PATH TESTING

Prepared By

Mr. C. R. Belavi
CSE, HSIT, NDS

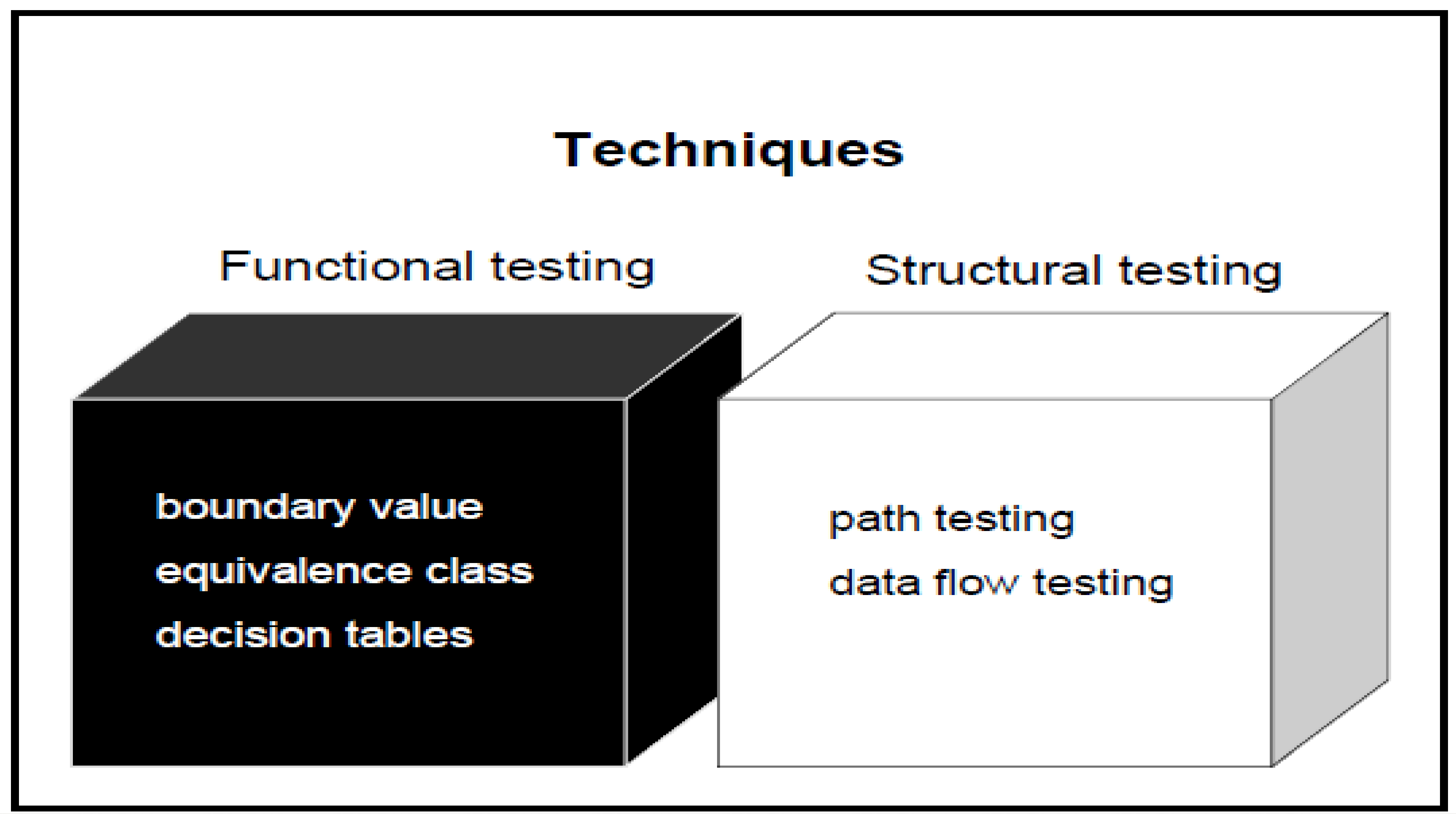
Unit 3

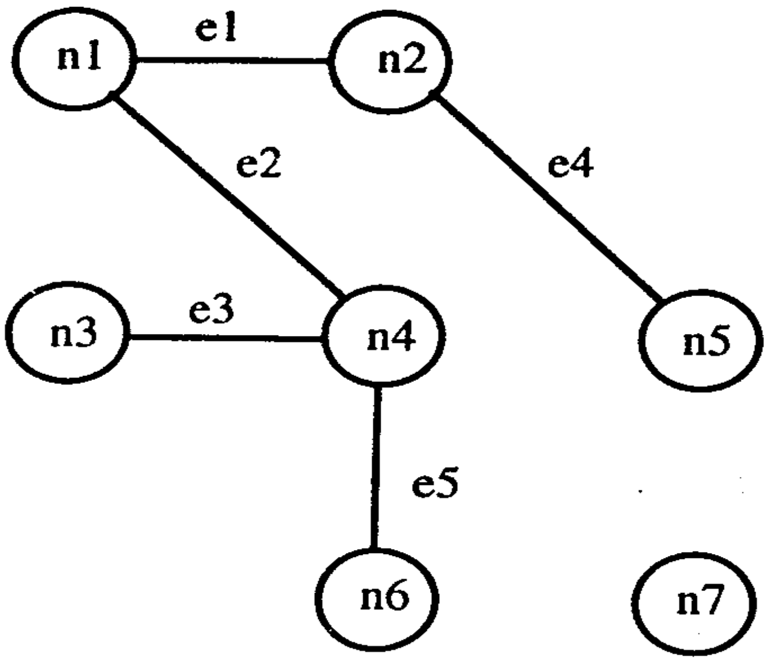
- **Path Testing, Data Flow Testing:** DD paths, Test coverage metrics, Basis path testing, guidelines and observations. Definition-Use testing, Slice-based testing, Guidelines and observations.

content

- DD Paths
- Test Coverage metrics
- Basis Path testing

Software Testing





$\deg(n_1) = 2$

$\deg(n_2) = 2$

$\deg(n_3) = 1$

$\deg(n_4) = 3$

$\deg(n_5) = 1$

$\deg(n_6) = 1$

$\deg(n_7) = 0$

G=(V,E)

$$V = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$$

$$E = \{e_1, e_2, e_3, e_4, e_5\}$$

$$= \{(n_1, n_2), (n_1, n_4), (n_3, n_4), (n_2, n_5), (n_4, n_6)\}$$

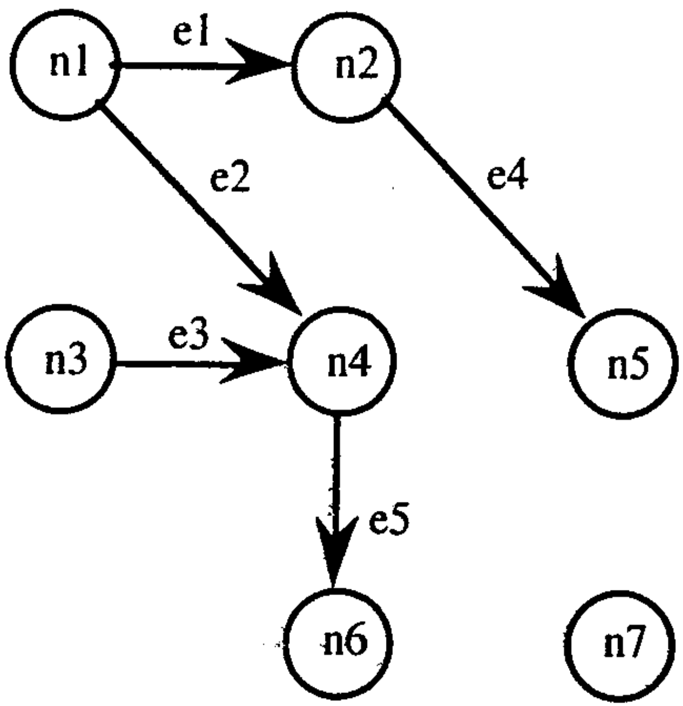
PATH

Definition

A path is a sequence of edges such that, for any adjacent pair of edges e_i, e_j in the sequence, the edges share a common (node) endpoint.

Path	*Node Sequence*	*Edge Sequence*
Between n_1 and n_5	n_1, n_2, n_5	e_1, e_4
Between n_6 and n_5	n_6, n_4, n_1, n_2, n_5	e_5, e_2, e_1, e_4
Between n_3 and n_2	n_3, n_4, n_1, n_2	e_3, e_2, e_1
Between n_1 and n_5	n_1, n_2, n_5	e_1, e_4

Directed Graph



$indeg(n_1) = 0$ $outdeg(n_1) = 2$

$indeg(n_2) = 1$ $outdeg(n_2) = 1$

$indeg(n_3) = 0$ $outdeg(n_3) = 1$

$indeg(n_4) = 2$ $outdeg(n_4) = 1$

$indeg(n_5) = 1$ $outdeg(n_5) = 0$

$indeg(n_6) = 1$ $outdeg(n_6) = 0$

$indeg(n_7) = 0$ $outdeg(n_7) = 0$

Program graph

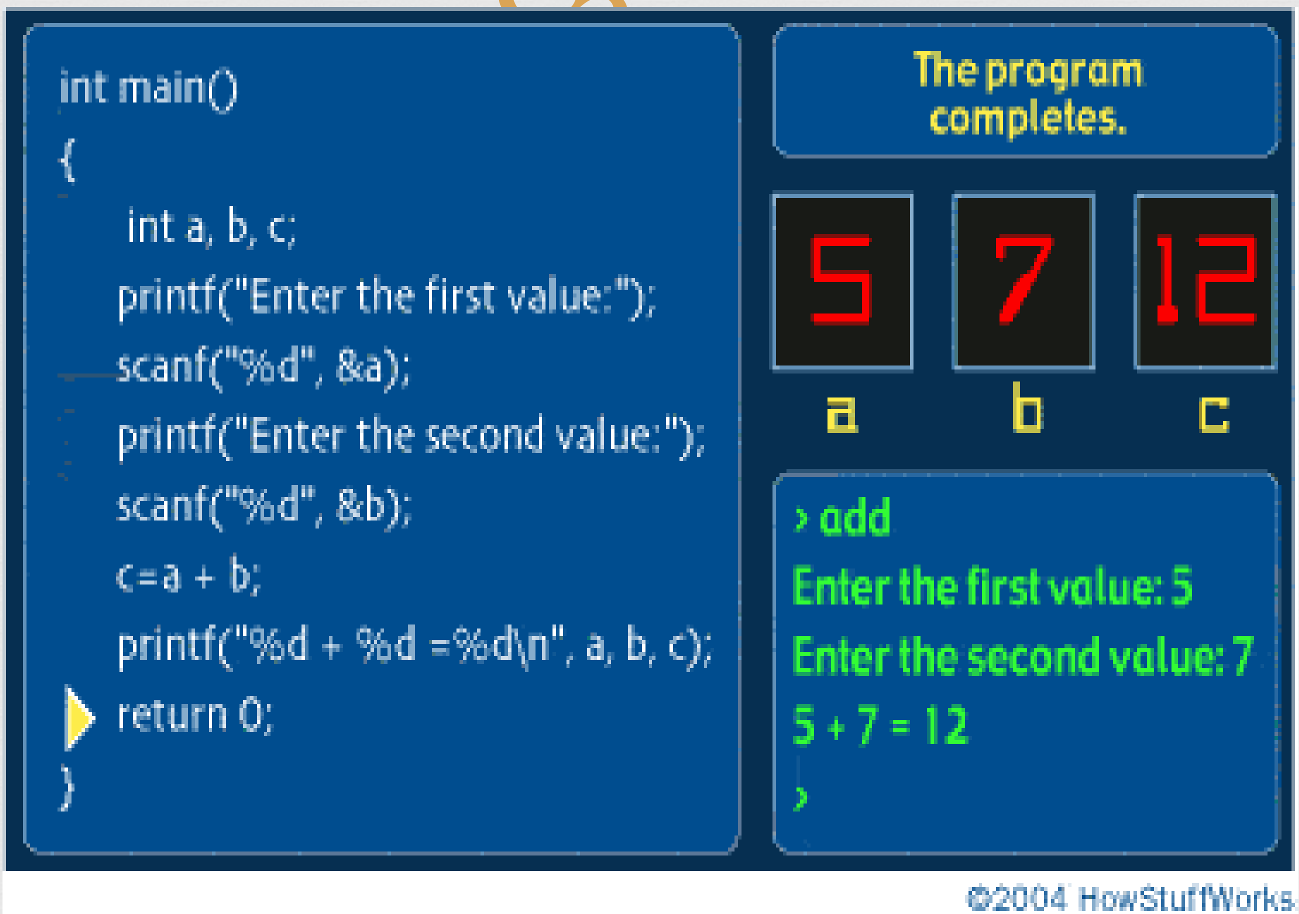
Given a program written in an imperative programming language, its program graph is a directed graph in which:

1. (Traditional Definition)
   Nodes are program statements, and edges represent flow of control (there is an edge from node i to node j iff the statement corresponding to node j can be executed immediately after the statement corresponding to node i).

PATH Testing

Given a program written in an imperative programming language, its program graph is a directed graph in which nodes are statement fragments, and edges represent flow of control. (A complete statement is a "default" statement fragment.)

If i and j are nodes in the program graph, an edge exists from node i to node j iff the statement fragment corresponding to node j can be executed immediately after the statement fragment corresponding to node i.


int main()
{
int a, b, c;
printf("Enter the first value:");
scanf("%d", &a);
printf("Enter the second value:");
scanf("%d", &b);
c=a + b;
printf("%d + %d =%d\n", a, b, c);
return 0;
}
The program completes.
5
7
12
a
b
c
› add
Enter the first value: 5
Enter the second value: 7
5 + 7 = 12
›
©2004 HowStuffWorks

DD Path

Definition

A DD-Path is a chain in a program graph such that:

Case 1: it consists of a single node with indeg = 0

Case 2: it consists of a single node with outdeg = 0

Case 3: it consists of a single node with indeg ≥ 2 or outdeg ≥ 2

Case 4: it consists of a single node with indeg = 1 and outdeg = 1

Case 5: it is a maximal chain of length ≥ 1

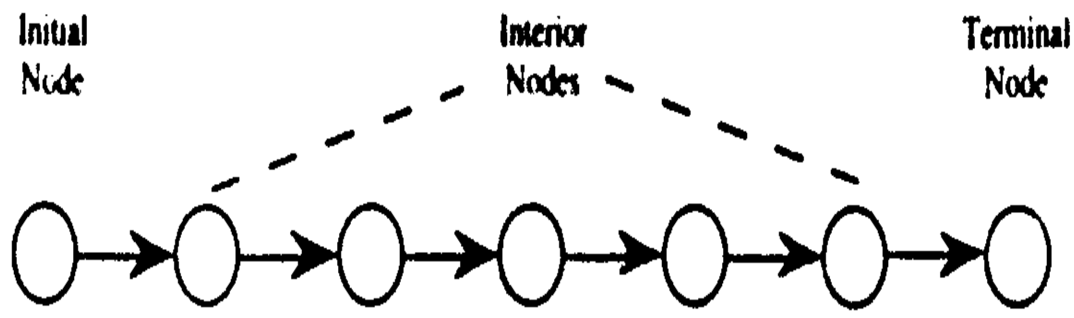
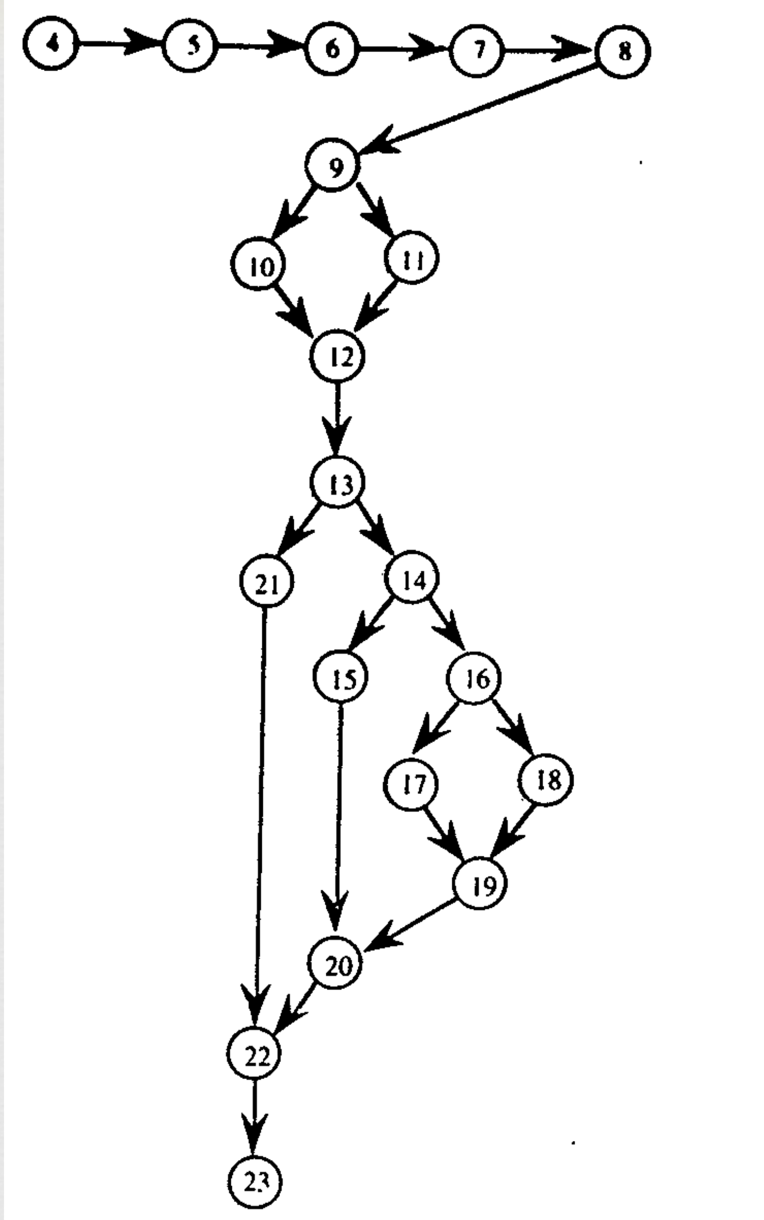


Figure 9.3 A chain of nodes in a directed graph.

```
1. Program triangle2 'Structured programming version of simpler specification
2. Dim a,b,c As Integer
3. Dim IsATriangle As Boolean
      'Step 1: Get Input
4. Output("Enter 3 integers which are sides of a triangle")
5. Input(a,b,c)
6. Output("Side A is ",a)
7. Output("Side B is ",b)
8. Output("Side C is ",c)
      'Step 2: Is A Triangle?
9. If (a < b + c) AND (b < a + c) AND (c < a + b)
10.     Then IsATriangle = True
11.     Else IsATriangle = False
12 EndIf
      'Step 3: Determine Triangle Type
13. If IsATriangle
14.     Then  If (a = b) AND (b = c)
15.              Then Output ("Equilateral")
16.              Else  If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
17.                       Then  Output ("Scalene")
18.                       Else  Output ("Isosceles")
19.                    EndIf
20.           EndIf
21.     Else  Output("Not a Triangle")
22. EndIf
23. End triangle2
```

→



Program graph of the triangle program.

Table 9.1 Types of DD-Paths in Figure 9.1

Program Graph Nodes	*DD-Path Name*	*Case of Definition*
4	first	1
5–8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	last	2

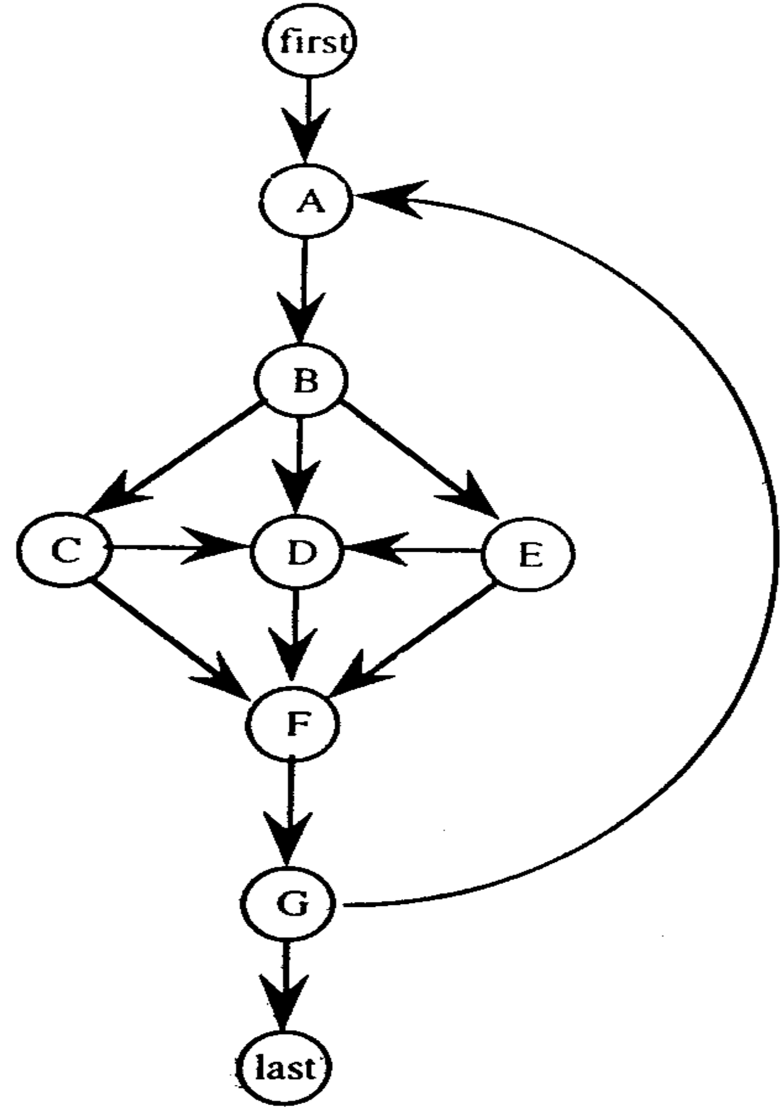
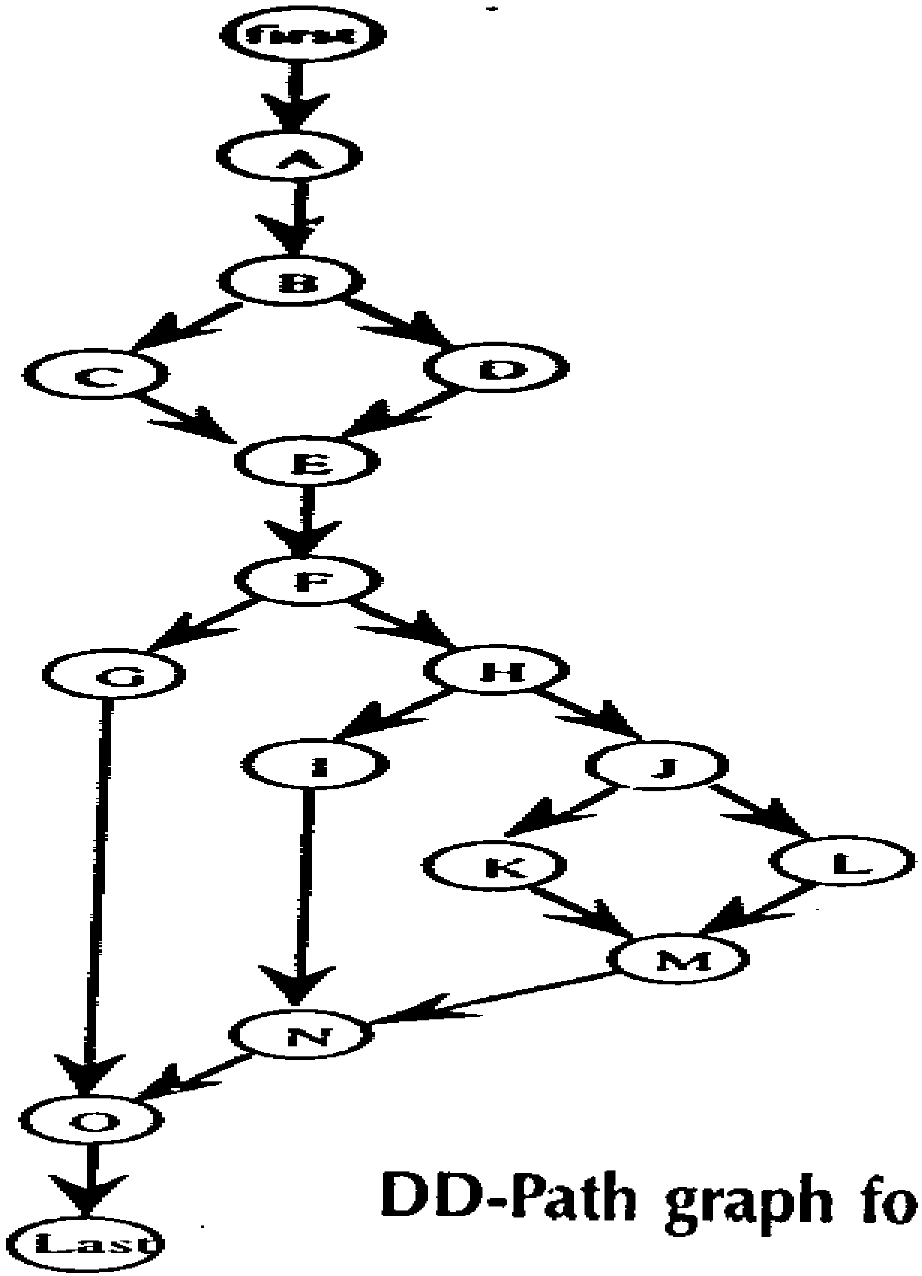
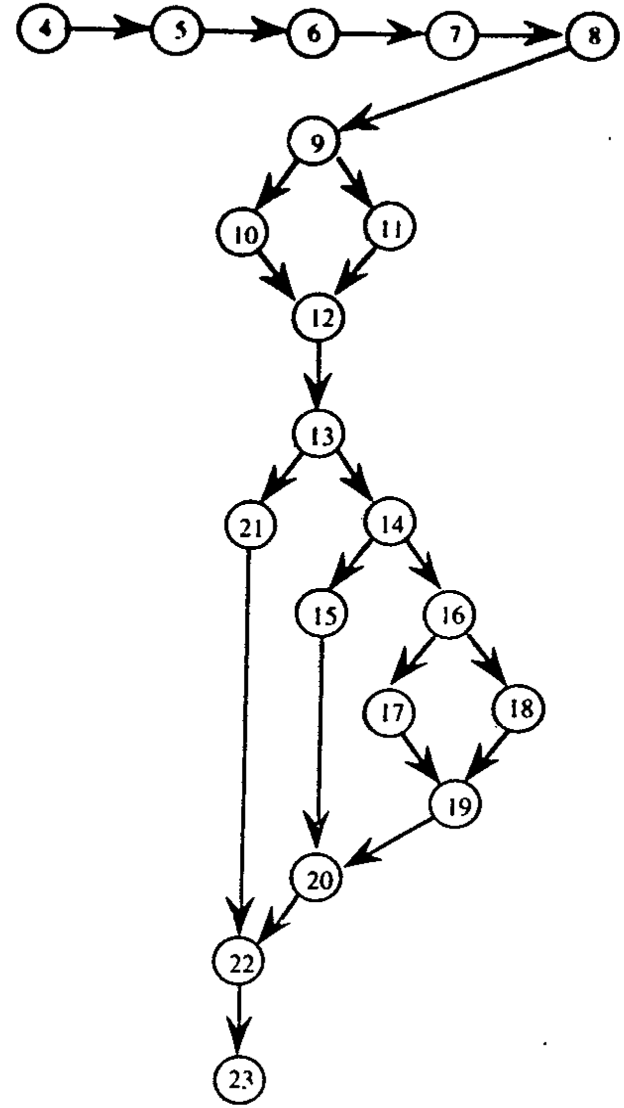


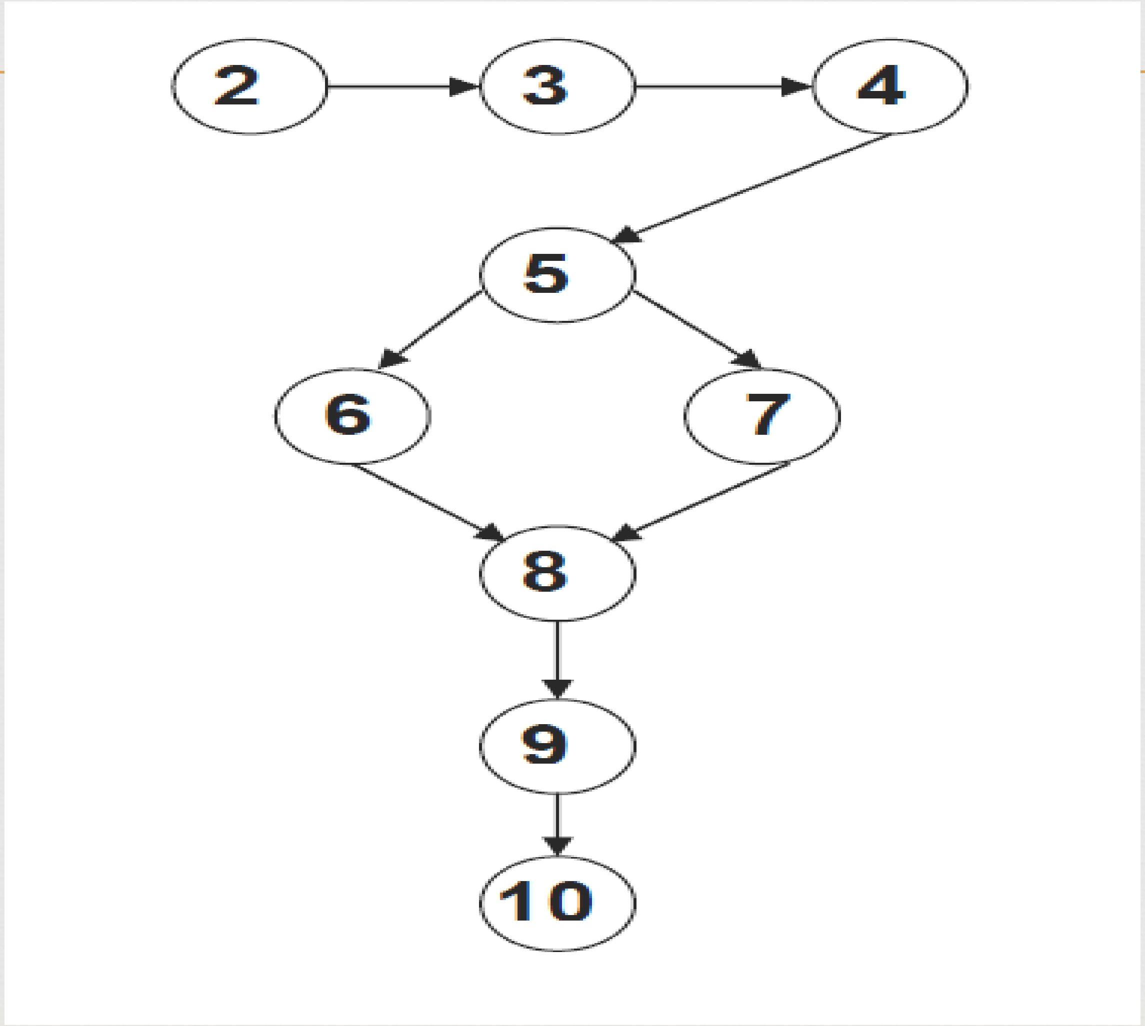
Figure 9.2 Trillions of paths.





DD-Path graph for the triangle program.

1. Program 'Simple Subtraction'
2. Input (x, y)
3. Output (x)
4. Output (y)
5. If $x > y$ then DO
6. $x - y = z$
7. Else $y - x = z$
8. EndIf
9. Output (z)
10. Output "End Program"


2
3
4
5
6
7
8
9
10

Test Coverage Metrics

- Test coverage metrics are a device to measure the extent to which a set of test cases covers a program.

Test Coverage Metrics

Metric	Description of Coverage
C_0	Every Statement
C_1	Every DD-Path
C_1P	Every predicate to each outcome
C_2	C_1 Coverage + loop coverage
C_d	C_1 Coverage + every dependent pair of DD-Paths
C_{MCC}	Multiple condition coverage
C_ik	Every program path that contains up to k repetitions of a loop (usually k=2)
C_{stat}	"Statistically significant" fraction of paths
C_∞	All possible execution paths

- Statement Testing: Every statement is executed by the test set and Predicate Testing: Every logical predicate is executed by the test set
- DD path testing: check for all possible paths
- Dependent pair of DD-paths: reference/Dependent
- Multiple condition coverage: use truth table instead of predicate.
- Loop coverage: concatenated, nested, horrible

Selection Statements

- Using `if` and `if...else`
- Nested `if` Statements
- Using `switch` Statements

Repetition Statements

- Looping: `while`, `do`, and `for`
- Nested loops
- Using `break` and `continue`

Boiler shutdown conditions

1. The water level in the boiler is below X lbs. (a)
2. The water level in the boiler is above Y lbs. (b)
3. A water pump has failed. (c)
4. A pump monitor has failed. (d)
5. Steam meter has failed. (e)

Boiler in degraded mode when either is true.

The boiler is to be shut down when a or b is true or the boiler is in degraded mode and the steam meter fails. We combine these five conditions to form a compound condition (predicate) for boiler shutdown.

Another example

A condition is represented formally as a predicate, also known as a Boolean expression. For example, consider the requirement

``*if the printer is ON and has paper then send document to printer*."

This statement consists of a condition part and an action part. The following predicate represents the condition part of the statement.

p_r: (printerstatus=ON) $\wedge$ (printertray!= empty)

Predicates

Relational operators (relop): $\{<, \leq, >, \geq, =, \neq.\}$
= and == are equivalent.

Boolean operators (bop): $\{!, \wedge, \vee$, xor$\}$ also known as {not, AND, OR, XOR}.

Relational expression: e1 relop e2. (e.g. a+b<c)
e1 and e2 are expressions whose values can be compared using relop.

Simple predicate: A Boolean variable or a relational expression. (x<0)

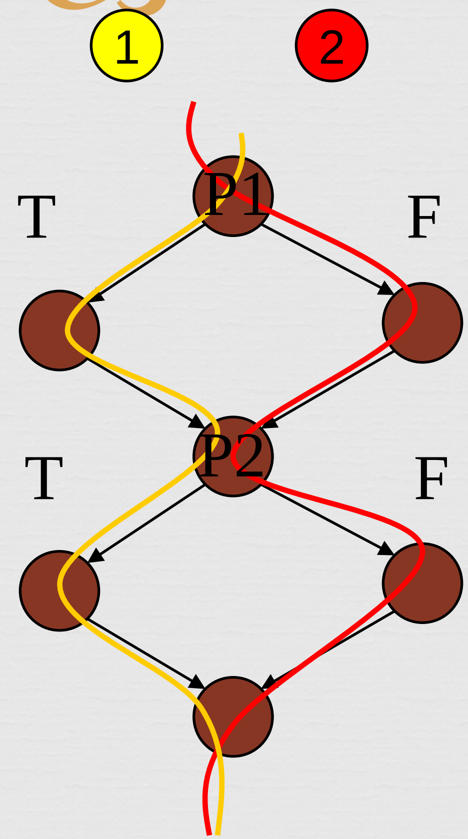
Compound predicate: Join one or more simple predicates using bop. (gender==“female”$\wedge$age>65)

Statement and Predicate Coverage Testing

- Statement coverage based testing aims to devise test cases that collectively exercise all statements in a program.
- Predicate coverage (or branch coverage, or decision coverage) based testing aims to devise test cases that evaluate each simple predicate of the program to True and False.
- For example in predicate coverage for the condition
  if(A or B) then C we could consider the test cases A=True, B= False (true case), and A=False, B=False (false case). Note if the program was encoded as *if(A) then* C we would not detect any problem.

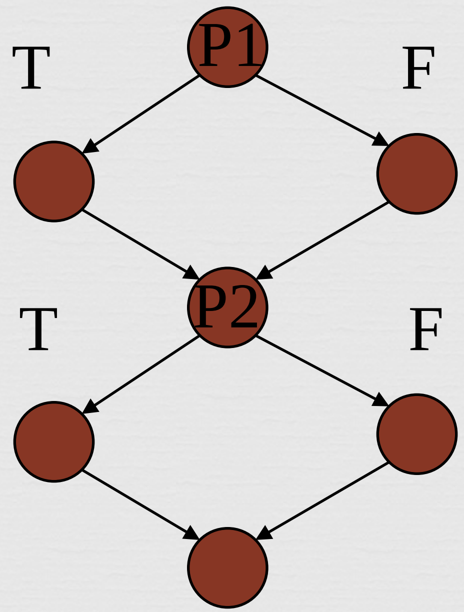
DD-Path Graph Edge Coverage C1

Here a T,T and F,F combination will suffice to have DD-Path Graph edge coverage or Predicate coverage C1



DD-Path Coverage Testing C_1P

- This is the same as the C_1 but now we must consider test cases that exercise all possible outcomes of the choices T,T, T,F, F,T, F,F for the predicates P1, and P2 respectively, in the DD-Path graph.
- If else, case statements are checked.



Multiple Condition Coverage Testing

- Now if we consider that the predicate P1 is a compound predicate (i.e. (A or B)) then Multiple Condition Coverage Testing requires that each possible combination of inputs be tested for each decision.

- Example: "if (A or B)" requires 4 test cases:

  A = True, B = True

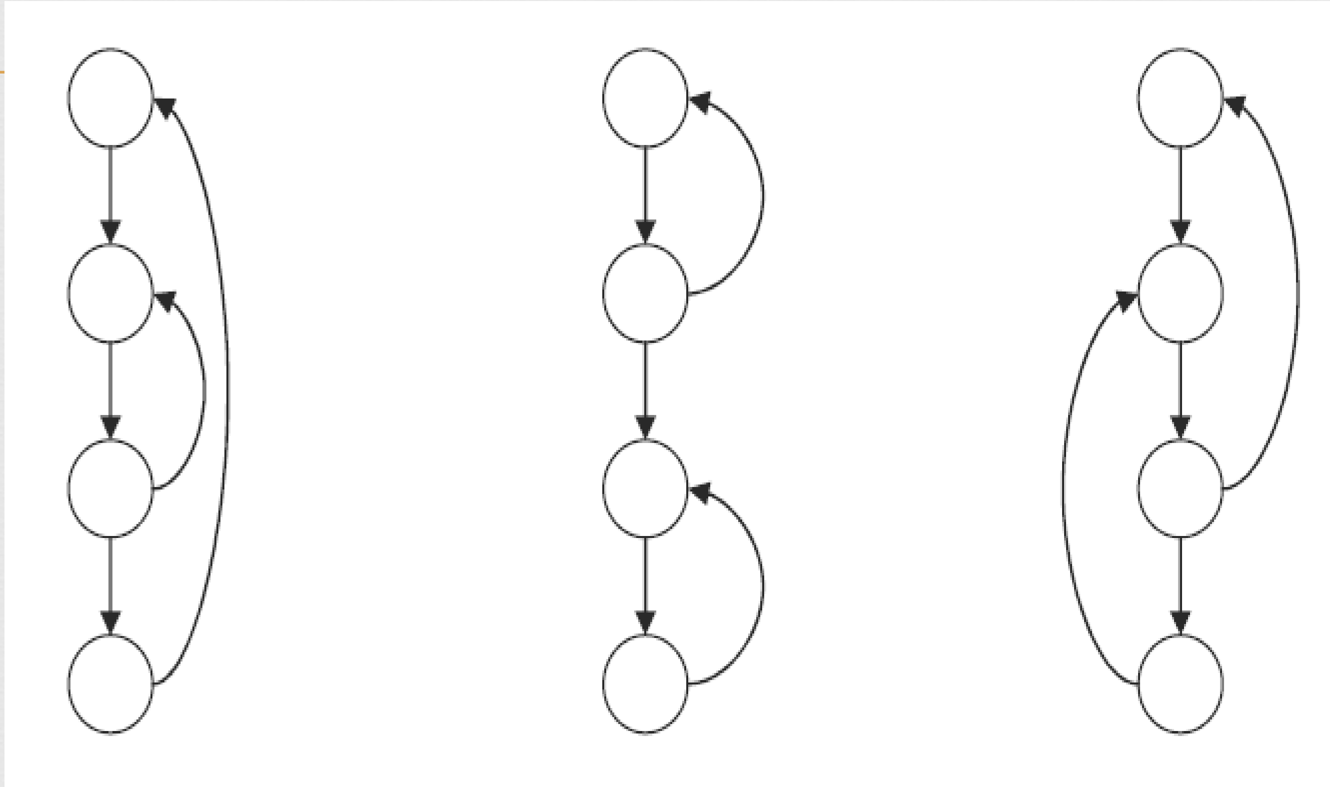
  A = True, B = False

  A = False, B = True

  A = False, B = False
- The problem: For n conditions, 2^n test cases are needed, and this grows exponentially with n.

Loop Coverage

- The simple view of loop testing coverage is that we must devise test cases that exercise the two possible outcomes of the decision of a loop condition that is one to traverse the loop and the other to exit (or not enter) the loop.
- An extension would be to consider a modified boundary value analysis approach where the loop index is given a minimum, minimum +, a nominal, a maximum -, and a maximum value or even robustness testing.
- Concatenated: sequence of disjoint loops
- Nested: one is contained inside another.
- Horrible:



Concatenated loop nested horrible

Basis Path Testing

- Mathematicians define a basis in terms of a structure called a vector space, which is a set of elements(vectors) as well as operations that correspond to multiplication & addition defined for the vectors.

McCabe's basis path method

- McCabe based his view of testing on a major result from graph theory.
- Which states that the cyclomatic no. of a strongly connected graph is the number of linearly independent circuits in the graph.
- We can create a strongly connected graph by adding an edge from the(every) sink node to the (every) source node.

Cont.,

V(G)=e-n+2p; arbitrary directed graph

V(G)=e-n+p; strong directed graph

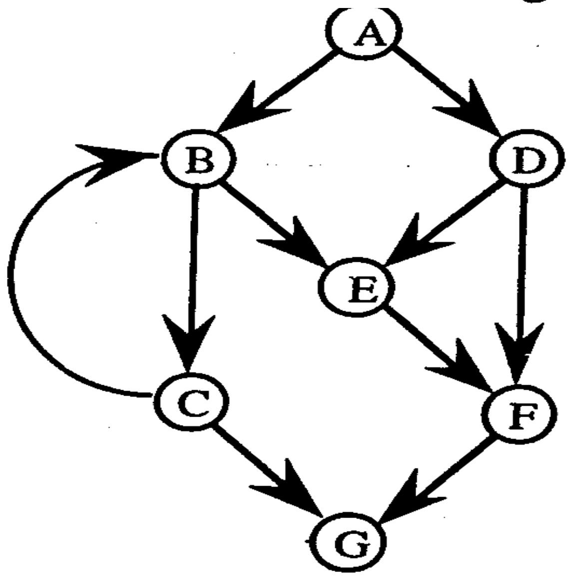
e– no of edges, n – no of nodes, p – no of connected regions.

- Two important points should be made here.

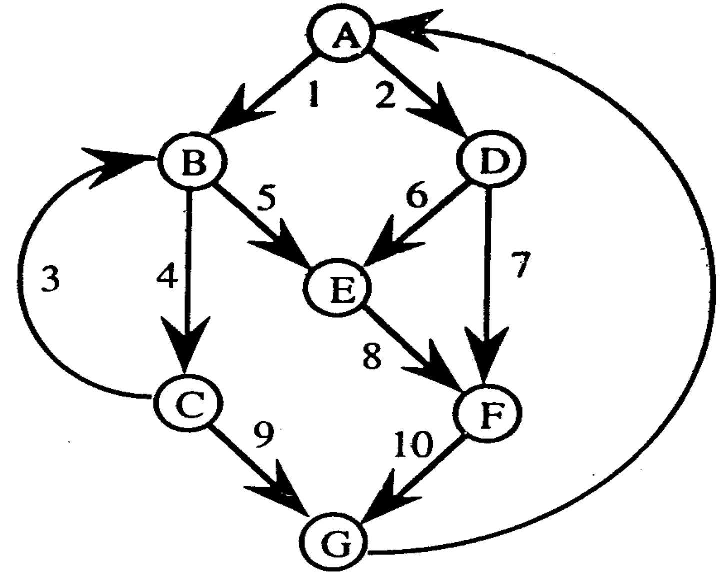
1)if there is a loop, it only has to be traversed once, or else the basis will contain redundant

2)it is possible for there to be more than one basis.

McCabe's control graph.



McCabe's derived strongly connected graph.



$$V(G) = e - n + 2p$$
$$= 10 - 7 + 2(1) = 5,$$

and the number of linearly independent circuits in the graph in Figure 9.7 is

$$V(G) = e - n + p$$
$$= 11 - 7 + 1 = 5,$$

Cont.,

- The cyclomatic complexity of the strong connected graph is 5; thus there are five linearly independent circuits.
- If we now delete the added edge from node G to node A. these 5 circuits become five linearly independent paths from node A to node G.
- In a small graphs, we can identify independent paths

P1:A,B,C,G

P2:A,B,C,B,C,G

P3:A,B,E,F,G

P4:A,D,E,F,G

P5:A,D,F,G

Cont.,

- Path addition is simply 1 path followed by another path, & multiplication corresponds to repetitions of a path.
- McCabe arrives at a vector space of program paths.
- Path A,B,C,B,E,F,G is the basis sum p2+p3-p1 & the path A,B,C,B,C,B,C,G is the linear combination 2p2-p1

Table 9.3 Path/Edge Traversal

Path/Edges Traversed	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
p1: A, B, C, G	1	0	0	1	0	0	0	0	1	0
p2: A, B, C, B, C, G	1	0	1	2	0	0	0	0	1	0
p3: A, B, E, F, G	1	0	0	0	1	0	0	1	0	1
p4: A, D, E, F, G	0	1	0	0	0	1	0	1	0	1
p5: A, D, F, G	0	1	0	0	0	0	1	0	0	1
ex1: A, B, C, B, E, F, G	1	0	1	1	1	0	0	1	0	1
ex2: A, B, C, B, C, B, C, G	1	0	2	3	0	0	0	0	1	0

Cont.,

- Each decision is “flipped” that is when a node of out degree>=2 is reached, a different edge must be taken.

Cont.,

the path through nodes A, B, C, B, E, F, G as the baseline. (This was expressed in terms of paths p1 – p5 earlier.) The first decision node (outdegree ≥2) in this path is node A; so for the next basis path, we traverse edge 2 instead of edge 1. We get the path A, D, E, F, G, where we retrace nodes E, F, G in path 1 to be as minimally different as possible. For the next path, we can follow the second path, and take the other decision outcome of node D, which gives us the path A, D, F, G. Now, only decision nodes B and C have not been flipped; doing so yields the last two basis paths, A, B, E, F, G and A, B, C, G.

Essential Complexity

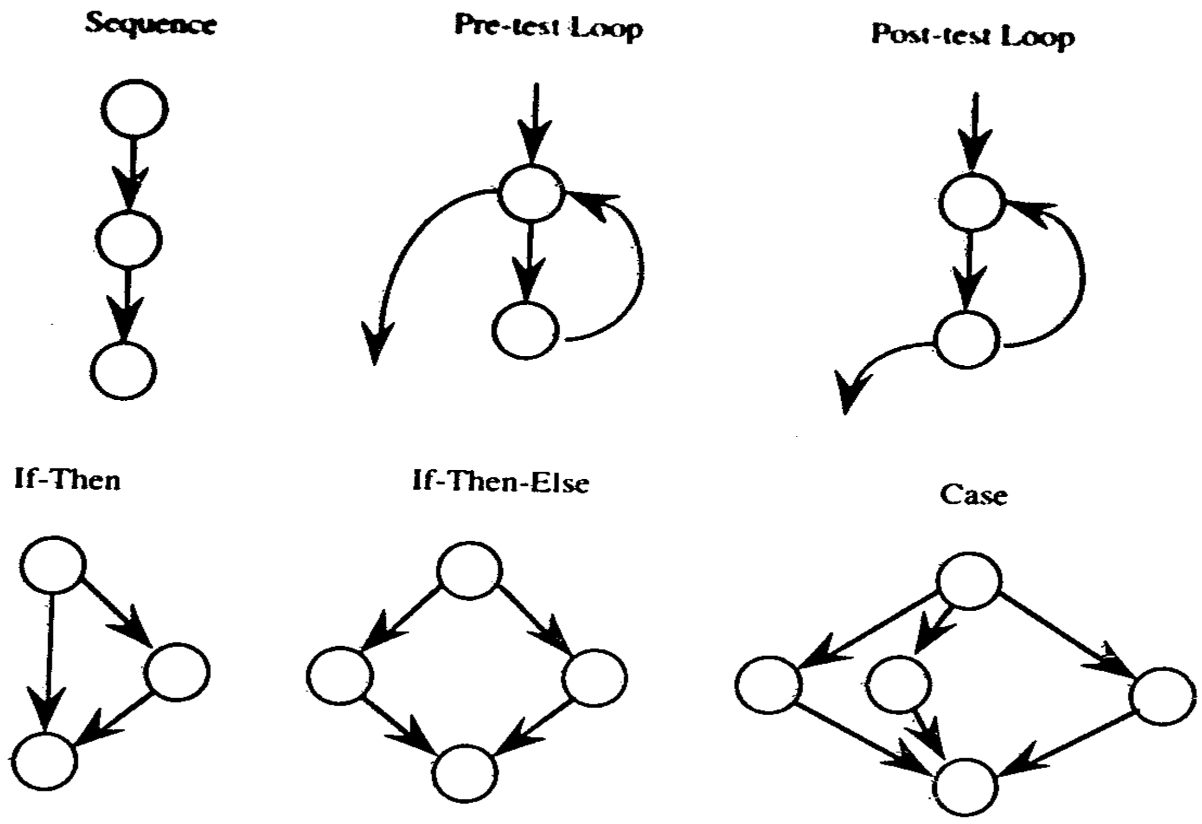


Figure 9.8 Structured programming constructs.

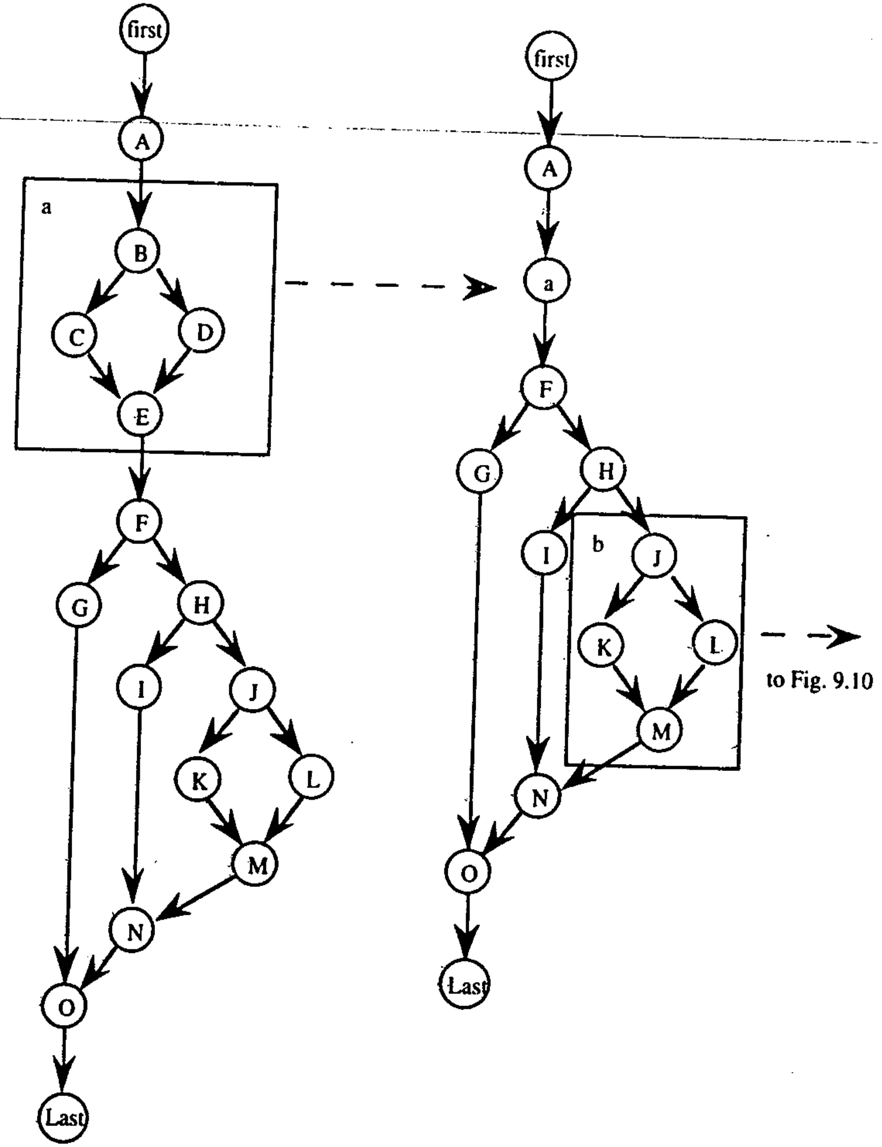
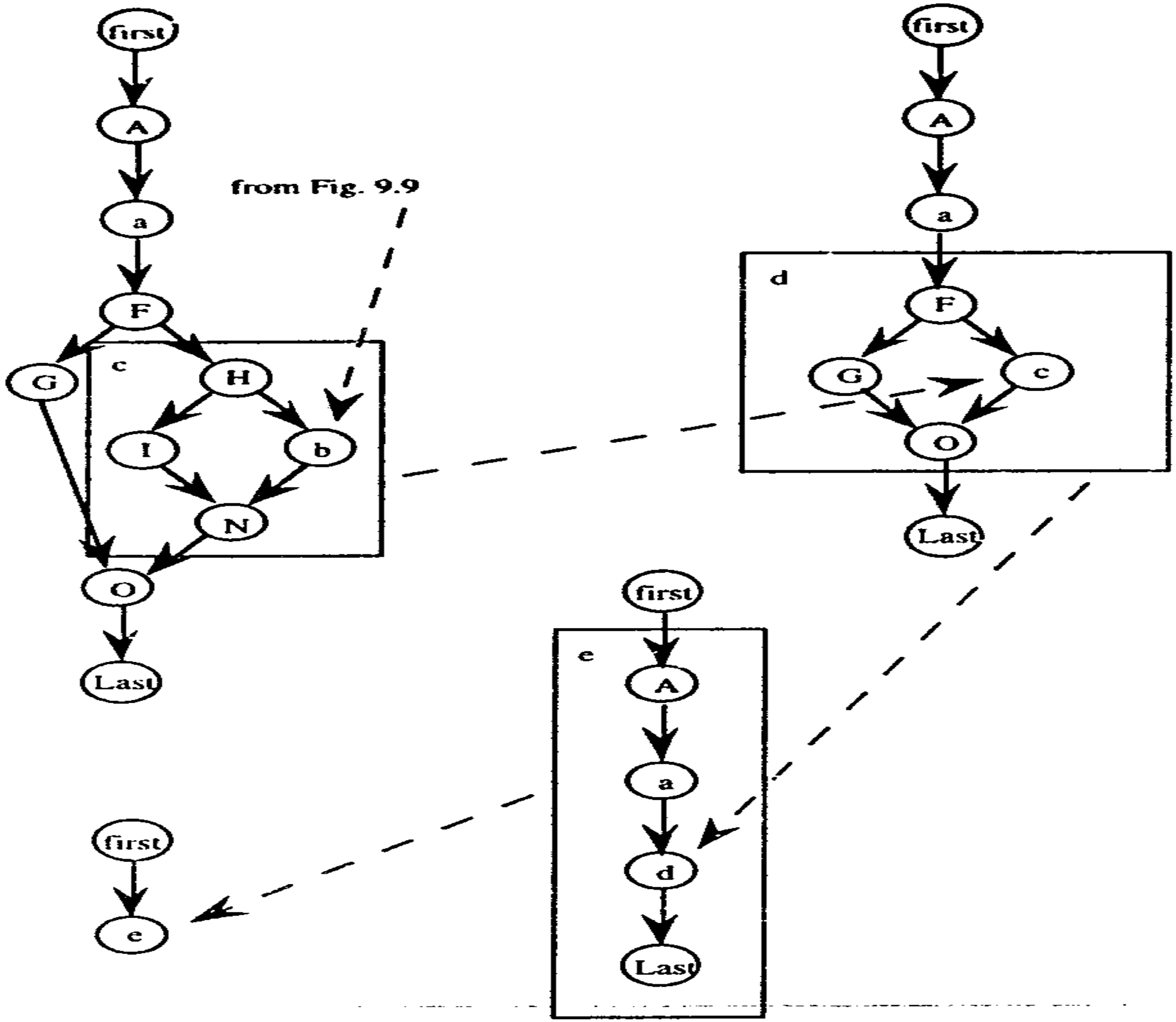
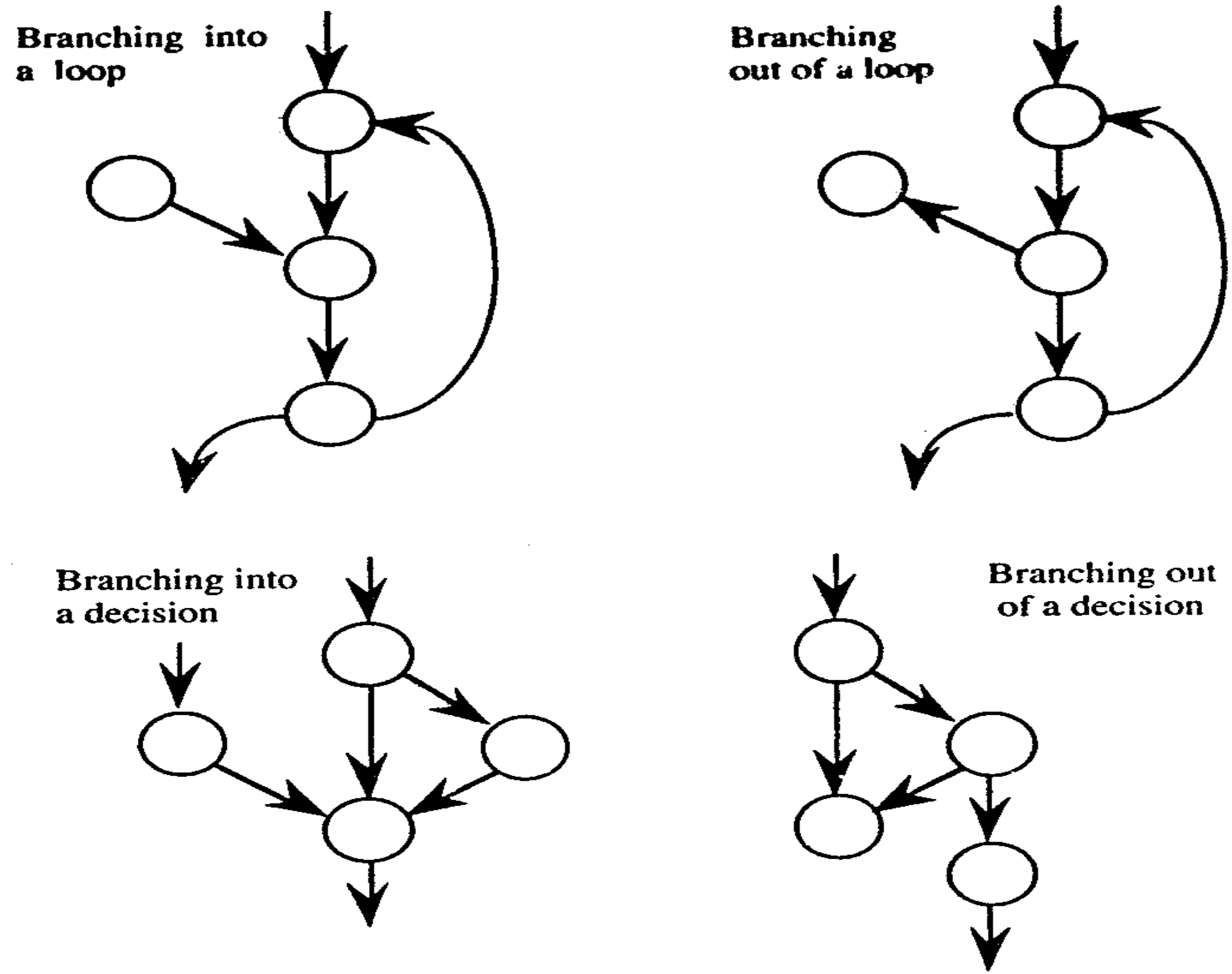


Figure 9.9 Condensing with respect to the structured programming constructs.


first
A
a
F
G
c
H
I
b
N
O
Last
from Fig. 9.9
first
A
a
d
F
G
c
O
Last
first
e
A
a
d
Last
first
e



1 Violations of structured programming.

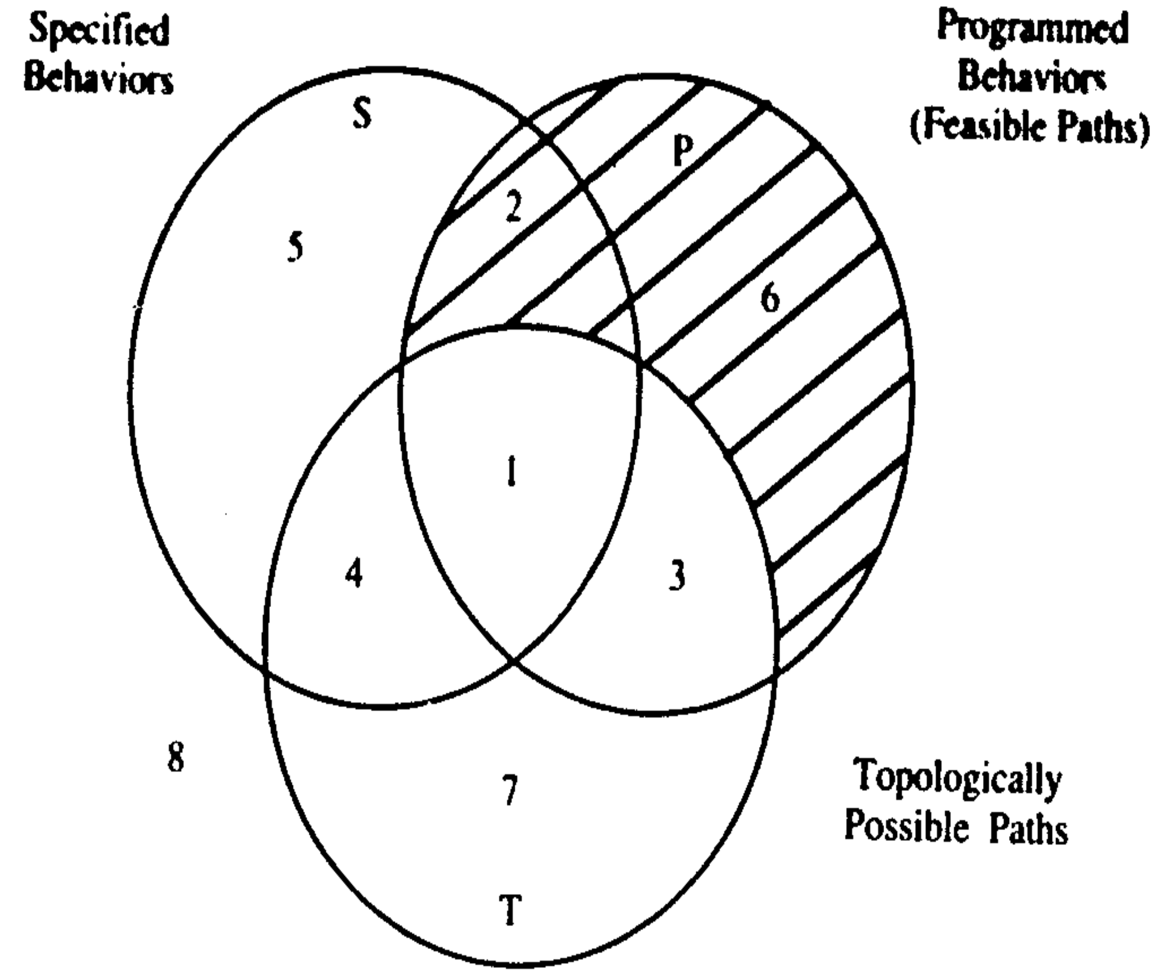


Figure 9.12 Feasible and topologically possible paths.

Slice Based Testing

Prepared By
Mr. C. R. Belavi
Asst. Professor
Dept. of CSE, HIT, NDS

Data Flow Testing

- Data flow testing(DFT) is NOT directly related to the design diagrams of data-flow-diagrams(DFD).
- It is a form of *structural testing* and a *White Box* testing technique that focuses on program variables and the paths:
  - From the point where a variable, v, is defined or assigned a value
  - To the point where that variable, v, is used

Static Analysis of Data

- *Static analysis* allows us to check (test or find faults) *without running the actual code*, and we can apply it to analyzing variables as follows:

  1. A variable that is **defined but never used**
  2. A variable that is **used but never defined**
  3. A variable that is **defined a multiple times prior to usage**.

- While these are dangerous signs, they may or may not lead to defects.
  1. A defined, but never used variable may just be extra stuff
  2. Some compilers will assign an initial value of zero or blank to all undefined variable based on the data type.
  3. Multiple definitions prior to usage may just be bad and wasteful logic

- *We are more interested in "executing" the code than just static analysis, though*.

Variable Define-Use Testing

- **In define-use testing, we are interested in testing (executing) certain paths that a variable is *defined* – to - its *usage*.**

- **These paths will provide further information that will allow us to decide on choice of test cases beyond just the earlier discussed paths analysis (all statements testing or dd-testing (branch) or linearly independent paths).**

Data Dependencies and Data Flow Testing(DFT)

- **In Data Flow Testing (DFT) we are interested in the "*dependencies" among data or "relationships" among data* ----- Consider a data item, X:**
  - **Data Definitions (value assignment) of X: via 1) initialization, 2) input, or 3) some assignment.**
    - **Integer X; (compiler initializes X to 0 or it will be "trash")**
    - **X = 3;**
    - **Input X;**
  - **Data Usage (accessing the value) of X: for 1) computation and assignment (C-Use) or 2) for decision making in a predicate (P-Use)**
    - **Z = X + 25; (C-Use)**
    - **If (X > 0) then ----- (P-Use)**

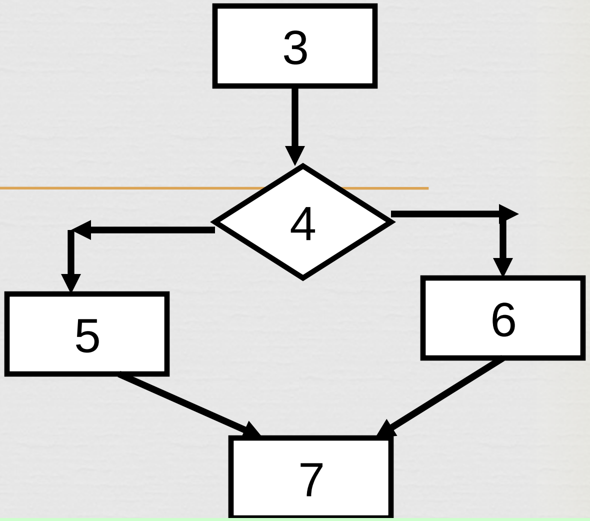
Some Definitions

- **Defining node, DEF(v,n), is a node, n, in the program graph where the specific variable, v, is *defined or given its value (value assignment)*.**
- **Usage node, USE(v,n), is a node, n, in the program graph where the specific variable, v, *is used*.**
- **A P-use node is a usage node where the variable, v, is used as a predicate (or for a branch-decision-making).**
- **A C-use node is any usage node that is not P-used.**
- **A Definition-Use path, *du-path*, for a specific variable, v, is a path where DEF(v,x) and USE(v,y) are the initial and the end nodes of that path.**
- **A Definition-Clear path for a specific variable, v, is a Definition-Use path with DEF(v,x) and USE(v,y) such that there is no other node in the path that is a defining node of v. (*e.g. v does not get reassigned in the path.*)**

Simple Example

```
1. Pseudo-code Sample
2.   int a, b
3.   input (a, b)
4.   if (a > b)
5.      then Output (a, " a bigger than b")
6.      else Output (b, " b is equal or greater than a")
7. end
```

← - - - - - - - *This is type defining not value* (points to line 2)



The following are examples of the definitions:

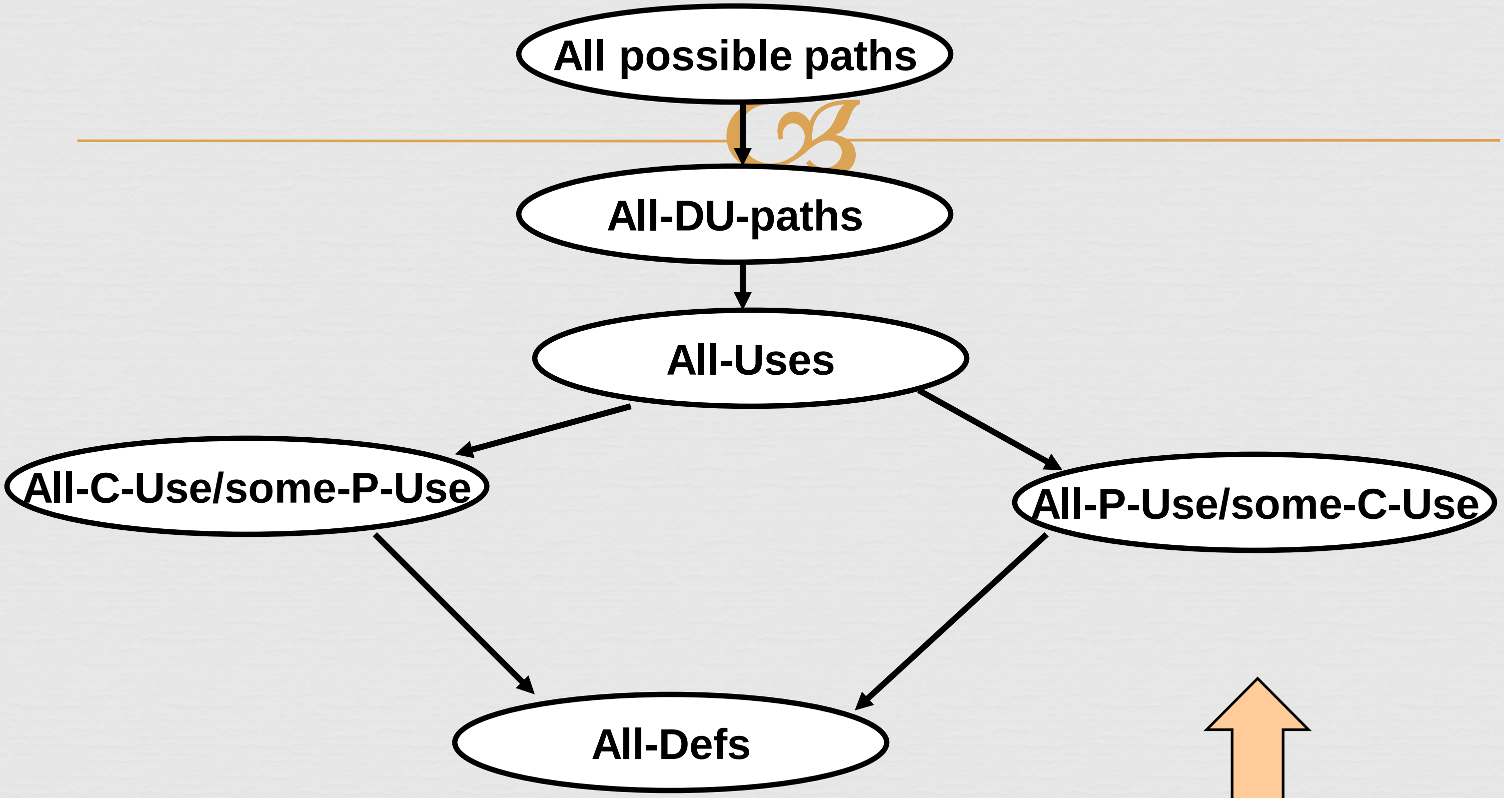
- **DEF(a, 3) – node 3 is a *<u>defining node</u>* of variable "a" --- a value is assigned to "a"**
- **USE(a, 4) – node 4 is a *<u>usage node</u>* of variable "a"**
- **USE(a, 5) – node 5 is a *<u>usage node</u>* of variable "a"**
- **USE (a,4) is a *<u>P-use node</u>* while**
- **USE(a,5) is *<u>C-use node</u>***
- **Path that begins with DEF(a,3) and ends with USE(a,4) is a *<u>definition-use path of a</u>***
- **Path that begins with DEF(a,3) and ends with USE(a,5) is a *<u>definition-use path of a</u>***
- **Path that begins with DEF(a,3) and ends with USE(a,5) is a *<u>definition-clear path of a</u>***
- **Path that begins with DEF(b,3) and ends with USE(b.6) is a *<u>definition-use path of b</u>***

Note that: if we choose the definition-use paths [last two examples above] of both variables a and b, then it is the *same as executing the decision-decision (dd) path or branch testing.*

Definitions of Definition-Use (DU) testing

- **All-Defs** : contains set of test paths, P, where for every variable v in the program, P includes definition-clear paths from *every DEF(v,n) to only one of its use node.*
- **All-Uses**: contains set of test paths, P, where for every variable v in the program, P includes definition-clear paths from every DEF(v,n) to *every* use of v and to the successor node of that use node.
- **All-P-Use/Some C-Use**: contains set of test paths, P, where for every variable v in the program, P contains definition-clear paths from DEF(v,n) to every predicate –use node of v; and if there is no predicate-use, then the definition-clear path leads to at least one C-use node of v.
- **All-C-Use/Some P-Use**: contains set of test paths, P, where for every variable v in the program, P contains definition-clear paths from DEF(v,n) to every computation-use node of v; and if there is no computation-use, then the definition-clear path leads to at least one predicate-use node of v.
- **All-DU-paths:** contains the set of paths, P, where for every variable v in the program, P includes definition-clear paths from every DEF(v,n) to every USE(v,n) and to the successor node of each of the USE(v,n), and that these paths are either *single loop traversals* or they are cycle free.

Summarizing hierarchy



Text page 160 has another chain
Under All-P-Use/some-C-Use;
take a look at that page.

Mark D. Weiser
(July 23, 1952 – April 27, 1999)
Slice Based Testing

- He was a chief scientist at Xerox PARC. Weiser is widely considered to be the father of ubiquitous computing, a term he coined in 1988.

What is a Program Slice?

- A program slice is a subset of a program.
- Program slicing enables programmers to view subsets of a program by filtering out code that is not relevant to the computation of interest.
- *E.g.*, if a program computes many things, including the average of a set of numbers, slicing can be used to isolate the code that computes the average.

Why is Program Slicing Useful?

- Program slices are more manageable for testing and debugging.
- When testing, debugging, or understanding a program, most of the code in the program is irrelevant to what you are interested in.
- Program slicing provides a convenient way of filtering out *"irrelevant"* code.
- Program slices can be computed automatically by statically analyzing the data and control flow of the program.

Definition of Program Slice

- Assume that:
  - *P* is a program.
  - *V* is the set of variables at a program location (line number) *n*.
- A slice *S(V,n)* produces the portions of the program that contribute to the value of *V* just before the statement at location *n* is executed.
- *S(V,n)* is called the ***slicing criteria***.

A Program Slice Must Satisfy the Following Conditions:

- Slice *S(V,n)* must be derived from *P* by deleting statements from *P*.
- Slice *S(V,n)* must be syntactically correct.
- For all executions of *P*, the value of *V* in the execution of *S(V,n)* just before the location *n* must be the same value of *V* in the execution of the program *P* just before location *n*.

Example

1.a=3;

2.b=6;

3.c=b^2;

4.d=a^2+b^2;

5.c=a+b;

S(c,5)

1.a=3;

2.b=6;

5.c=a+b;

S(c,3)

2.b=6;

3. c=b^2;

Example: Assume the Following Program …

```
main() {
int mx, mn, av;
int tmp, sum, num;

tmp = readInt():
mx = tmp;
mn = tmp;
sum = tmp;
num = 1;

while(tmp >= 0)
  {
  if (mx < tmp)
    mx = tmp;
  if (mn > tmp)
    mn = tmp;
  sum += tmp;
  ++num;
  tmp = readInt();
  }

av = sum / num;
printf(“\nMax=%d”, mx);
printf(“\nMin=%d”, mn);
printf(“\nAvg=%d”, av);
printf(“\nSum=%d”, sum);
printf(“\nNum=%d”, num);
}
```

Slice S(num,26)

```
    main() {
2.   int tmp, num;
4.   tmp = readInt():
8.   num = 1;
10. while(tmp >= 0)
11.    {
17.     ++num;
18.     tmp = readInt();
19.    }
26. printf("\nNum=%d", num);
     }
```

Slice S(sum, 25)

```
    main() {
2.   int tmp, sum;
4.   tmp = readInt():
7.   sum = tmp;
10. while(tmp >= 0)
11.   {
16.    sum += tmp;
18.    tmp = readInt();
19.   }
25. printf("\nSum=%d", sum);
}
```

Slice S(av, 24)

```
    main() {
1.  int av;
2.  int tmp, sum, num;
4.  tmp = readInt():
7.  sum = tmp;
8.  num = 1;
10. while(tmp >= 0)
11.   {
16.   sum += tmp;
17.   ++num;
18.   tmp = readInt();
19.   }
21. av = sum / num;
24. printf("\nAvg=%d", av);
}
```

Slice S(mn, 23)

```
   main() {
1.  int mn;
2.  int tmp;
4.  tmp = readInt():
6.  mn = tmp;
10. while(tmp >= 0)
11.   {
14.   if (mn > tmp)
15.      mn = tmp;
18.   tmp = readInt();
19.   }
23. printf("\nMin=%d", mn);
}
```

Slice S(mx, 22)

```
    main() {
1.  int mx;
2.  int tmp;
4.  tmp = readInt():
5.  mx = tmp;
10. while(tmp >= 0)
11.   {
12.   if (mx < tmp)
13.      mx = tmp;
18.   tmp = readInt();
19.   }
22. printf("\nMax=%d", mx);
}
```

Observations about Program Slicing

- Given a slice *S(X,n)* where variable *X* depends on variable *Y* with respect to location *n*:
  - All **d-uses** and **p-uses** of *Y* before *n* are included in *S(X,n)*.
  - The **c-uses** of *Y* will have no effect on *X* unless *X* is a **d-use** in that statement.
- Slices can be made on a variable at any location.

Program Slicing Process

- Select the slicing criteria (*i.e.*, a variable or a set of variables and a program location).
- Generate the program slice(s).
- Perform testing and debugging on the slice(s). During this step a sliced program may be modified.
- Merge the modified slice with the rest of the modified slices back into the original program.

Tools for Program Slicing

- **Spyder**
  - A debugging tool based on program slicing.
- **Unravel**
  - A program slicer for ANSI C.

References

- [Weiser84] Weiser, M., *Program Slicing*, IEEE Transactions on Software Engineering, Vol. SE-10, No. 4, July, 1984.
- [Gallagher91] Gallagher, K. B., Lyle, R. L., *Using Program Slicing in Software Maintenance*, IEEE Transactions on Software Engineering, Vol. SE-17, No. 8, August, 1991.
- [DeMillo96] DeMillo, R. A., Pan, H., Spafford, E. H, *Critical Slicing for Software Fault Localization*, Proc. 1996 International Symposium on Software Testing and Analysis (ISSTA), San Diego, CA, January, 1996.

THANK YOU☺