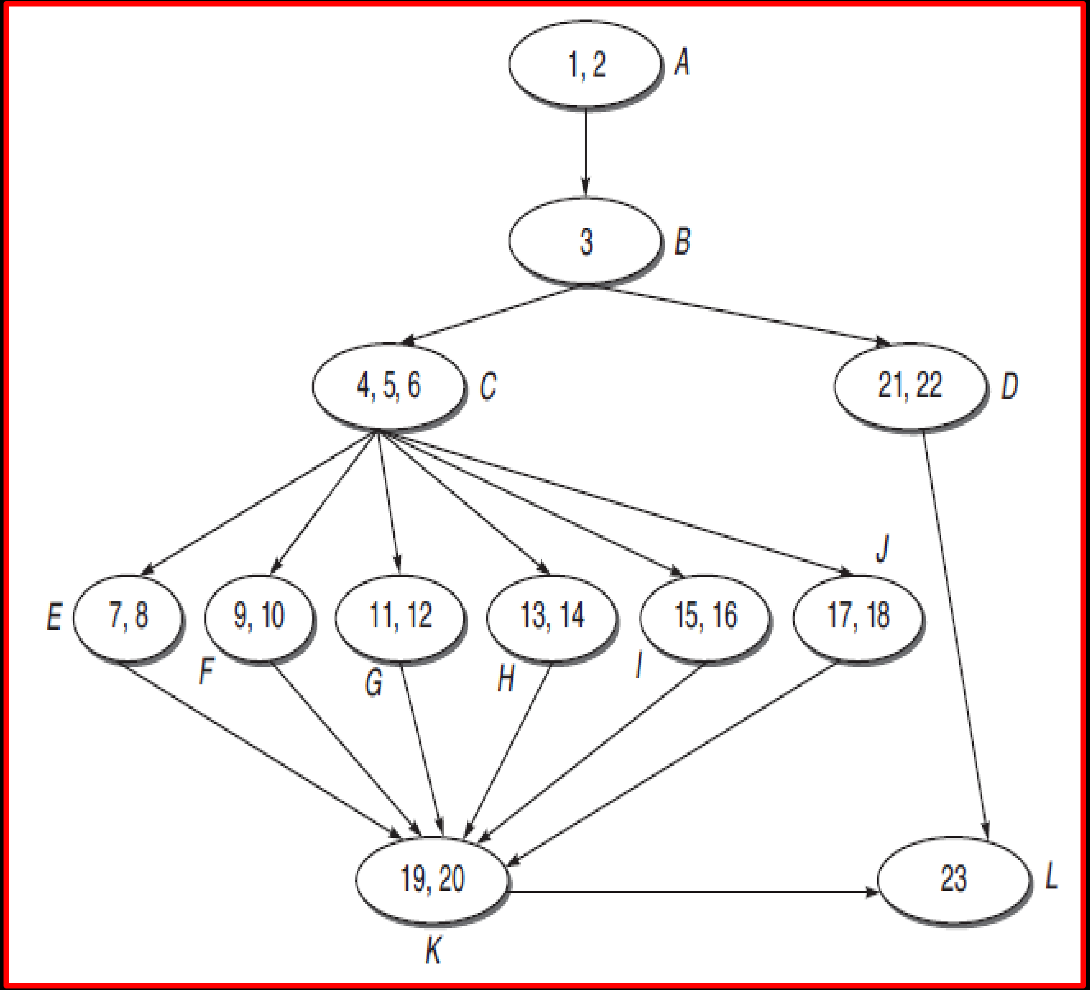
(iii) A-B-C-F-K-L

(iv) A-B-C-G-K-L

(v) A-B-C-H-K-L

(vi) A-B-C-I-K-L

(vii) A-B-C-J-K-L



Test case design

Test case design from the list of independent paths:

Test Case ID	Input Character	Expected Output	Independent path covered by Test Case
1	(	You have not entered a character	A-B-D-L
2	*	It is a special character	A-B-C-E-K-L
3	#	It is a special character	A-B-C-F-K-L
4	@	It is a special character	A-B-C-G-K-L
5	!	It is a special character	A-B-C-H-K-L
6	%	It is a special character	A-B-C-I-K-L
7	$	You have not entered a special character	A-B-C-J-K-L

EXAMPLE - 5

Consider a program to arrange numbers in ascending order from a given list of N numbers.

```
        main()
        {
        int num,small;
        int i,j,sizelist,list[10],pos,temp;
1.      clrscr();
        printf(“\nEnter the size of list :\n ”);
        scanf(“%d”,&sizelist);
2.      for(i=0;i<sizelist;i++)
        {
3.      printf(“\nEnter the number”);
        scanf (“%d”,&list[i]);
        }
4.      for(i=0;i<sizelist;i++)
        {
5.      small=list[i];
        pos=i;
6.      for(j=i+1;j<sizelist;j++)
        {
7.      if(small>list[j])
        {
        small=list[j];
8.      pos=j;
        }
9.      }
        temp=list[i];
10.     list[i]=list[pos];
        list[pos]=temp;
11.     }
12.     printf(“\nList of the numbers in ascending order : ”);
13.     for(i=0;i<sizelist;i++)
14.     printf(“\n%d”,list[i]);
        getch();
15.     }
```

BASIS PATH TESTING

(a) Draw the DD (Decision –To – Decision) graph for the program.

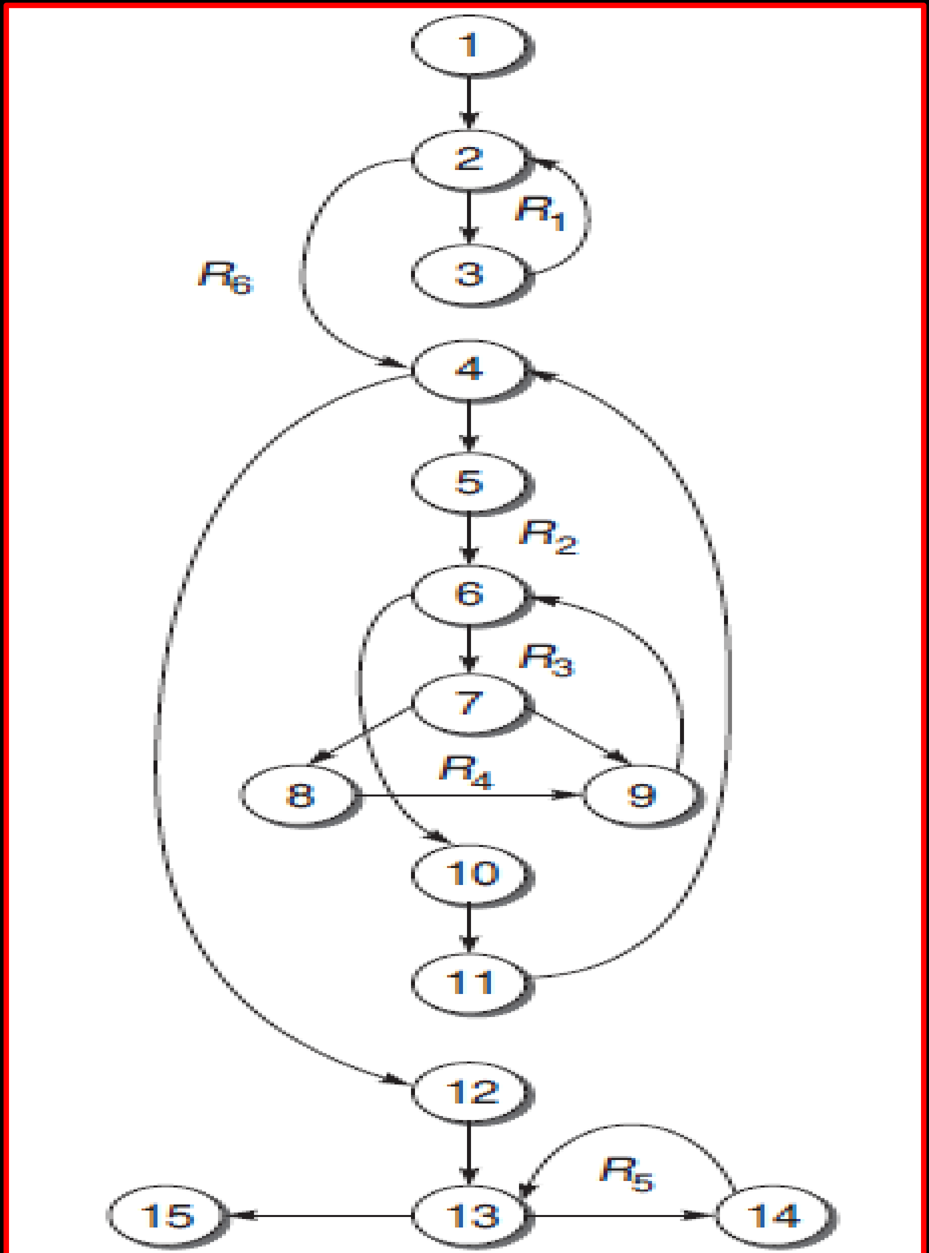
(b) Calculate the cyclomatic complexity of the program using all the methods.

(c) List all independent paths.

(d) Design test cases from independent paths.

DD graph

```
        main()
        {
        int num,small;
1.      int i,j,sizelist,list[10],pos,temp;
        clrscr();
        printf("\nEnter the size of list :\n ");
        scanf("%d",&sizelist);
2.      for(i=0;i<sizelist;i++)
        {
3.      printf("\nEnter the number");
        scanf ("%d",&list[i]);
        }
4.      for(i=0;i<sizelist;i++)
        {
5.      small=list[i];
        pos=i;
6.      for(j=i+1;j<sizelist;j++)
        {
7.      if(small>list[j])
        {
8.      small=list[j];
        pos=j;
        }
9.      }
        temp=list[i];
10.     list[i]=list[pos];
        list[pos]=temp;
11.     }
12.     printf("\nList of the numbers in ascending order : ");
13.     for(i=0;i<sizelist;i++)
14.     printf("\n%d",list[i]);
15.     getch();
        }
```



Cyclomatic complexity

(i) $V(G) = e - n + 2 * p$
$= 19 - 15 + 2*1$
$= 6$

(ii) V(G) = Number of predicate nodes + 1
= 5 + 1
= 6

(iii) V (G) = Number of regions
= 6

Independent paths

Since the cyclomatic complexity of the graph is 6, there will be 6 independent paths in the graph as shown below:

(i) 1-2-3-2-4-5-6-7-8-9-6-10-11-4-12-13-14-13-15

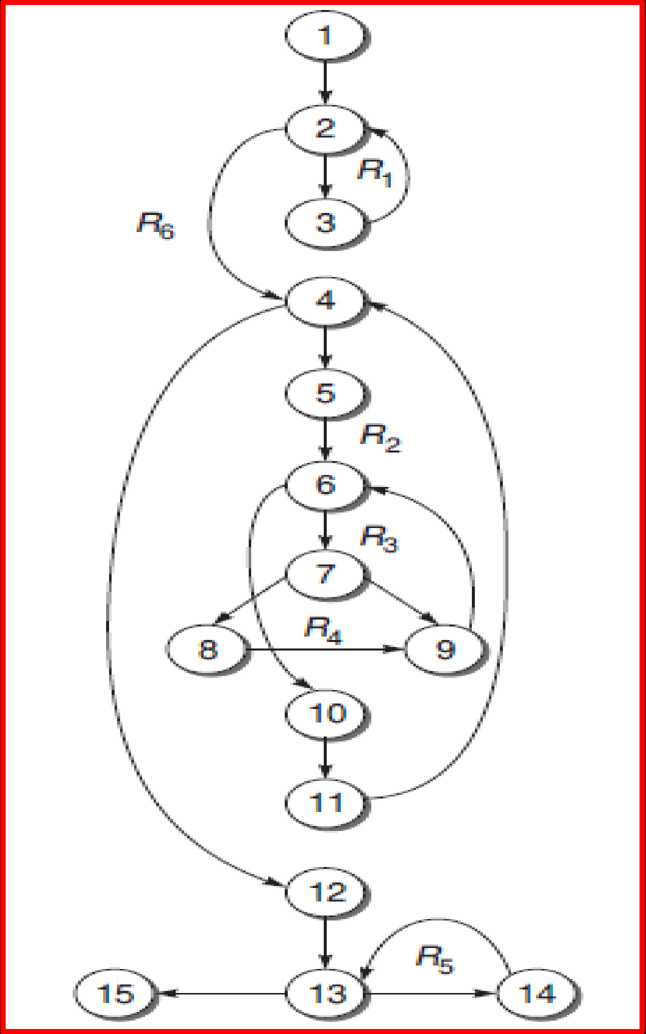
(ii) 1-2-3-2-4-5-6-7-9-6-10-11-4-12-13-14-13-15

(iii) 1-2-3-2-4-5-6-10-11-4-12-13-14-13-15

(iv) 1-2-3-2-4-12-13-14-13-15 (path not feasible)

(v) 1-2-4-12-13-15

(vi) 1-2-3-2-4-12-13-15 (path not feasible)



Test case design

Test case design from the list of independent paths:

Test Case ID	Input	Expected Output	Independent path covered by Test Case
1	Sizelist = 5 List[] = {17,6,7,9,1}	1,6,7,9,17	1
2	Sizelist = 5 List[] = {1,3,9,10,18}	1,3,9,10,18	2
3	Sizelist = 1 List[] = {1}	1	3
4	Sizelist = 0	blank	blank

EXAMPLE - 6

Consider the program for calculating the factorial of a number. It consists of main() program and the module fact(). Calculate the individual cyclomatic complexity number for main() and fact() and then, the cyclomatic complexity for the whole program.

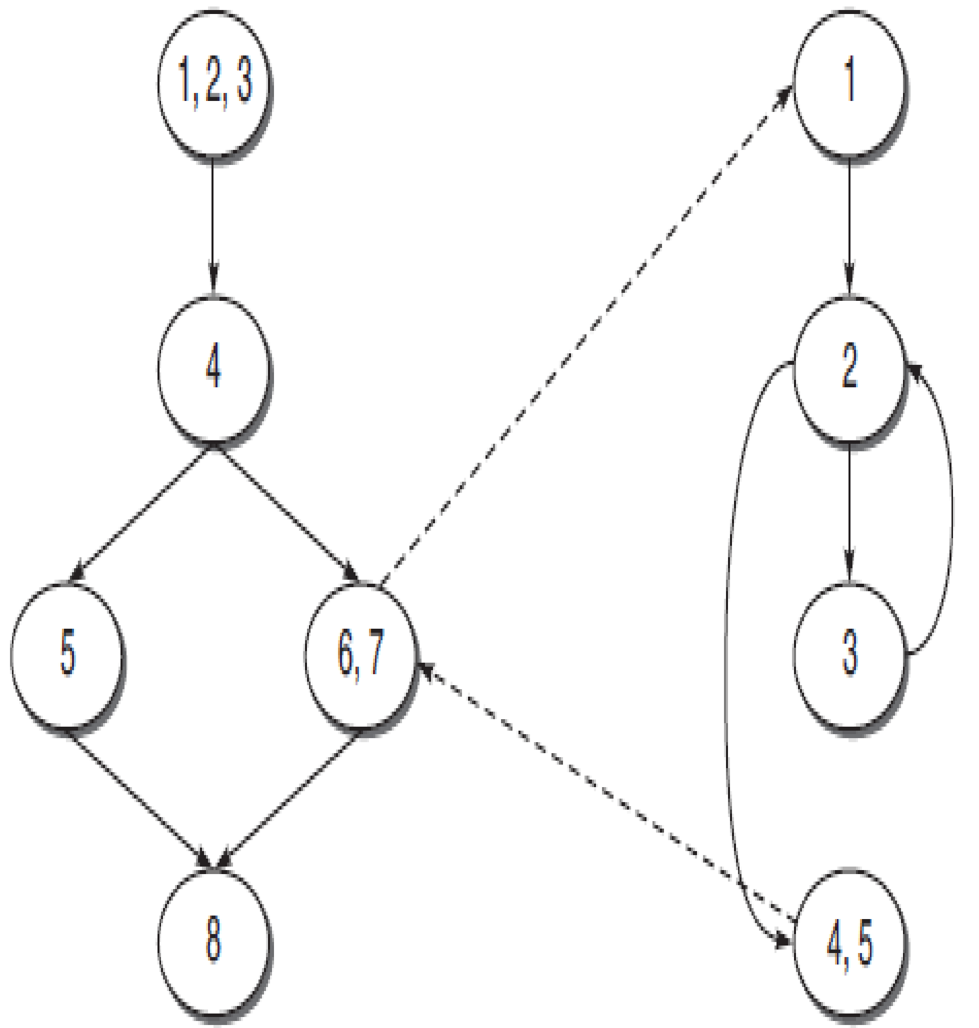
```
        main()
        {
              int number;
              int fact();
1.            clrscr();
2.            printf("Enter the number whose factorial is to be found out");
3.            scanf("%d", &number);
4.            if(number <0)
5.                printf("Facorial cannot be defined for this number);
6.            else
7.                printf("Factorial is %d", fact(number));
8.      }

        int fact(int number)
        {
              int index;
1.            int product =1;
2.            for(index=1; index<=number; index++)
3.                product = product * index;
4.            return(product);
5.      }
```

DD graph

```
    main()
    {
        int number;
        int fact();
1.      clrscr();
2.      printf("Enter the number whose factorial is to be found out");
3.      scanf("%d", &number);
4.      if(number <0)
5.          printf("Facorial cannot be defined for this number);
6.      else
7.          printf("Factorial is %d", fact(number));
8.  }

    int fact(int number)
    {
        int index;
1.      int product =1;
2.      for(index=1; index<=number; index++)
3.          product = product * index;
4.      return(product);
5.  }
```



(a) Flow graph for main ()

(b) Flow graph for fact ()

Cyclomatic complexity

Cyclomatic complexity of main()

(i) $V(M) = e - n + 2 * p$
$= 5 - 5 + 2$
$= 2$

(ii) V(M) = Number of predicate nodes + 1
= 1 + 1
= 2

(iii) V (M) = Number of regions
= 2

Cyclomatic complexity of fact()

(i) $V(R) = e - n + 2 * p$
$= 4 - 4 + 2$
$= 2$

(ii) V(R) = Number of predicate nodes + 1
= 1 + 1
= 2

(iii) V (R) = Number of regions
= 2

Cyclomatic complexity

Cyclomatic complexity of the whole graph considering the full program:

(i) $V(G) = e - n + 2 * p$
$= 9 - 9 + 2 * 2$
$= 4$
$= V(M) + V(R)$

(ii) $V(G) = d + p$
$= 2 + 2$
$= 4$
$= V(M) + V(R)$

(iii) $V(G) =$ Number of regions
$= 4$
$= V(M) + V(R)$

GRAPH MATRICES

GRAPH MATRICES

Flow graph is an **effective aid in path testing** as seen in the previous section.

As the size of graph increases, manual path tracing becomes difficult and leads to errors. [i.e. A link can be missed or covered twice].

Graph matrix, a data structure, is the solution which can assist in developing a tool for automation of path tracing.

GRAPH MATRICES

GRAPH MATRIX:

It is a square matrix whose rows and columns are equal to the number of nodes in the flow graph.

Each row and column identifies a particular node.

Matrix entries represent a connection between the nodes.

GRAPH MATRICES

The following points describe a graph matrix:

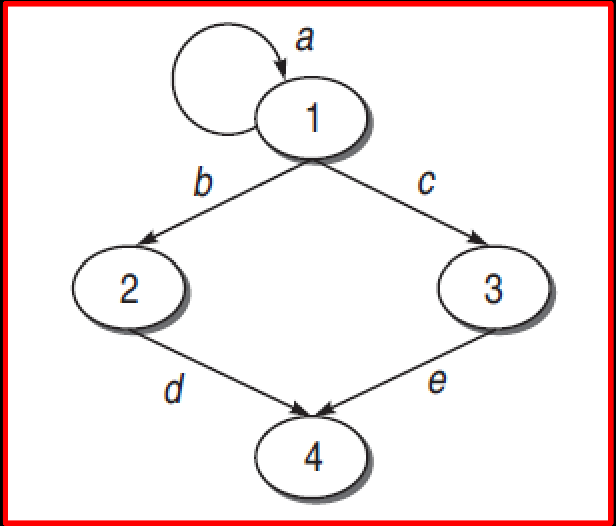
(i) Each cell in the matrix can be a direct connection or link between one node to another node.

(ii) If there is a connection from node ‘a’ to node ‘b’, then it does not mean that there is connection from node ‘b’ to node ‘a’.

(iii) Conventionally, to represent a graph matrix, digits are used for nodes and letter symbols for edges or connections.

EXAMPLE - 1

Consider the below graph and represent it in the form of a graph matrix.

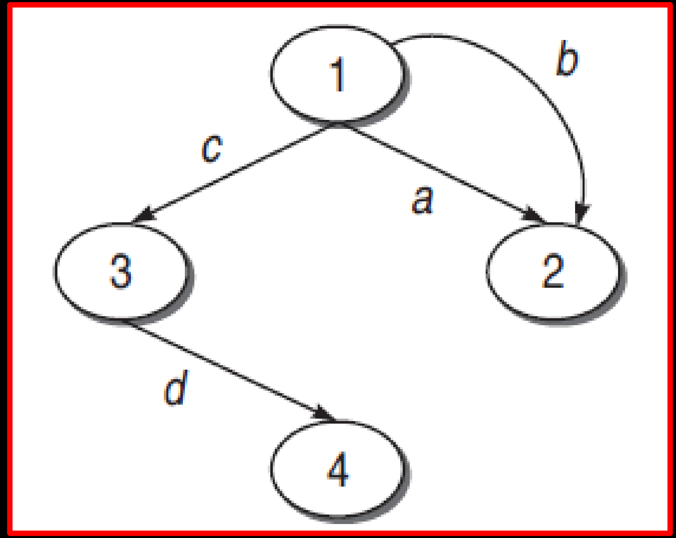


Solution: The graph matrix is shown below.

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

EXAMPLE - 2

Consider the below graph and represent it in the form of a graph matrix.



Solution: The graph matrix is shown below.

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

CONNECTION MATRIX

CONNECTION MATRIX

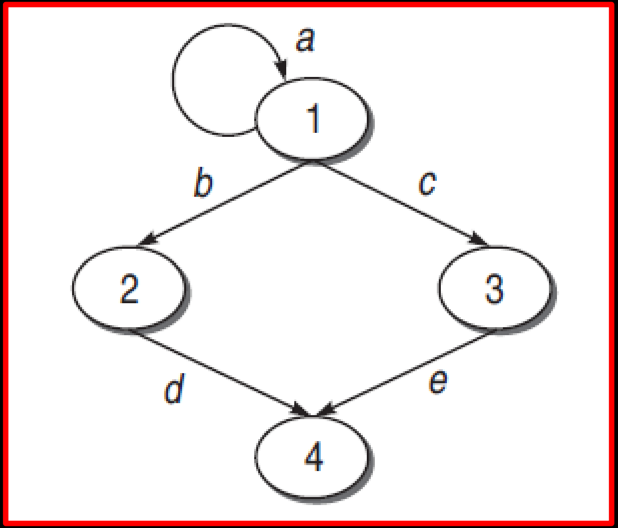
A **matrix** defined with **link weights** is called a **connection matrix**.

If we add **link weights** to **each cell entry**, then **graph matrix** can be used as a **powerful tool** in testing.

In the simplest form, when the **connection exists**, then the **link weight is 1, otherwise 0**.

CONNECTION MATRIX (EXAMPLE – 1)

Consider the below graph and represent it in the form of a Connection matrix.

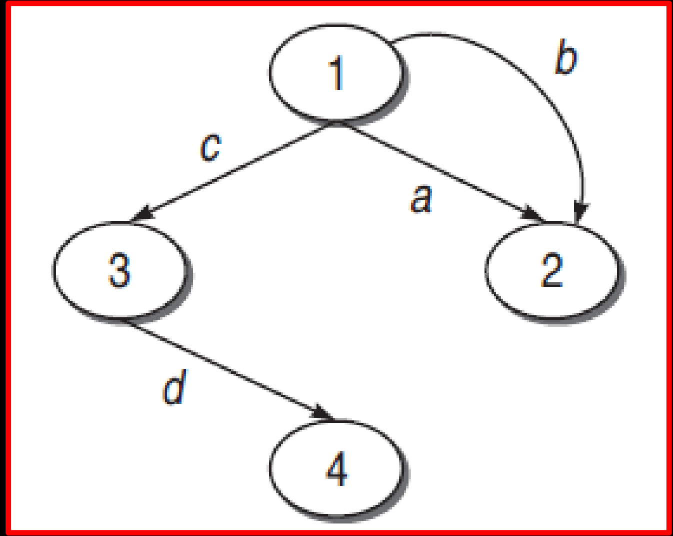


Solution: The connection matrix is shown below.

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

CONNECTION MATRIX (EXAMPLE – 2)

Consider the below graph and represent it in the form of a Connection matrix.



Solution: The connection matrix is shown below.

	1	2	3	4
1		1	1	
2				
3				1
4				

CONNECTION MATRIX- cyclomatic number

Procedure to find the Cyclomatic number from the connection matrix:

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

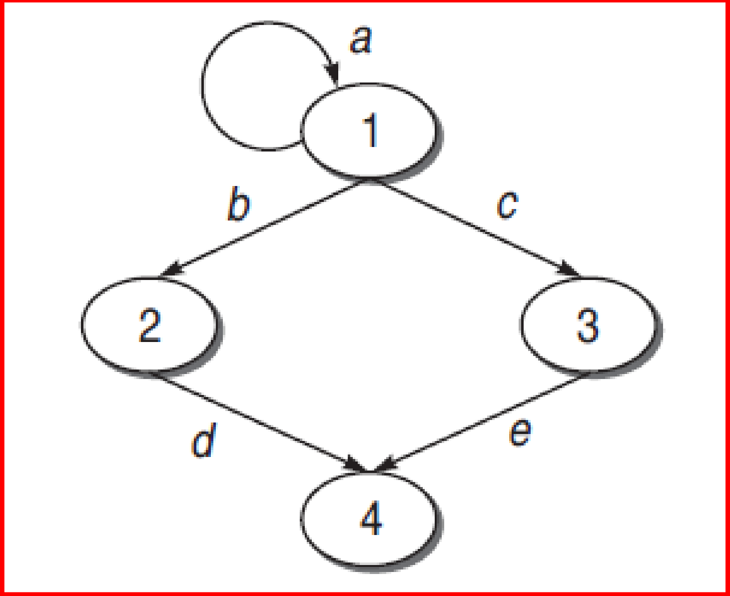
Step 2: Subtract 1 from that count. Ignore the blank rows, if any.

Step 3: Add the final 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.

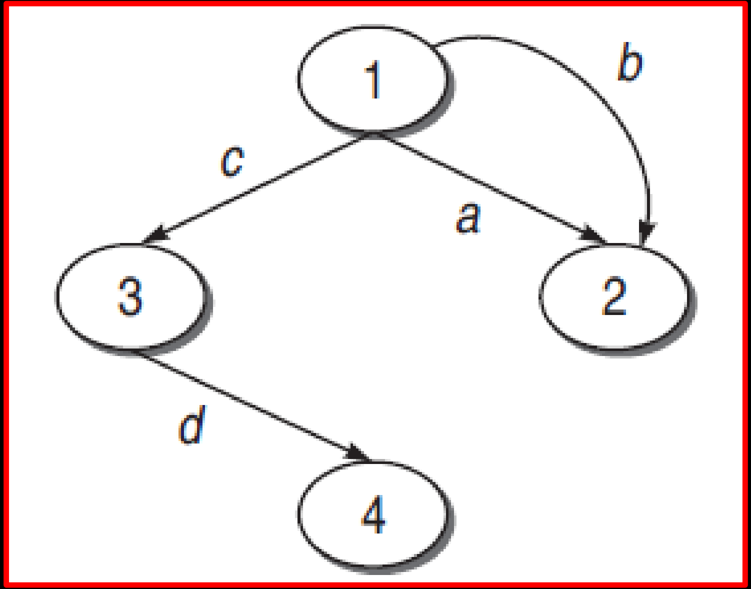
CONNECTION MATRIX - cyclomatic number



Solution: The cyclomatic number calculated from the connection matrix shown below:

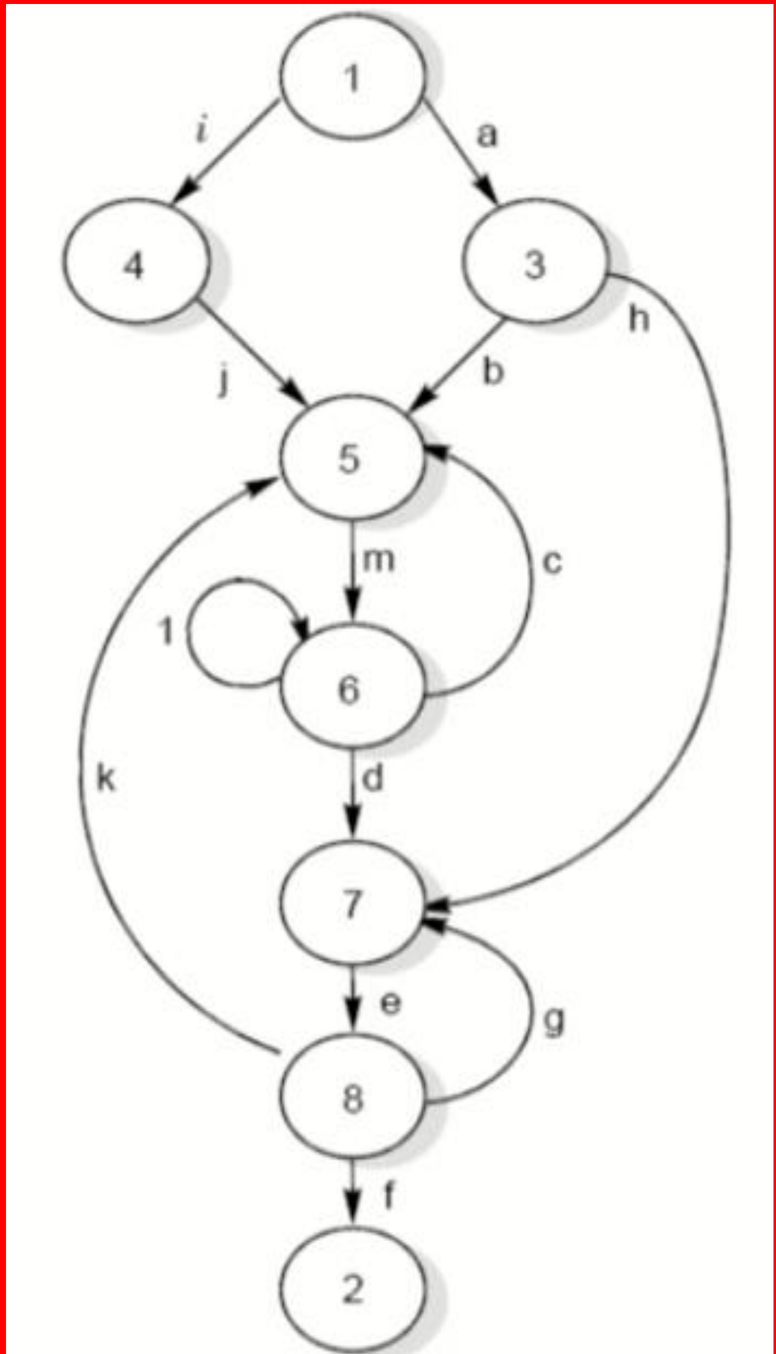
	1	2	3	4	
1	1	1	1		3 – 1 = 2
2				1	1 – 1 = 0
3				1	1 – 1 = 0
4					
	Cyclomatic number = 2+1 = 3				

CONNECTION MATRIX - cyclomatic number



Solution: The cyclomatic number calculated from the connection matrix shown below:

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



	1	2	3	4	5	6	7	8
1			a	i				
2								
3					b		h	
4					j			
5						m		
6					c	l	d	
7								e
8		f			k		g	

Graph Matrix

Graph Matrix – To find set of all paths

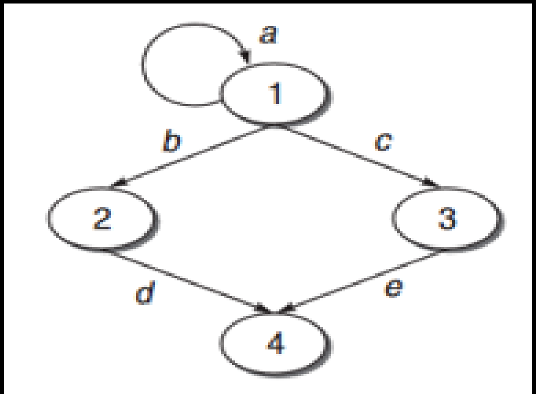
The set of all paths between all nodes is easily expressed in terms of matrix operations.

For example, the square of matrix represents path segments that are 2-links long.

The cube power of matrix represents path segments that are 3-links long.

GRAPH MATRIX (EXAMPLE)

Consider the graph & its graph matrix below and find 2-link paths for each node.



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

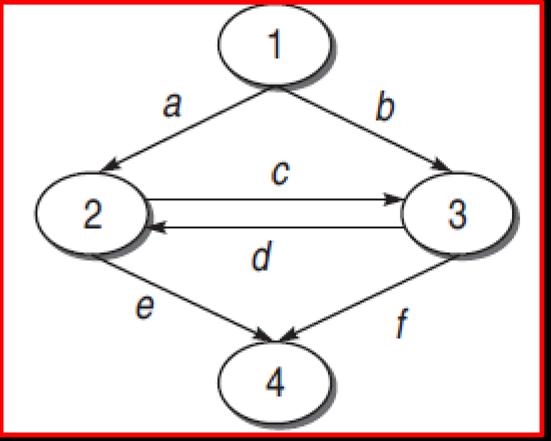
Solution: For finding 2-link paths, we should square the matrix. (Squaring the matrix yields a new matrix having 2-link paths.)

$$\begin{pmatrix} a & b & c & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & e \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} a & b & c & 0 \\ 0 & 0 & 0 & d \\ 0 & 0 & 0 & e \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} a^2 & ab & ac & bd + ce \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

The resulting matrix shows all the 2-link paths from one node to another. For example, from node 1 to node 2, there is one 2-link, i.e., ab.

GRAPH MATRIX (EXAMPLE)

Consider the following graph. Derive its graph matrix and find 2-link and 3-link set of paths.



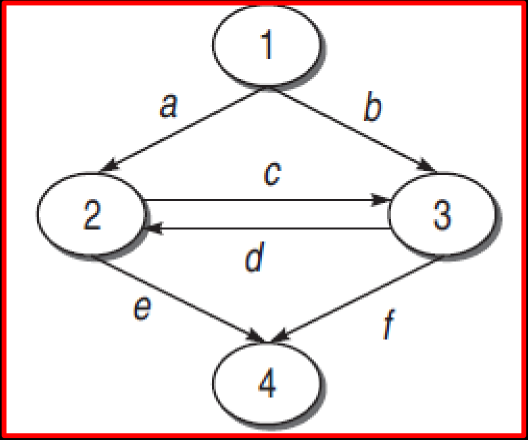
Solution: The graph matrix of the graph is shown below.

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

GRAPH MATRIX (EXAMPLE)

First we find 2-link set of paths by squaring this matrix as shown below:

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



Next, we find 3-link set of paths by taking the cube of matrix as shown below:

$$\begin{pmatrix} 0 & bd & ac & ae + bf \\ 0 & cd & 0 & cf \\ 0 & 0 & dc & de \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & a & b & 0 \\ 0 & 0 & c & e \\ 0 & d & 0 & f \\ 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & acd & bdc & bde + acf \\ 0 & 0 & cdc & cde \\ 0 & dcd & 0 & dcf \\ 0 & 0 & 0 & 0 \end{pmatrix}$$