SE-3002
SOFTWARE QUALITY ENGINEERING

RUBAB JAFFAR
RUBAB.JAFFAR@NU.EDU.PK

Part II-Software Testing

Structural Testing

TODAY'S OUTLINE

- Structural Testing
- Control Flow Testing
  - Branch testing
  - Statement testing
  - Condition testing
  - Path testing
- Data Flow Testing

STRUCTURAL TESTING

- More technical than functional testing.
- It attempts to design test cases from the source code and not from the specifications.
- The source code becomes the base document which is examined thoroughly in order to understand the internal structure and other implementation details.
- Structural testing techniques are also known as white box testing techniques
- Many structural testing techniques are available
  - control flow testing,
  - data flow testing,
  - slice based testing and
  - mutation testing.

CONTROL FLOW TESTING

- Identify paths of the program and write test cases to execute those paths. PATHS?
- There may be too many paths in a program and it may not be feasible to execute all of them. As the number of decisions increase in the program, the number of paths also increase accordingly.
- Every path covers a portion of the program. We define ‘coverage’ as a ‘percentage of source code that has been tested with respect to the total source code available for testing’.
- Write test cases to achieve a reasonable level of coverage using control flow testing.
- The most reasonable level may be to test every statement of a program at least once before the completion of testing.
- Testing techniques based on program coverage criterion may provide an insight about the effectiveness of test cases.

CONTROL FLOW TESTING

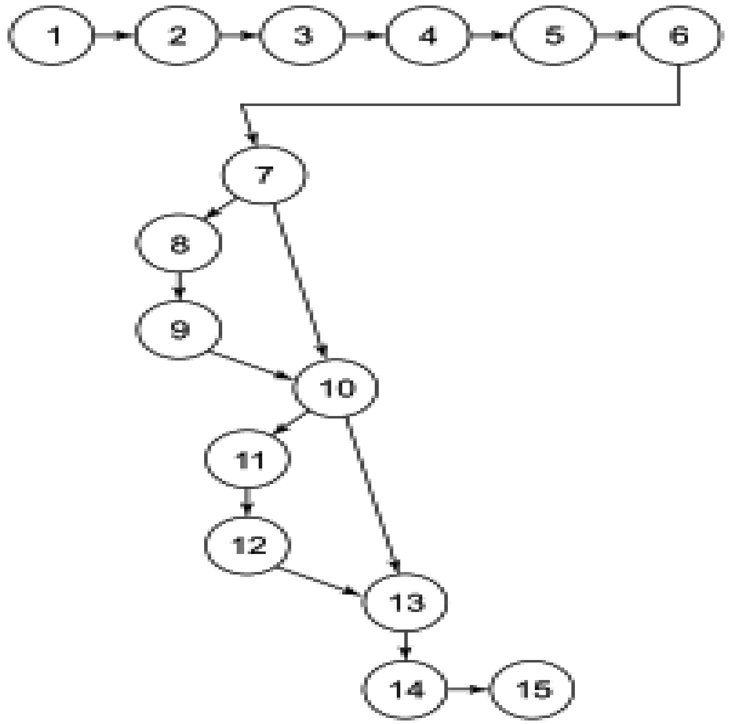
- Some of such techniques are discussed which are part of control flow testing.
- Statement Coverage
- Branch Coverage
- Condition Coverage

STATEMENT COVERAGE

- We want to execute every statement of the program in order to achieve 100% statement coverage.
- Consider the following portion of a source code along with its program graph.

```
        #include<stdio.h>
        #include<conio.h>

1.      void main()
2.      {
3.      int a,b,c,x=0,y=0;
4.      clrscr();
5.      printf("Enter three numbers:");
6.      scanf("%d %d %d",&a,&b,&c);
7.      if((a>b)&&(a>c)){
8.                  x=a*a+b*b;
9.      }
10.     if(b>c){
11.                 y=a*a-b*b;
12.     }
13.     printf("x= %d y= %d",x,y);
14.     getch();
15.     }
```



TEST CASE

- a=9, b=8, c=7, all statements are executed and we have achieved 100% statement coverage by only one test case. The total paths of this program graph are given as:
  - 1–7, 10–15
  - 1–7, 10, 13–15
  - 1–10, 13–15
  - 1–15
- The cyclomatic complexity of this graph is:
- $V(G) = e - n + 2P = 16 - 15 + 2 = 3$
- Hence, independent paths are three and are given as:
  - 1–7, 10, 13–15
  - 1–10, 13–15
  - 1–7, 10–15
- Only one test case may cover all statements but will not execute all possible four paths and not even cover all independent paths.

BRANCH COVERAGE

- We want to test every branch of the program. Hence, we wish to test every 'True' and 'False' condition of the program.
- If we select a = 9, b = 8, c = 7, we achieve 100% statement coverage and the path followed is given as (all true conditions): Path = 1–15
- We also want to select all false conditions with the following inputs:
- a = 7, b = 8, c = 9, the path followed is Path = 1–7, 10, 13–15
- These two test cases out of four are sufficient to guarantee 100% branch coverage. The branch coverage does not guarantee 100% path coverage but it does guarantee 100% statement coverage.

CONDITION COVERAGE

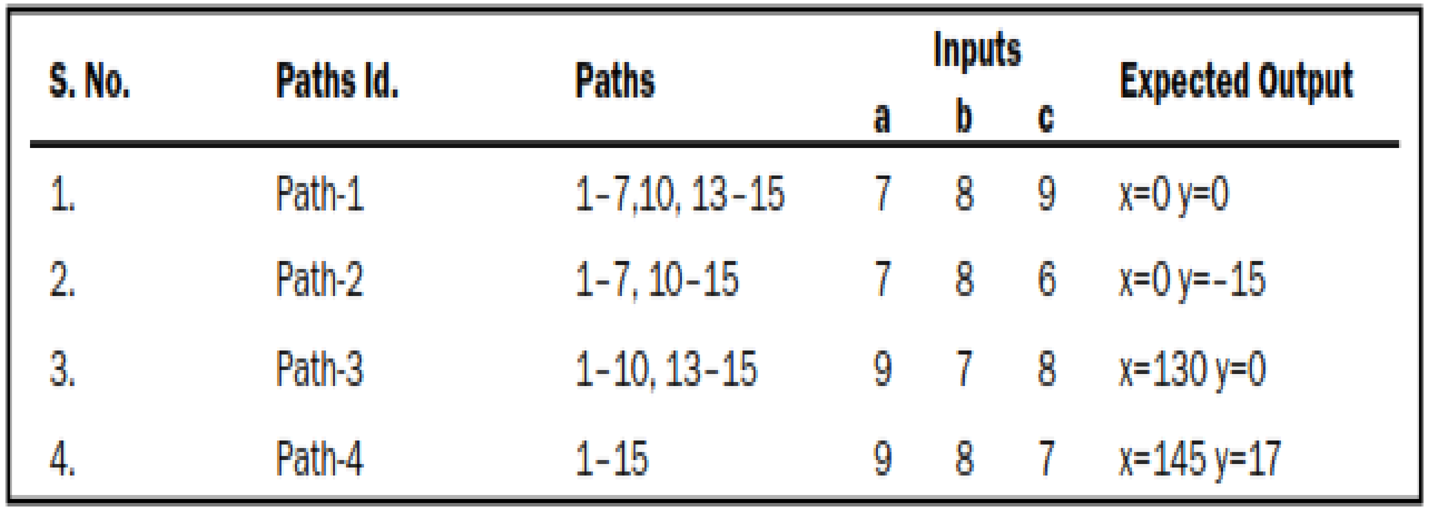
- Condition coverage is better than branch coverage because we want to test every condition at least once. However, branch coverage can be achieved without testing every condition.
- Considering the example on slide 6, statement number 7 has two conditions (a>b) and (a>c). There are four possibilities namely:
  - First is true, second is false
  - Both are true
  - First is false, second is true
  - Both are false
- If a > b and a > c, then the statement number 7 will be true (first possibility). However, if a < b, then second condition (a > c) would not be tested and statement number 7 will be false (third and fourth possibilities). If a > b and a < c, statement number 7 will be false (second possibility). Hence, we should write test cases for every true and false condition. Selected inputs may be given as:
- a = 9, b = 8, c = 10 (second possibility – first is true, second is false)
- a = 9, b = 8, c = 7 (first possibility when both are true)
- a = 7, b = 8, c = 9 (third and fourth possibilities- first is false, statement number 7 is false)
- Hence, these three test cases out of four are sufficient to ensure the execution of every condition of the program.

PATH COVERAGE

- In this coverage criteria, we want to test every path of the program. There are too many paths in any program due to loops and feedback connections. It may not be possible to achieve this goal of executing all paths in many programs. If we do so, we may be confident about the correctness of the program. If it is unachievable, at least all independent paths should be executed.
- 1–7, 10–15
- 1–7, 10, 13–15
- 1–10, 13–15
- 1–15

TEST CASES FOR ALL PATHS

- Execution of all these paths increases confidence about the correctness of the program. Inputs for test cases are given as:

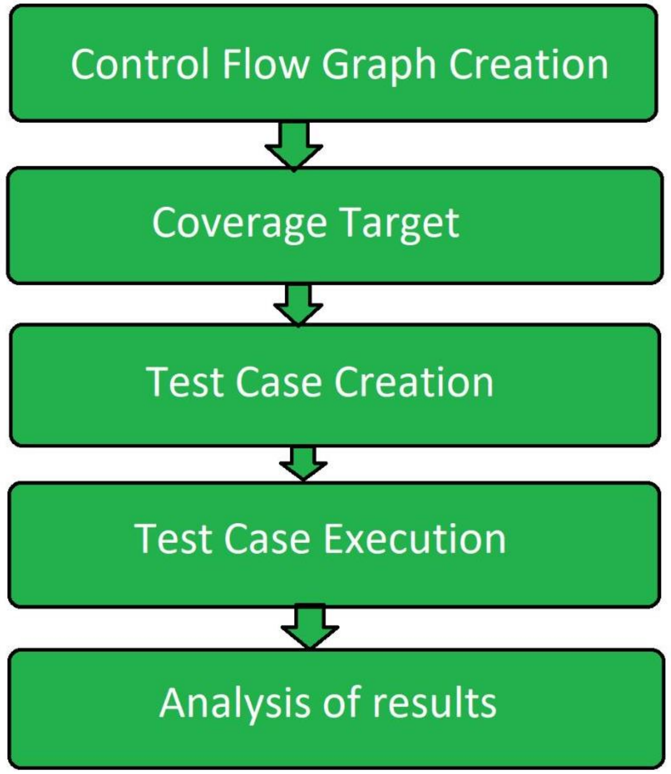


S. No.	Paths Id.	Paths	Inputs			Expected Output
			a	b	c	
1.	Path-1	1-7,10, 13-15	7	8	9	x=0 y=0
2.	Path-2	1-7, 10-15	7	8	6	x=0 y=-15
3.	Path-3	1-10, 13-15	9	7	8	x=130 y=0
4.	Path-4	1-15	9	8	7	x=145 y=17

PATH TESTING

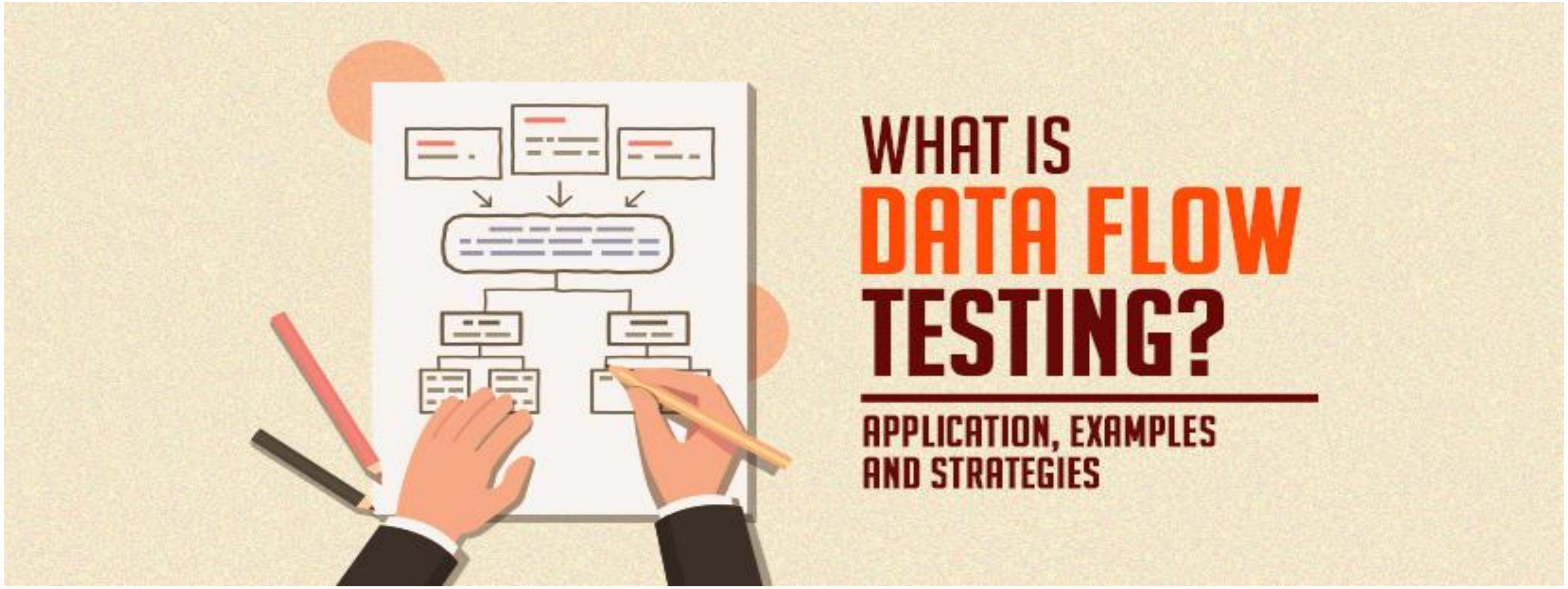
- Path testing guarantee statement coverage, branch coverage and condition coverage.

PATH TESTING



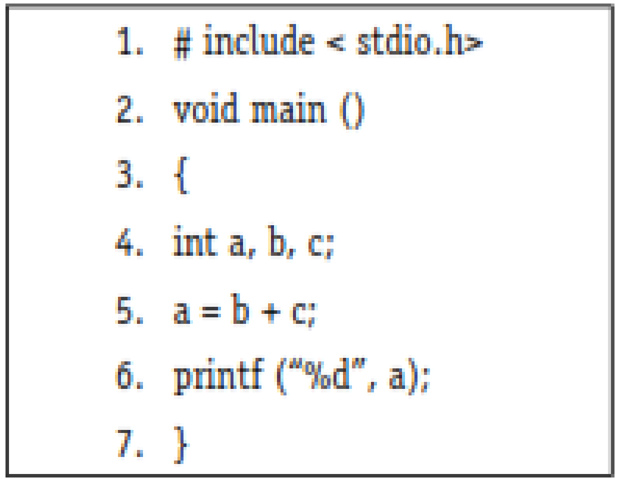
EXAMPLE: PERFORM PATH TESTING

```
public static void search ( int key, int 0elemArray, Result r ) {
1. int bottom =0 ;
2. int top =elemArray.length - 1 ; int mid;
3. r.found =false ;
4. r.index =-1 ;
5. while ( bottom <= top ) {
6 mid =(top + bottom) / 2 ;
7 if (elemArray [mid] = key) {
8 rindex = mid;
9 r.found =true ;
10 return ; } // if part
else {
11 if (elemArray [mid] < key)
12 bottom = mid + 1 ;
else
13 top = mid - 1 ; }
} //while loop
14. } //search
```


WHAT IS
DATA FLOW
TESTING?
APPLICATION, EXAMPLES
AND STRATEGIES

DATA FLOW TESTING

- In control flow testing, we find various paths of a program and design test cases to execute those paths.
- We may like to execute every statement of the program at least once before the completion of testing.
- Consider the following program:



```
# include < stdio.h>
void main ()
{
int a, b, c;
a = b + c;
printf ("%d", a);
}
```

DATA FLOW TESTING

- Data flow testing may help us to minimize such mistakes. It is done to cover the path testing and branch testing gap.
- It has nothing to do with dataflow diagrams. It is based on variables, their usage and their definition(s) (assignment) in the program.
- The main points of concern are:
  - Statements where these values are used (referenced).
  - Statements where variables receive values (definition).
- Data flow testing focuses on variable definition and variable usage.
- The process is conducted to detect the bugs because of the incorrect usage of data variables or data values.

DEFINE/REFERENCE ANOMALIES

- Some of the define / reference anomalies are given as:
  - A variable is defined but never used / referenced.
  - A variable is used but never defined.
  - A variable is defined twice before it is used.
  - A variable is used before even first-definition.
- Define / reference anomalies may be identified by static analysis of the program.

DATA FLOW TESTING TERMS DEFINITIONS

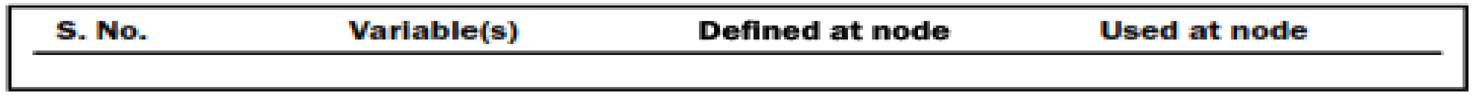
- A program is first converted into a program graph.
- Defining node
- A node of a program graph is a defining node for a variable , if and only if, the value of the variable is defined in the statement corresponding to that node. It is represented as DEF (, n) where is the variable and n is the node corresponding to the statement in which is defined.
- Usage node
- A node of a program graph is a usage node for a variable , if and only if, the value of the variable is used in the statement corresponding to that node. It is represented as USE (, n), where ‘ ’ is the variable and ‘n’ in the node corresponding to the statement in which ‘ ’ is used.
- A usage node USE (, n) is a predicate use node (denoted as P-use), if and only if, the statement corresponding to node ‘n’ is a predicate statement otherwise USE (, n) is a computation use node (denoted as C-use).

DATA FLOW TESTING TERMS DEFINITIONS

- Definition use Path
- A definition use path (denoted as du-path) for a variable ‘ ’ is a path between two nodes ‘m’ and ‘n’ where ‘m’ is the initial node in the path but the defining node for variable ‘ ’ (denoted as DEF (, m)) and ‘n’ is the final node in the path but usage node for variable ‘ ’ (denoted as USE (, n)).

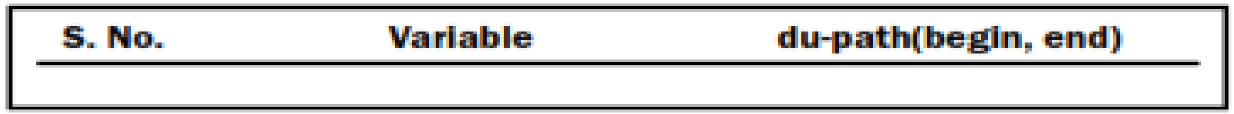
IDENTIFICATION OF DU PATHS

- The various steps for the identification of du and dc paths are given as:
- Draw the program graph of the program.
- Find all variables of the program and prepare a table for define / use status of all variables using the following format:



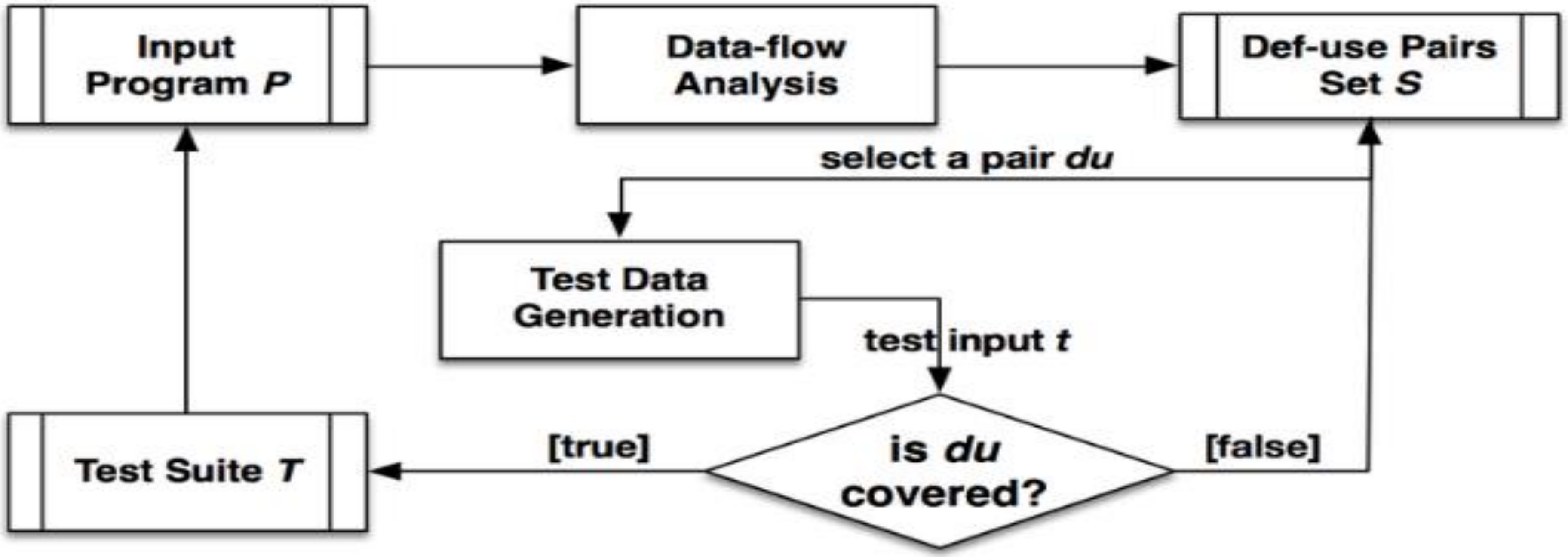
S. No.	Variable(s)	Defined at node	Used at node

- Generate all du-paths from define/use variable table of above step using the following



S. No.	Variable	du-path(begin, end)

DATA FLOW TESTING CYCLE



TESTING STRATEGIES USING DU-PATHS

- We want to generate test cases which trace every definition to each of its use and every use is traced to each of its definition. Some of the testing strategies are given as:
- **Test all du-paths**
  - All du-paths generated for all variables are tested. This is the strongest data flow testing strategy covering all possible du-paths.
- **Test all uses**
  - Find at least one path from every definition of every variable to every use of that variable which can be reached by that definition.
  - For every use of a variable, there is a path from the definition of that variable to the use of that variable.

TESTING STRATEGIES USING DU-PATHS

- **Test all definitions**
  - Find paths from every definition of every variable to at least one use of that variable;
- The first requires that each definition reaches all possible uses through all possible du-paths, the second requires that each definition reaches all possible uses, and the third requires that each definition reaches at least one use.

TYPES OF DATA FLOW TESTING

- Static Data Flow Testing
- No actual execution of the code is carried out in Static Data Flow testing. Generally, the definition, and usage pattern of the data variables is scrutinized through a control flow graph.
- Dynamic Data Flow Testing
- The code is executed to observe the transitional results. Dynamic data flow testing includes:
  - Identification of definition and usage of data variables.
  - Identifying viable paths between definition and usage pairs of data variables.
  - Designing & crafting test cases for these paths.

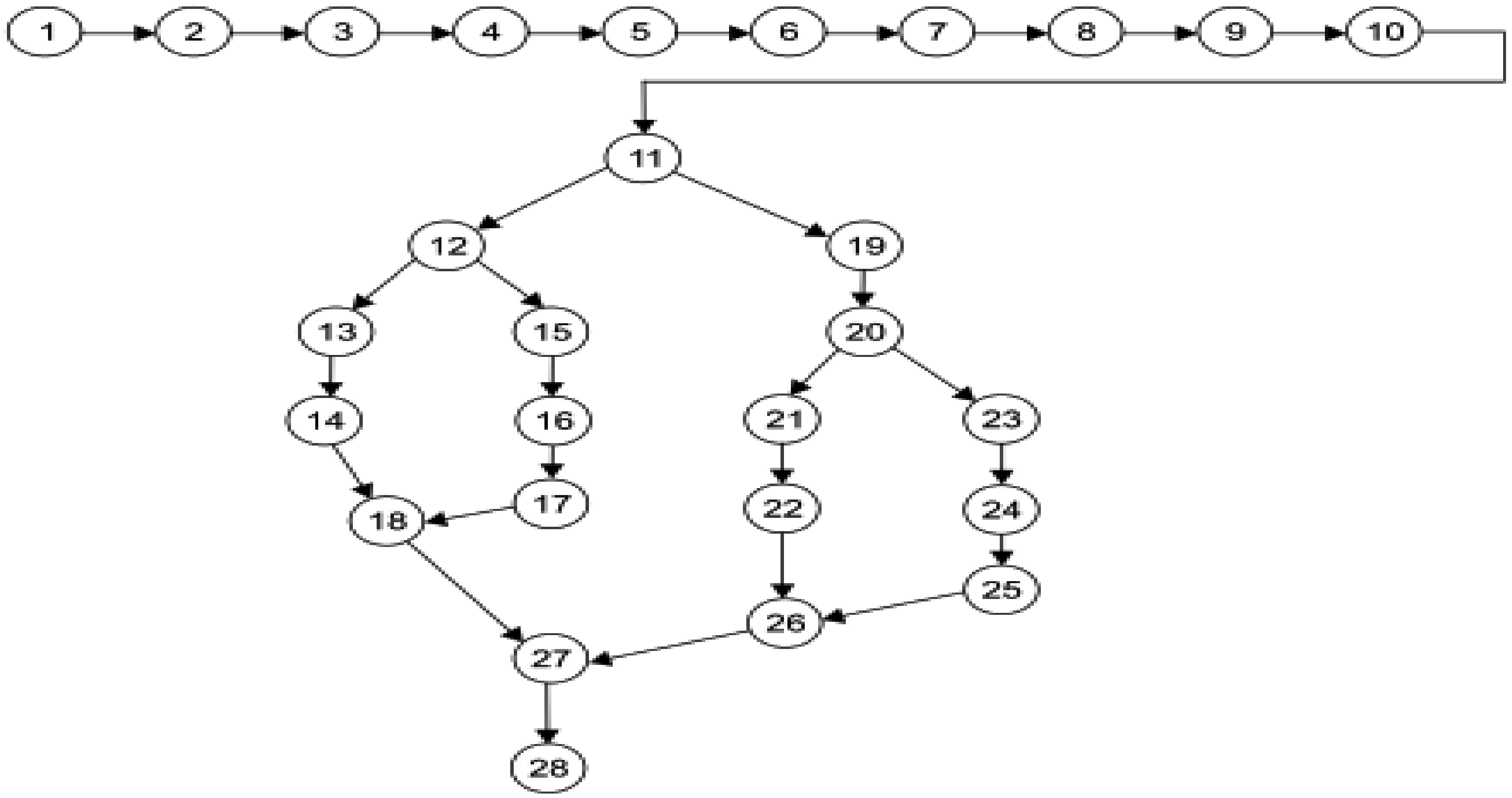
DATA FLOW TESTING LIMITATIONS

- Testers require good knowledge of programming.
- Time-consuming
- Costly process.

EXAMPLE: FIND THE LARGEST NUMBER AMONGST THREE NUMBERS.

```
            #include<stdio.h>
            #include<conio.h>
1.          void main()
2.          {
3.          float A,B,C;
4.          clrscr();
5.          printf("Enter number 1:\n");
6.          scanf("%f", &A);
7.          printf("Enter number 2:\n");
8.          scanf("%f", &B);
9.          printf("Enter number 3:\n");
10.         scanf("%f", &C);
            /*Check for greatest of three numbers*/
11.         if(A>B) {
12.         if(A>C) {
13.                 printf("The largest number is: %f\n",A);
14.                 }
15.         else  {
16.                 printf("The largest number is: %f\n",C);
17.                 }
18.         }
19.         else {
20.         if(C>B) {
21.                 printf("The largest number is: %f\n",C);
22.                 }
23.         else  {
24.                 printf("The largest number is: %f\n",B);
25.                 }
26.         }
27.         getch();
28.         }
```

STEP I



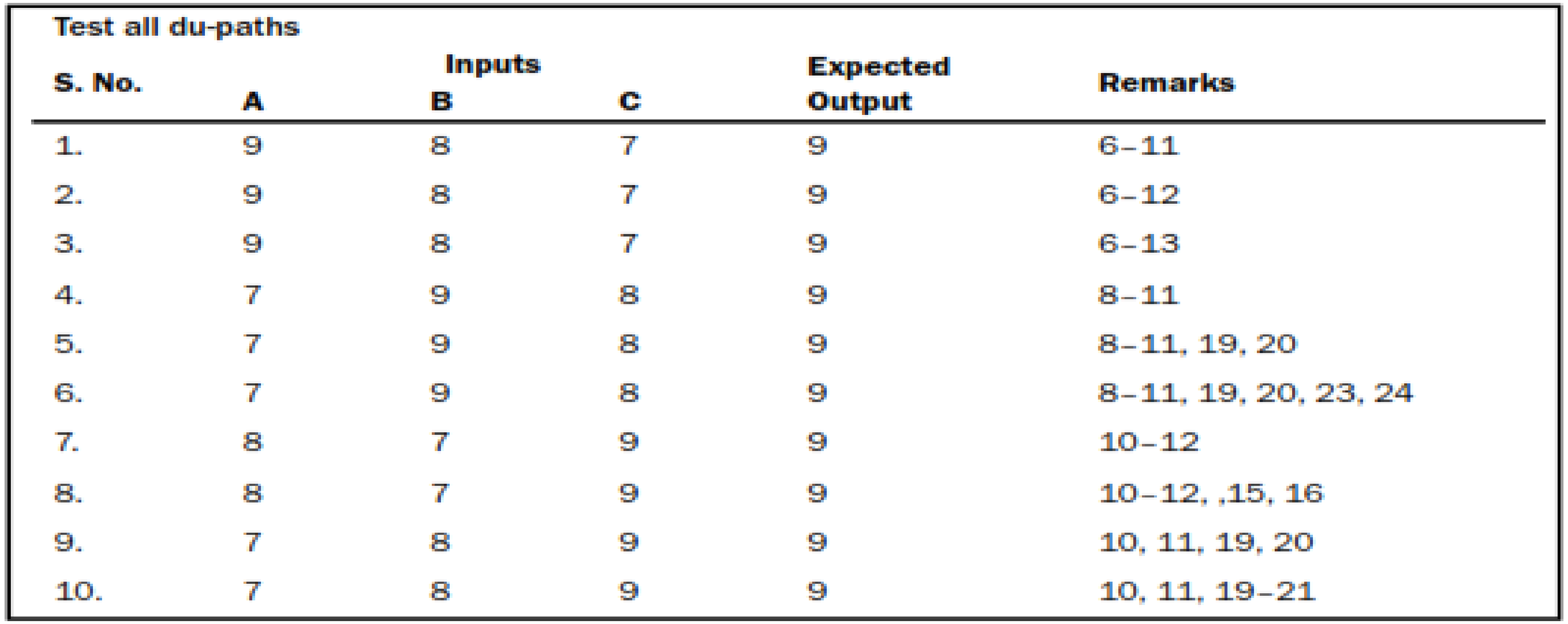
STEP II & III

S. No.	Variable	Defined at node	Used at node
1.	A	6	11, 12, 13
2.	B	8	11, 20, 24
3.	C	10	12, 16, 20, 21

The du-paths with beginning node and end node are given as:

Variable	du-path (Begin, end)
A	6, 11
	6, 12
	6, 13
B	8, 11
	8, 20
	8, 24
C	10, 12
	10, 16
	10, 20
	10, 21

TEST CASES



Test all du-paths

S. No.	Inputs A	Inputs B	Inputs C	Expected Output	Remarks
1.	9	8	7	9	6–11
2.	9	8	7	9	6–12
3.	9	8	7	9	6–13
4.	7	9	8	9	8–11
5.	7	9	8	9	8–11, 19, 20
6.	7	9	8	9	8–11, 19, 20, 23, 24
7.	8	7	9	9	10–12
8.	8	7	9	9	10–12, ,15, 16
9.	7	8	9	9	10, 11, 19, 20
10.	7	8	9	9	10, 11, 19–21

CLASS TASK

- Consider the following program:

```
static int find (int list[], int n, int key)
{
// binary search of ordered list
int lo = 0, mid;
int hi = n - 1;
int result = -1;
while ((hi >= lo) && (result == -1)) {
mid = (lo + hi) / 2;
if (list[mid] == key)
    result = mid;
else if (list[mid] > key)
    hi = mid - 1;
else // list[mid] < key
    lo = mid + 1;
}
return result;
}
```

ACTIVITIES

1. (10 Minutes) First individually construct the flow graph corresponding to this program.

2. (5 Minutes) Find a partner to work with in the group and check that you agree on the structure of the flow-graph for the program.

3. (10 minutes) For each variable, write down the < D, U > pairs.

4. (10 minutes) Write down tests that satisfy one of the following coverage criteria:

(a) All < D, U > pairs

(b) All < D, U > paths

5. (10 minutes) As a whole class, compare the tests sets devised for the two coverage criteria and discuss which is stronger. Can you think of a test that passes one of the two criteria but fails the other.

That is all