Module 3

Structural Testing - Path Testing

Structural methods are all based on the **source code** of the program to be tested, and not on the specification. Because of this **absolute basis**, structural testing methods are very **agreeable** to **rigorous definitions, mathematical analysis**, and **precise measurement.**

➢ Definition

- Given a **program** written in an **imperative programming language**, the **program graph** is a **directed graph** in which **nodes** are **statement fragments** and **edges** represent **flow of control.**
- 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.**
  - The **groups of statements** that make up a **node** in the Program Graph are called a **basic block**.
  - There is a **straightforward algorithm** to segment a **code fragment** into **basic blocks** and create the corresponding **Program Graph.**
- Construction of a **program graph** from **a given program** is illustrated here with the **pseudo code** implementation of the **triangle program** and **maximum of 3 numbers.**
- **Line numbers** refer to **statements** and **statement fragments.** The importance of the program graph is that program **executions** correspond to **paths** from the **source to the sink** nodes.
- We also need to decide whether to **associate nodes** with **non executable** statements such as **variable** and **type declarations.** Here we **do not.**

Example 1

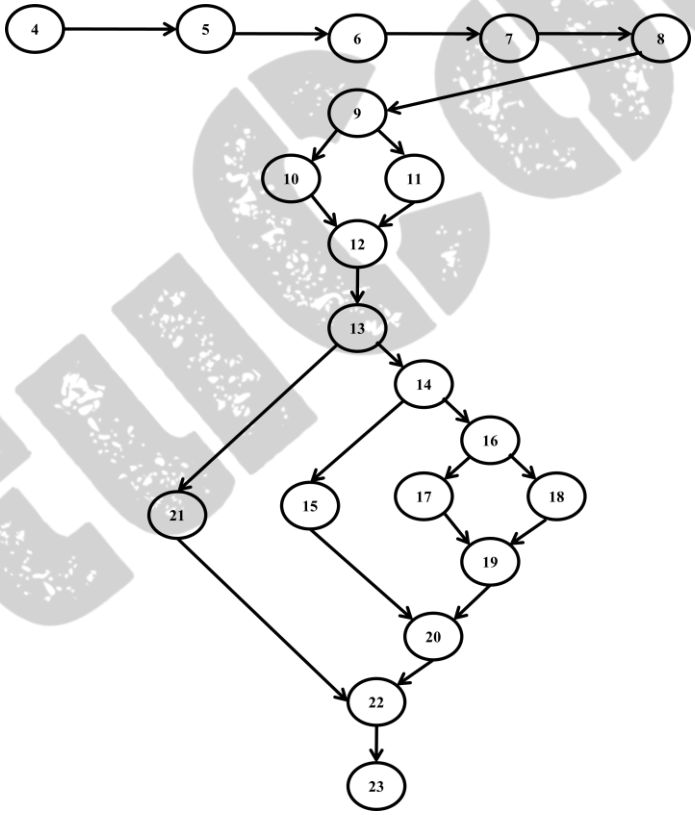
```
1.   Program triangle 2      'Structured programming version of simpler specification
2.   Dim a ,b, c As Integer
3    Dim IsATriangle As Boolean

' Step1 : 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)

' Step2 : 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
‘ Step3 : 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
```

Figure: Program graph of the triangle program.

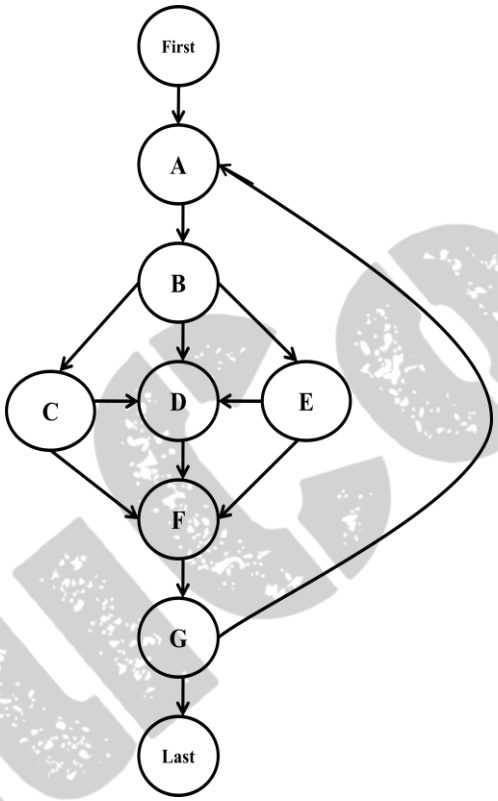


- Nodes **4 through 8** are a **sequence,** nodes **9 through 12** are an **if-then-else** construct, and nodes **13 through 22** are **nested if-then-else** constructs. Nodes **4** and **23** are the **program source** and **sink nodes**, corresponding to the **single-entry, single-exit** criteria. **No loops** exist, so this is **directed acyclic graph**.

- **Test cases** force the execution of **program paths**. we have very explicit description of the **relationship** between a **test case** and **the part of the program** it exercises.
- We also have an **elegant, theoretically respectable** way to **deal** with the potentially **large number of execution paths** in a program.

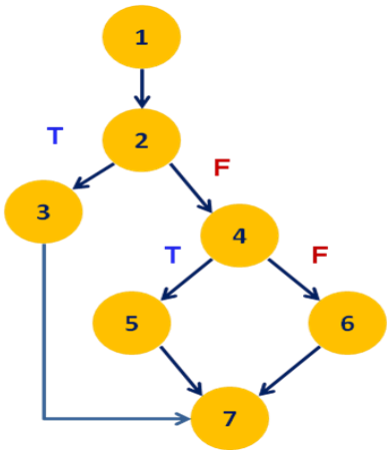
❖ **Figure below** is a graph of a **simple (but unstructured)** program. It is typical kind of **example** used to show the **impossibility of completely testing** even simple programs. In this program, **five paths** lead from **node B** to **node F** in the interior of the loop. If the loop may have to repeat for **18 times,** some **4.77 trillion** distinct program execution paths exist.

Figure: Trillions of paths.



Example 2

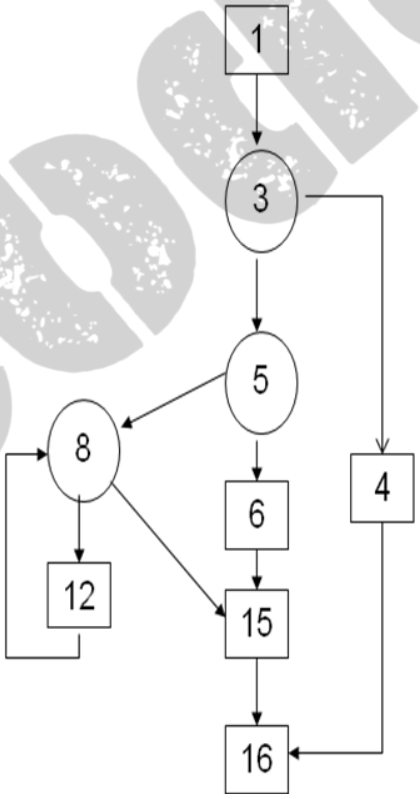
```
      int MAX3(int a, int b, int c)
       {
1           int max;
2            if( a > b && a > c)
3               max=a;
4            else if(b > c)
5               max=b;
6            else max=c;
7           return max;
      }
```



Example 3 (Study purpose)

```
function gcd (int a, int b) {
 1. int temp, value;
 2. a := abs(a);
 3. If (a=0 and b=0) then
 4.     raise exception
 5. else if (a=0) then
 6.     value := b;    // b is the GCD
 7. else
 8.     while (b ≠ 0 ) loop
 9.         temp := b;
10.         b := a mod b;
11.         a := temp;
12.     end loop
13.     value := a;
14. end if;
15. return value;
16. end gcd
```

Note: Based on Euclid's algorithm

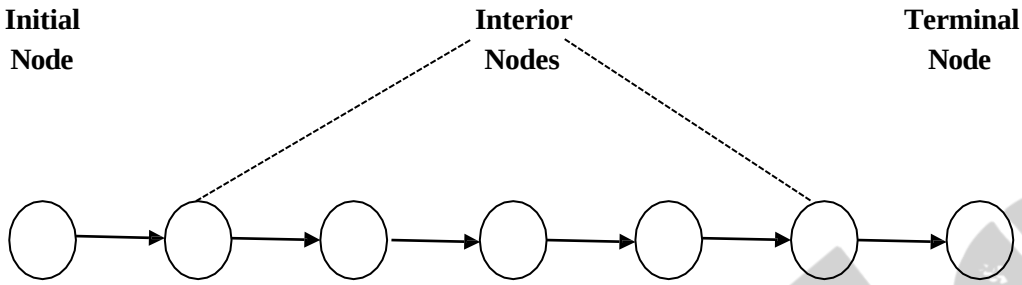


➢ DD Paths

- The **best-known form** of structural testing is based on a **construct** known as a **Decision-to-Decision path (DD-Path).**
- The name refers to a **sequence of statements** that, in **Miller's** words **begins** with the **"outway"** of a decision statement and **ends** with the **"inway"** of the next decision statement.
- We will **define DD-Paths** in terms of **paths of nodes** in a **directed graph.** We might call these paths as **chains,** where a **chain is a path** in which **the initial** and **terminal nodes are distinct,** and **every interior node** has **indegree = 1** and **outdegree = 1.**
- Notice that the **initial node is 2-connected to every other node** in the chain, and **no instances**

of **1-connected** or **3-connected nodes** occur, as shown in **below figure.** The **length** (number of edges) of the chain in **below figure** is **6.**

Figure: A chain of nodes in a directed graph.



Definition

A **DD-Path** is a **sequence of nodes** 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.**

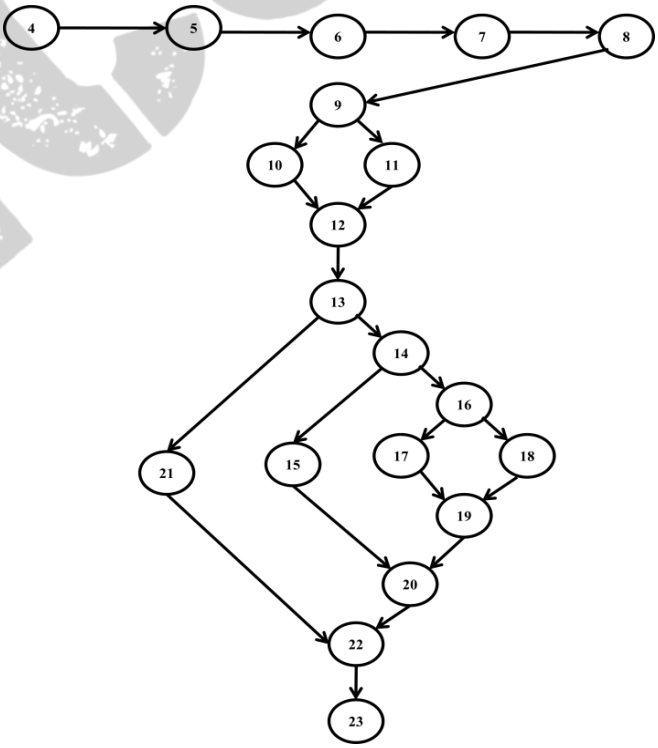
Case 3: It consists of a **single node** with **indeg ≥ 2 OR outdeg ≥ 2.**

Case 4: It consists of a **single node** with **indeg =1 AND outdeg=1.**

Case 5: It is a **maximal chain of length ≥ 1.**

Table: Types of DD-Paths for Triangle problem (table *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



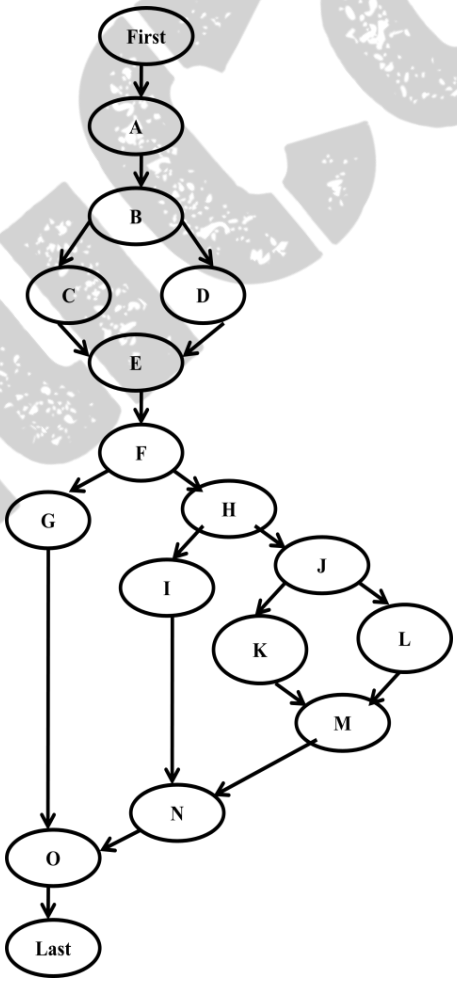
This is a **complex definition**, so we will apply it to the program graph above.

- Node 4 is a **case 1** DD-Path; we will call it **first**.
- Similarly, node 23 is a **case 2** DD-Path; we will call it **last**.
- Nodes 5 through 8 are **case 5** DD-Paths.We know that node 8 is the last node in this DD-Path because it is the last node that preserves the **2-connectedness** property of the chain.
- If we stop at node 7, we violate the **"maximal"** criterion.
- Nodes 10, 11, 15, 17, 18, and 21 are **case 4** DD-Paths.
- Nodes 9, 12, 13, 14, 16, 19, 20, and 22 are **case 3** DD-Paths.

Definition: Given a program written in an imperative language, its **DD-Path graph** is a **labeled directed graph**, in which **nodes** are **DD-Paths** of **its program graph**, and **edges** represent **control flow** between **successor DD-Paths.**

It is also known as **Control Flow Graph.** DD-Path is a **condensation graph.** For example, **2-connected** program graph nodes are **collapsed** to a **single DD-Path graph node** as shown in the **table *1** and **below figure.**

Figure: DD-Path graph for Triangle problem



➢ Test coverage metrics

- The motivation of using DD-paths is that they **enable** very **precise descriptions** of test coverage.
- In our quest, to **identify gaps** and **redundancy** in the test cases as these are used to exercise (test) different aspects of a program, we use **formal models** of the **program structure** to reason about testing effectiveness.
- **Test coverage metrics** are a **device to measure** the **extent to which a set of test cases covers a program**.
- Several widely accepted test coverage metrics are used, most of those are in **Table below (Miller, 1977).** Having an organized view of the **extent** to which a **program is tested** makes it possible to **sensibly manage** the testing process.
- Most quality organizations now expect the **C_1 metric** (DD-Path coverage) as the **minimum acceptable level** of test coverage. **Less adequate**, the **statement coverage metric (C_0)** is still widely accepted.

Table: Structural Test coverage metrics

Metric	Description of Coverage
C_0	Every Statement i.e., All statements
C_1	Every DD-Path, i.e., All Decisions
C_{1P} or C_{CC}	Every predicate to each outcome, i.e., each individual condition in the decision is tested for both true and false results
C_2	C_1 Coverage + loop coverage
C_d	C_1 Coverage + every dependent pair of DD-Paths, i.e., Define/Usage paths
C_{MCC}	Multiple condition coverage, i.e., all possible combinations of condition outcomes in each decision
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, $C_\infty \rightarrow C_1 \rightarrow C_0$

✓ Metric – Based Testing

❖ Statement and Predicate Coverage

- **Statement coverage** based testing aims to **devise test cases** that collectively exercise **all statements** in a program - **C_o**
- **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 - C_1**

- Here the term simple predicate refers to either a **single predicate** or a **compound Boolean expression** that is considered as a **single unit** that evaluates to **True or False.** This amounts to **traversing every edge** in the **DD-Path graph.**
- **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).**

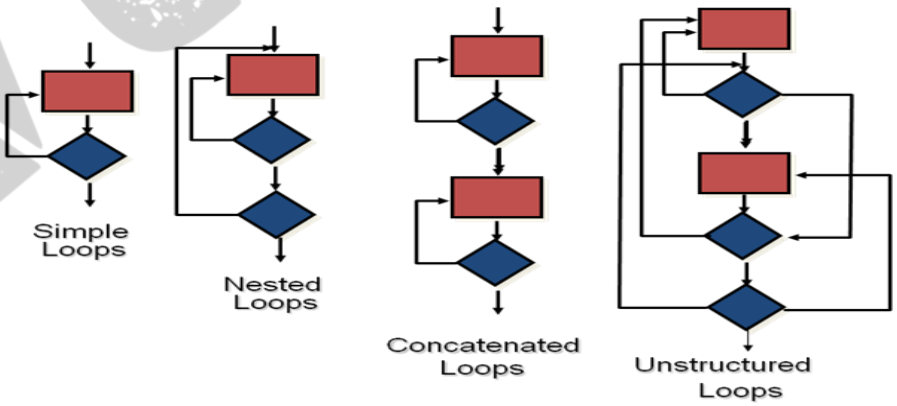
❖ **Condition Testing - C_{1P}**

- Decision coverage is good for **exercising faults** in the way a computation has been **decomposed** into cases.
- Condition coverage takes this decomposition in **more detail**, **forcing execution** of not only possible outcome of a Boolean expression but also of **different combinations** of the individual conditions in **compound Boolean expression.**
- A **test suite T** for a **program P** covers all **C_{1P}** iff for **each atomic condition** in P, it has **at least two test cases** in **T:** one forcing P to have **true out come** and the other one forcing P to have a **false outcome.**

❖ **Loop Coverage - C_2**

- 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.
- Once a loop is tested, then the tester can **collapse it into a single node** to simplify the graph for the next loop tests. In the case of nested loops we start with the **inner most** loop and we **proceed outwards**.
- If loops are **knotted / unstructured** then we must apply **data flow analysis** testing techniques.
- **Loop Testing** focuses exclusively on the **validity of loop** constructs.

Figure: Types of Loops



❖ **Dependent DD-Path Pairs Coverage - C_d**

- In simple **C_1** coverage criterion we are interested simply to traverse **all edges** in the DD-Path graph.
- If we enhance this coverage criterion by ensuring that we also **traverse dependent pairs of DD-Paths,** also we may have the **chance of revealing more errors** that are based on **data flow** dependencies.
- More specifically, **two DD-Paths** are said to be **dependent** iff there is a **define/reference relationship** between these DD-Paths, in which a variable is defined (receives a value) in one DD-Path and is referenced in the other.
- In **C_d** testing we are interested on covering **all edges** of the DD-Path graph and all **dependent DD-Path pairs**.
- Eg: **C and H nodes** are **dependent** in triangle DD-path graph. In node C **IsATriangle** is True, then control passes to H node. But node D and H are not dependent.

❖ **Multiple (Compound) Condition Coverage - C_{MCC}**

- In **C_{MCC}**, a more complete extension that includes both the **basic condition** and **branch adequacy** criteria
- C_{MCC} requires a **test case T** for each possible evaluation of compound conditions.
- For **N basic** conditions, **2^N combinations** of test cases are required..
- For a compound predicate **P1** (A or B), C_{MCC} requires that each possible combination of inputs be tested for each decision. **Example: if (A or B)** requires,
  A = True/False , B = True/False (4 Combinations of test cases)

❖ **Every program path that contains upto K repetitions of a loop -C_{ik}**

❖ **Statistically Significant Path Coverage Testing -C_{stat}**

- **Exhaustive testing** of software is **not practical** because variable input values and variable sequencing of inputs result in **too many possible combinations** to test.
- **NIST** developed techniques for applying **statistical methods** to derive sample test cases would address how to select the **best sample of test cases** and would provide a statistical **level of confidence** or **probability** that a program implements its functional specification correctly.
- The goal of statistically significant coverage is to develop methods for software testing based on statistical methods, such as **Multivariable Testing, Design of Experiments,** and **Markov Chain usage model**s, and to develop methods for software testing based on **statistical measures** and **confidence levels.**

❖ **Path testing - C_∞ (or P_∞)**

- A **test suite T** for a **program P** satisfies the **path adequacy criterion** iff, for each path **p_i of P,** there exists **at least one test case in T** that causes the **execution of p_i.**
- This is same as stating that, **every path** in the flow graph model of **program P** is exercised by **at least one test case** in T.

➢ Basis Path Testing

- Was proposed by **Tom McCabe.**
- Focused on test techniques that are based on the **selection of test paths** through a **program graph.**
- **Set of paths** should be properly chosen to measure the **test thoroughness**.
- **Fault assumption:** If something has gone wrong with the software, makes it **take a different path** than the one intended.
- Structurally, a **path** is a **sequence of statements** in a program unit.
- Semantically, a **path** is an **execution instance** of the program unit.
- **Objective** is to ensure that, **each path** through the program is **executed at least once** from the set of test cases

✓ Basis Path Testing – Motivation

- If we consider the **paths in a program graph** (or DD-Graph) to form a **vector space V**, we are interested to **devise a subset of V** say **B that captures the essence of V.** That is every element of **V** can be r**epresented as a linear combination of elements of B.**
- **Addition of paths** means that **one path is followed by another** and **multiplication** of a number by a path denotes the **repetition of a path.**
- If such a **vector space B** contains **linearly independent paths** and forms a **"basis"** for **V,** then it certainly captures the **essence of V.**

✓ Basis Path Testing - Process

- **Input**
  - **Source code** and **a path selection** criterion
- **Process**
  - **Generation/Construction** of a **CFG** using the design or code as a foundation.
  - Determine the **Cyclomatic Complexity** of the resultant flow graph that compute measure of the unit's logical **complexity.**
  - Determine a **basis set of linearly independent paths** using the measure.
  - **Selection** of Paths.
  - **Prepare test cases** that will force execution of each path.
  - **Feasibility Test** of a Path.
  - **Evaluation of Program's Output** for the Selected Test Cases

✓ Cyclomatic Complexity: McCabe's Cyclomatic Number V(G)

- Introduced in **1976 by McCabe**, is one of the most commonly used metrics in software development.
- Provides a **quantitative measure** of the **logical complexity** of a program in terms of **Cyclomatic number.**
- Defines the number of **independent paths** in the basis set.

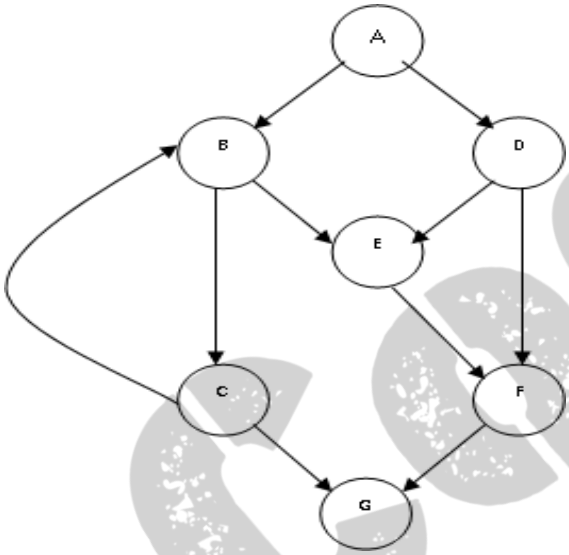
- Provides an **upper bound** for the **number of tests** that must be conducted to **ensure** all statements have been executed at least once.
- The Cyclomatic Complexity of the program V(G), can be computed from its **Control Flow Graph (CFG) G** in two ways,

  V(G)=e-n+2p

  V(G)=e-n+p

  where **e=no. of edges, n=no. of nodes, p=no. of connected regions.**

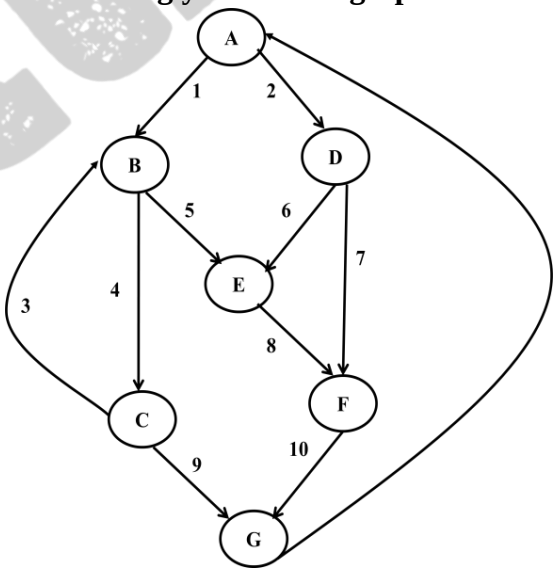
Figure: McCabe's control graph



- The number of **linearly independent paths** for the **above graph** is **V(G)=e-n+2p=10-7+2(1)=5**
- The number of **linearly independent circuits** for the graph in **figure below** is

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

Figure: McCabe's derived strongly connected graph.



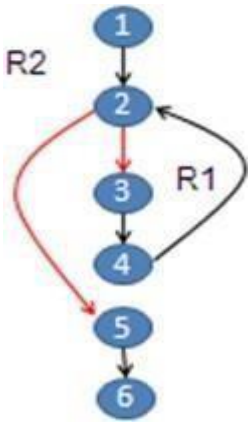
- There are **five** linearly **independent paths.**
  P1: A, B, C, G
  P2: A, B, C, B, C, G
  P3: A, B, E, F, G
  P4: A, D, E, F, G
  P5: A, D, F, G
- These paths can be made to look like a **vector space** by defining **notions** of **addition and scalar multiplication** ie., path addition is simply one path followed by another path, and multiplication corresponds to repititions of a path.With this formulation, the path **A,B,C,B,E,F,G** is the **basis sum** of **p2+p3-p1**, and the path **A,B,C,B,C,B,C,G** is the **linear combination of 2p2-p1.**
- The entries in this **table below** are obtained by following a path and noting which edges are traversed. **Example:** Path p1 traverses edges **1, 4,** and **9,** while path p2 traverses the following edge sequence: **1, 4 ,3, 4, 9.** Because edge 4 is traversed **twice** by path p2, the entry for the edge column is made **2**. The table is called as **incidence matrix.**

Table: Path/Edge Traversal

Path/Edge 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

- The i**ndependence** of the **paths p1 to p5** can be checked by examining the **first five rows** of the **incidence matrix.** The **bold entries** show edges that appear in exactly one path, so **paths p2 to p5** are **independent.** Path p1 is independent of all of these, because any attempt to express p1 in terms of the others, introduce unwanted edges, so **all five paths** are independent. At this point, we might check the **linear combinations** of the two example paths.
- McCabe next develops an **algorithmic procedure** called the **baseline method** to determine a set of basis paths. The method begins with the selection of a baseline path, which should correspond to some **"normal case"** program execution. This can be somewhat arbitrary. McCabe advises choosing a path with as many decision nodes as possible. Next, the **baseline path** is retraced, and in turn each decision is **"flipped";** that is, when a node of **outdegree ≥ 2** is reached, a different edge must be taken.

Example 1 (For practice)



Two linearly independent Paths:

P1: 1, 2, 5, 6

P2: 1, 2, 3, 4, 2, 5, 6

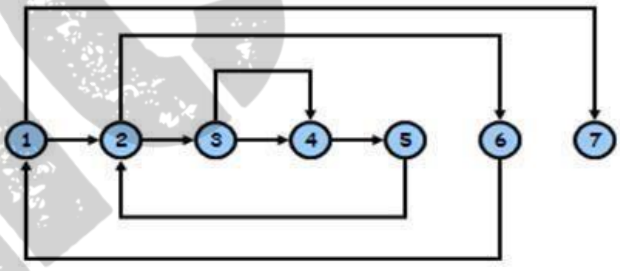
V(G) = E-N+2P
V(G) = 6-6+2(1) = 2

Example 2

Bubble Sort Algorithm

```
1 for (j=1; j<N; j++) {
    last = N - j + 1;
2     for (k=1; k<last; k++) {
3         if (list[k] > list[k+1]) {
              temp = list[k];
              list[k] = list[k+1];
              list[k+1] = temp;
4         }
5     }
6 }
7 print("Done\n");
```

#linearly independent paths
i.e., V(G) = 9-7+2 = 4



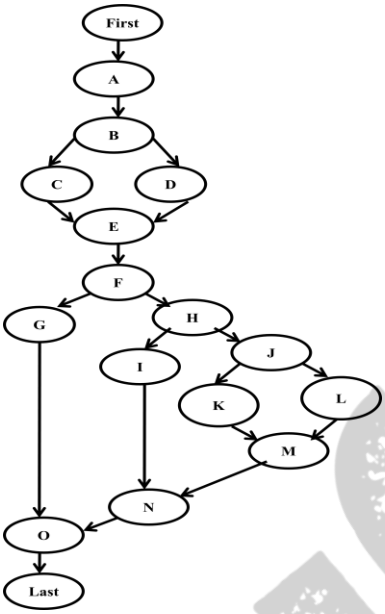
✓ Observations on McCabe's basis path Method

- Let us consider the **DD-path graph** of the triangle program which has **five basis paths**. Paths **p2 and p3 are infeasible.**

Table: Basis Path for Figure below (triangle problem)

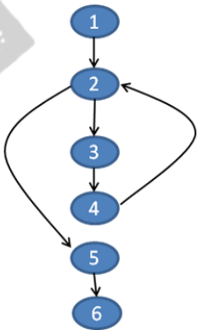
Original	P1: A-B-C-E-F-H-J-K-M-N-O-Last	Scalene
Flip p1 at B	**P2: A-B-D-E-F-H-J-K-M-N-O-Last**	**Infeasible**
Flip p1 at F	**P3: A-B-C-E-F-G-O-Last**	**Infeasible**
Flip p1 at H	P4 : A-B-C-E-F-H-I-N-O-Last	Equilateral
Flip p1 at J	P5: A-B-C-E-F-H-J-L-M-N-O-Last	Isosceles

- **Path p2 is infeasible**, because paasing through **node D** means the sides are **not triangle;** so the outcome of the decision at **node F must be node G**. Simmilarly, in p3, passing through **node C** means the sides **do form a triangle**, so **node G cannot** be traversed. Other paths are feasible and produce corresponding results.
- We can identify two rules:
  - If **node C is traversed**, then we must **traverse node H.**
  - If **node D is traversed**, then we must **traverse node G.**
- The **logical dependencies** reduce the **size of a basis set** when **basis paths** must be **feasible** as **shown below.**



P1: A-B-C-E-F-H-J-K-M-N-O-Last	**Scalene**
P6: A-B-D-E-F-G-O-Last	**Not a Triangle**
P4 : A-B-C-E-F-H-I-N-O-Last	**Equilateral**
P5: A-B-C-E-F-H-J-L-M-N-O-Last	**Isosceles**

Example 1 (For practice)

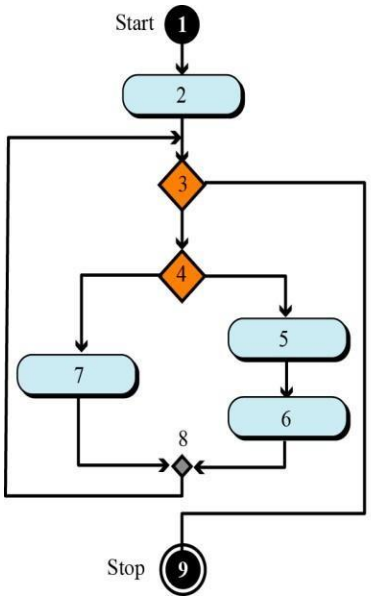


Many paths exists between **1 (begin) and 6 (end)**

1, 2, 5, 6
1, 2, 3, 4, 2, 5, 6
1, 2, 3, 4, 2, 3, 4, 2, 5, 6
…

Prepare test cases that will force the **execution of each path** in the **basis set.**
Test case : **{ (inputs …) , (expected outputs …) }**

Example 2

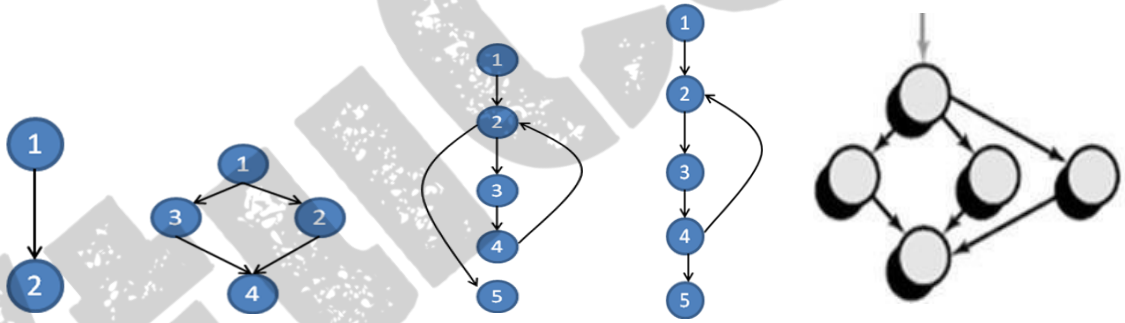


Independent path:

Path1: (1, 2, 3, 9)
Path2: (1, 2, 3, 4, 5, 6, 8, 3, 9)
Path3: (1, 2, 3, 4, 7, 8, 3, 9)

✓ Essential Complexity

- Essential complexity is the **cyclomatic complexity** of yet another form of **condensation graph**. The basic idea is to **collapse the structured programming constructs into single node,** and **repeat until no more** structured programming constructs can be found.
  Figure: Structured programming Constructs



A sequence:

X = 1;
Y = X * 10;

If condition:

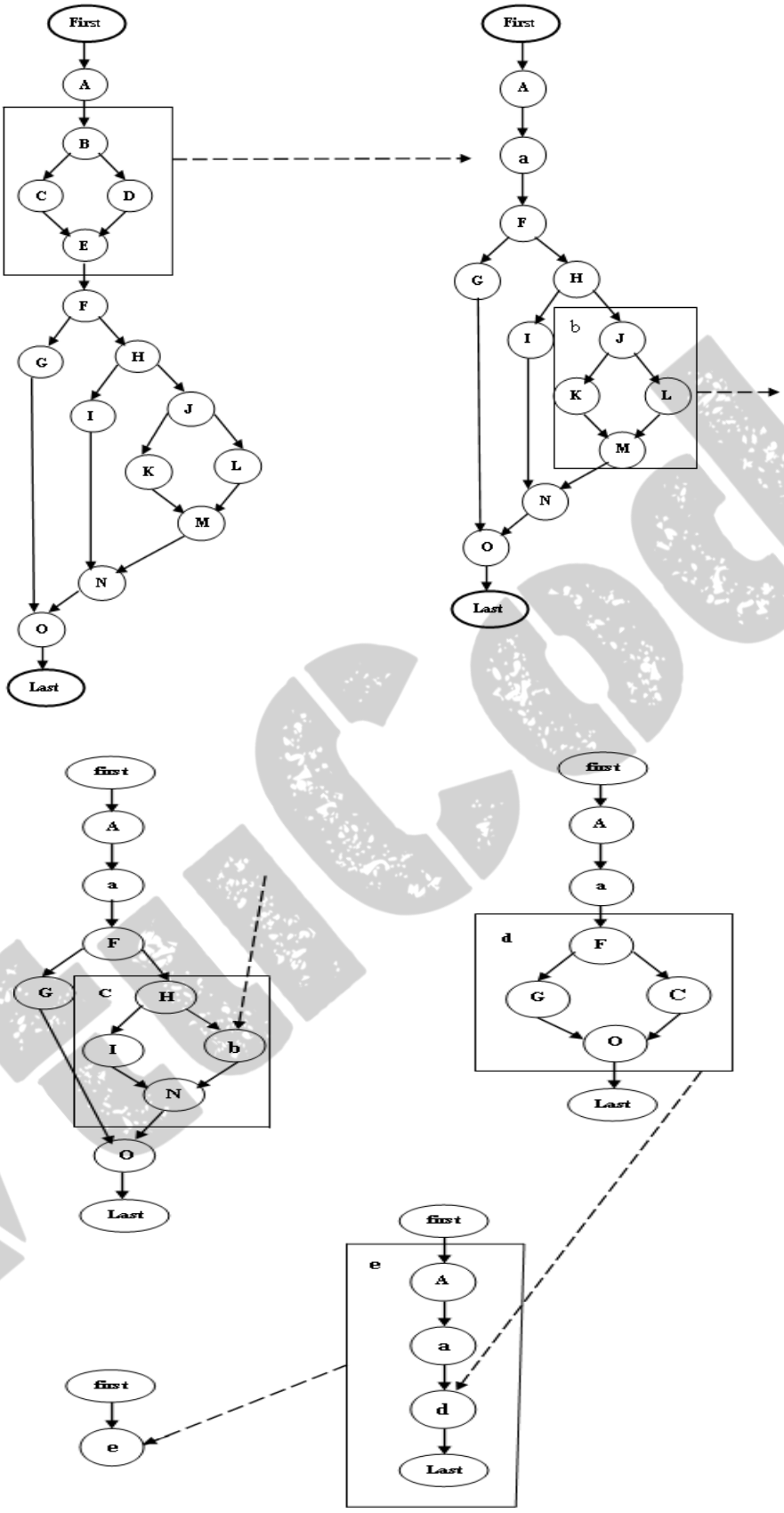
If ... Then
...
Else
...
End if

While loop:
While ... do
...
statements
...
End while

Do While loop
(Repeat until):
do
...
statements
...
While ...

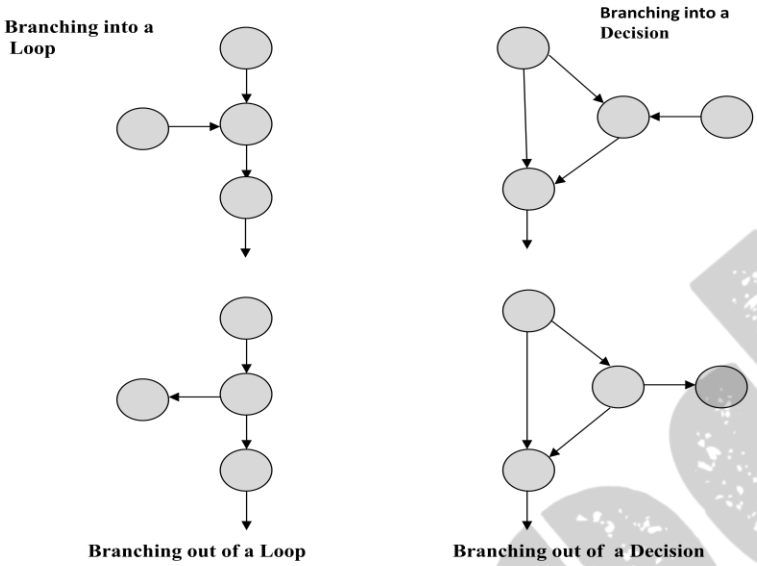
case (choice)
case 1:
case 2:
.
.
case: n
End case

Figure: Condensing with respect to the Structured Programming Constructs



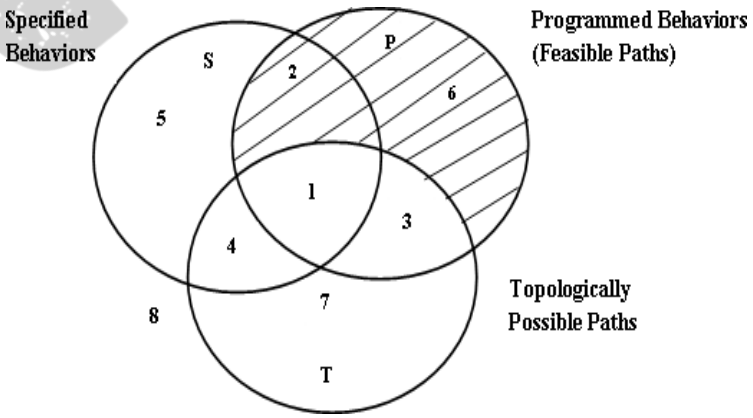
- Finally we have graph with **cyclomatic complexity V(G)=1,** ie., when program is **well structured** it can always be reduced to a **graph with one path**.

Figure: Violations of Structured Programming



➢ Guidelines and Observations

- Basis path testing gives us a **lower boundary** on **how much** testing is necessary.
- Path-based testing **provides** us with a **set of metrics** that act as **crosschecks** on specification - based testing. We can use these metrics to **resolve** the **gaps** and **redundancies** questions.
- When we find that the **same program path** is traversed by several functional **test cases**, we suspect that this **redundancy** is **not revealing new faults.** When we fail to attain **DD-path coverage,** we know that there are gaps in the functional test cases.
- **The below figure** shows the relationship among **S, P, and T**.
  Figure: Feasible and topologically possible paths



- **Region 1 is most desirable as it contains specified behaviors that are implemented by feasible paths.**
- **Region 2 and 6 must be empty because all feasible paths are in region 1 .**
- **Region 3 contains feasible paths that correspond to unspecified behaviors.**
- **Region 4 and 7 contains infeasible paths.**
- **Region 5 corresponds to specified behaviors that have not been implemented.**
- **Region 7 is unspecified, infeasible, yet topologically possible paths.**

Dataflow Testing

➢ Dataflow Testing

- **Data flow** testing is a term, which is having **no connection** with **dataflow diagrams**. Data flow testing refers to **forms of structural testing** that focus on the points **at which variables receive values** and **the points at which these values are used** (or referenced). We will see that data flow testing **serves** as a **reality check** on path testing.
- Most programs **deliver functionality** in terms of **data. Variable**s that represent data somehow receive **values**, and these values are used to **compute values** for **other variables**. Since early 1960s, programmers have analyzed source code in terms of the **points** (statements) at which variables **receive values** and points at which these **values are used**. Early data flow analysis often centered on a **set of faults** that are now known as **define/reference anomalies**:
  - A variable that is **defined but never used** (referenced).
  - A variable that is **used but never defined**.
  - A variable that is **defined twice** before it is used
- Each of these anomalies can be recognized from the **concordance** of a program. Because the concordance information is **compiler generated**, these anomalies can be discovered by what is known as **static analysis,** that is **finding faults in source code without executing it.**
- Data flow testing can be performed at **two conceptual levels**.
  - **Static data flow testing**
    - Identify potential defects, commonly known as **Data Flow Anomaly.**
    - **Analyze** source code.
    - Do not **execute code.**
  - **Dynamic data flow testing**
    - Involves actual **program execution.**
    - Bears similarity with control flow testing, ie., **Identify paths** to execute and paths are identified based on **data flow testing criteria.**
- There are **two major forms** of dataflow testing
  - **Define/Use testing**
    - **paths** to the **locations** and properties of **references** to variables **within** the program code.
    - a program can be **analyzed** in terms of how the **variables** are **affected**, **assigned** and **changed** throughout the course of the program.
  - **Slice- Based Testing**
    - **"Slice"** the program into a **number of individually executable** components
    - Each focusing on **one particular variable** at **one particular location** within the program.
    - Slicing techniques which are used especially on **larger projects** with **large teams** of developers.

➢ Define/Use Testing

- Much of the formalization of **define/use** testing was done in the early 1980s. It presumes a program graph in which **nodes** are **statement fragments** (a fragment may be an entire statement), and **programs** that follow the **structured programming precepts**.
- The following definitions refer to a **program P** that has a **program graph G(P),** and a set of program **variables V**.
  - **Nodes** correspond to program statements.
  - **Edges** correspond to the flow of information.
  - **Program Graph G(P)** with respect to **program(P)** has,
    - **Single entry** and **single exit** node.
    - **no edges** from **node to itself.**
    - Set of program **variables : V**.
    - Set of all **paths in P** : **PATHS(P)**

- **Definitions**

  - Node n ∈ G (P) is a **defining node** of the variable **v ∈ V,** written as **DEF (v, n),** iff the value of the variable v is defined at the statement fragment corresponding to **node n.**

    Ex : Input (x)

    Ex : x = 20

    Input statements, assignment statements, loop control statements, and **procedure calls** are all examples of statements that are **defining nodes**.

    When the code corresponding to such statements executes, the **contents** of the **memory location(s)** associated with the **variables are changed**.

  - **Definition**

    Node **n ∈ G (P)** is a **usage node** of the variable **v ∈ V,** written as **USE (v, n)**, iff the value of the **variable v** is used at the **statement fragment** corresponding to **node n.**

    Output statements, assignment statements, conditional statements, loop control statements, and **procedure calls** are all examples of statements that are **usage nodes.**

    When the code corresponding to such statements executes, the contents of the **memory location(s)** associated with the **variables remain unchanged**.

  - **Definition**

    A usage node **USE(v, n)** is a **predicate use**, denoted as **P-use,** iff the **statement n** is a predicate statement, otherwise **USE(v, n)** is a **computation use**, denoted as **C-use.**

    The nodes corresponding to predicate uses always have an **out degree ≥ 2,** and nodes corresponding to **computation uses** always have **out degree ≤ 1.**

- **Definition**

  Definition-use (du-path): A **definition-use path** with respect to a **variable v,** denoted as **du-path,** is a path in **PATHS(P)** such that, for some **v ∈ V,** there are **define and usage nodes DEF(v, m)** and **USE(v, n),** where **m** and **n** are the **initial** and **final nodes** of the path.

- **Definition**

  Definition-clear (dc-path): A **definition-clear path** with respect to a **variable v,** denoted as **dc-path**, is a **definition-use** path in **PATHS (P)** with **initial** and **final** nodes **DEF (v, m)** and **USE (v, n)** such that **no other node** in the path is a **defining node of v.**

Testers have to notice **how these definitions capture the essence of computing with stored data values**. **Du-paths** and **dc-paths** describe the flow of data across source statements from points at which the values are defined to points at which the values are used. **Du-paths that are not definition-clear are potential trouble spots.**

✓ Example

This program computes the commission on the sales of the total numbers of locks, stocks, and barrels sold. The While-loop is a classical sentinel controlled loop in which a value of -1 for locks signifies the end of the sales data. The totals are accumulated as the data values are read in the while loop. After printing this preliminary information, the sales value is computed, using the constant item prices defined at the beginning of the program. The sales value is then used to compute the commission in the conditional portion of the program.

Data Flow testing: key steps

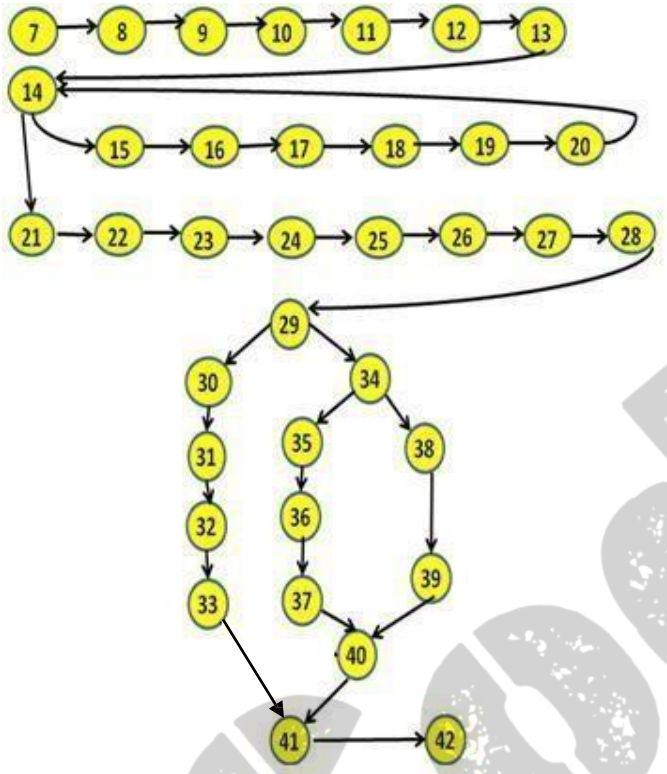
Given a code (program or pseudo-code), the steps to be followed are:

1. **Numbe**r the lines
2. **List** the variables
3. **List** occurrences & assign a category to each variable
4. Identify **du-pairs** and their **use** (p- use or c- use)
5. Define **test cases**, depending on the required coverage

1. Program commission (INPUT, OUTPUT)
2. Dim locks, stocks, barrels As Integer
3. Dim lockprice, stockprice, barrelprice As Real
4. Dim totalLocks, totalStocks,totalBarrels as Integer
5. Dim lockSales, stockSales, barrelSales As Real
6. Dim sales, commission As Real
7. Lockprice = 45.0
8. Stockprice = 30.0
9. barrelPrice = 25.0
10. totalLocks = 0

```
11.  totalStocks = 0
12.  totalBarrels = 0
13.  Input (locks)
14.  While NOT (locks = -1)    //Input device uses -1 to indicate end of data
15.     Input (stocks, barrels)
16.     totalLocks = totalLocks + locks
17.     totalStocks = totalStocks + stocks
18.     totalBarrels = totalBarrels + barrels
19.     Input (locks)
20.  EndWhile
21.  Output ("Locks sold : ", totalLocks)
22.  Output ("Stocks sold : ", totalStocks)
23.  Output ("Barrels sold : ", totalBarrels)
24.  lockSales = lockPrice * totalLocks
25.  stockSales = stockPrice * totalStocks
26.  barrelSales = barrelPrice * totalBarrels
27.  sales = lockSales + stockSales + barrelSales
28.  Output ("total sales : ", sales)
29.  If (sales > 1800.0)
30.    Then
31.          Commission = 0.10 * 1000.0
32.          Commission = commission + 0.15 * 800.0
33.          Commission = commission + 0.20 * (sales-1800.0)
34.     Else If (sales > 1000.0)
35.        Then
36.          Commission = 0.10 * 1000.0
37.          Commission = commission + 0.15 * (sales – 1000.0)
38.     Else
39.          Commission =0.10 * sales
40.     EndIf
41.  EndIf
42.  Output ("commission is $ ", commission)
43.  End commission
```

Figure: Program graph of the Commission program.



- **Figure below** shows the **decision-to-decision path** (DD-Path) **graph** of the **program graph** given **above.** More compression exists in this DD-Path graph because of the increased computation in the commission problem. **Table below** details the statement fragments associated with DD-Paths. Some DD-Paths are combined to simplify the graph.

Figure:DD-path graph of the Commission program**.**

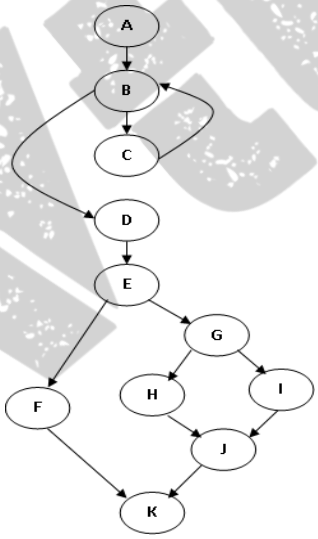


Table: DD-Paths for Figure above

DD-Path	Nodes
A	**7, 8, 9, 10,11, 12, 13,**
B	**14**
C	**15, 16, 17, 18, 19, 20**
D	**21, 22, 23, 24, 25, 26, 27, 28**
E	**29**
F	**30, 31, 32, 33**
G	**34**
H	**35, 36, 37**
I	**38, 39**
J	**40**
K	**41, 42, 43**

Table below lists **define** and **usage nodes** for the **variables** in the commission problem. We use this information in conjunction with the program graph to identify various **definition-use** and **definition-clear** paths.

- We will refer to the various **paths** as **sequences of node numbers.**

Step 2 and 3: **Define/Use Nodes** for variable in the commission problem

Variable	Defined at Node	Used at Node
Lock price	7	24
Stock price	8	25
Barrel price	9	26
Total Locks	10,16	16,21,24
Total Stocks	11,17	17,22,25
Total Barrels	12,18	18,23,26
Locks	13,19	14,16
Stocks	15	17
Barrels	15	18
LockSales	24	27
StockSales	25	27
BarrelSales	26	27
Sales	27	28,29,33,34,37,39
commission	31,32,33,36,37,39	32,33,37,42

- **Table below** presents some of the **du-paths** in the commission problem. They are named by their **beginning** and **ending nodes** (from fig. **Program graph of the Commission program)**
- The **third column** in table indicates whether the **du-paths** are **definition-clear.**
- The **while loop** (node sequence <14, 15, 16, 17, 18, 19, 20>) inputs and accumulates values for **totalLocks, totalStocks, totalBarrels**.
- The initial value definition for **totalStocks** occurs at **node 11**, and it is first used at **node 17.** Thus, the **path(11,17),** which consists of the **node sequence <11,12,13,14,15,16,17>,** is **definition clear.**
- The **path (11, 22),** which consists of the node sequence <11,12,13, (14, 15, 16, 17, 18, 19, 20)*, 21, 22> is **not definition-clear** because values of totalStocks are **defined at node 11** and at **node 17.**

Step 4: Identify du-pairs and their use (p-use or c-use)

Selected Define/Use paths

`

Variable	Path (Beginning,End Nodes)	Definition-Clear?
lockprice	7, 24	Yes
stockprice	8, 25	Yes
barrelprice	9, 26	Yes
totalStocks	11, 17	Yes
totalStocks	11, 22	No
totalStocks	11, 25	No
totalStocks	17, 17	Yes
totalStocks	17, 22	No
totalStocks	17, 25	No
locks	13, 14	Yes
locks	13, 16	Yes
locks	19, 14	Yes
locks	19, 16	Yes
sales	27, 28	Yes
sales	27, 29	Yes
sales	27, 33	Yes
sales	27, 34	Yes
sales	27, 37	Yes
sales	27, 39	Yes

✓ du-paths for Stocks

First, let us look at a simple path: the du-path for the variable **stocks**. We have **DEF (stocks, 15)** and **USE (stocks, 17),** so the **path <15, 17>** is a du-path with respect to **stocks**. No other defining nodes are used for stocks, therefore this path is **definition clear.**

✓ du-paths for Locks

Two defining and **two usage nodes** make the **locks** variable more interesting. We have **DEF (locks, 13), DEF (locks, 19), USE (locks, 14),** and **USE (locks, 16).** These yield four du-paths:

P1 = <13, 14>
P2 = <13, 14, 15, 16>
P3 = <19, 20, 14>
P4 = <19, 20, 14, 15, 16>

- Du- paths **p1 and p2** refer to the priming value of locks, which is read at **node 13. Locks** has a **predicate use** in the While Statement **(node 14),** and if the condition is true (as in path p2), a **computation use** at **statement 16.**

- The other **two du-paths** start near the **end of the While loop** and occur when the loops repeats.
- These four paths provide the **loop coverage** – **bypass the loop, begin the loop, repeat the loop, and exit the loop.** All these du-paths are **definition-clear.**

✓ du-paths for total Locks

- The du-paths for **totalLocks** will lead us to typical test cases for computations.
- With two defining nodes **(DEF(totalLocks, 10)** an**d DEF (totalLocks, 16))** and three usage nodes **(USE(totalLocks, 16), (USE(totalLocks, 21), (USE(totalLocks, 24)),** we might expect six du-paths.
- Path p5 = <10,11,12,13,14,15,16> is a du-path in which the initial value of **tatalLocks = 0 (at line 16)** has a **computation use**. This path is **definition-clear.**
- The next path is problematic: P6 = <10,11,12,13,14,15,16,17,18,19,20,14,21>
- Path P6 ignores the possible repetition of the While loop. We could highlight this by nothing that the **subpath <16, 17, 18, 19, 20, 14, 15>** might be traversed several times. We still have a du-path which is **not definition-clear.**
- If a problem occurs with the value of **totalLocks at node 21** (the output statement), we should look at the intervening **DEF(totalLocks,16) node.**
- The next path contains **p6**. We can show this by using a **path name** in place of its corresponding node sequence:

  P7 = <10,11,12,13,14,15,16,17,18,19,20,14,21,22,23,24>
  P7 = <P6, 22, 23, 24>

- Du-path **P7 is not definition-clear** because it includes **node 16.**
- Subpaths that begin with **node 16** (an assignment statement) are interesting. The first, <16, 16>, seems degenerate. The remaining two du-paths are both subpaths of **P7:**

  P8 = <16, 17, 18, 19, 20, 14, 21>
  P9 = <16, 17, 18, 19, 20, 14, 21, 22, 23, 24>

- Both are **definition –clear,** and both have the loop iteration.

✓ du-paths for Sales

- **Only one defining node** is used for **sales,** therefore, all the du-paths with respect to sales must be **definition-clear**. They are interesting because they illustrate **predicate** and **computation uses**. The first **three du-paths** are easy:

  P10 = <27, 28>
  P11 = <27, 28, 29>
  P12 = <27, 28, 29, 30, 31, 32, 33>

- **Path P12** is a **definition-clear** path with **three usage nodes**; it also contains **paths P10 and P11.**

- The **IF, ELSE IF logic in statements 29 through 40 highlights an ambiguity** in the original research. Two choices for du-paths begin with **path p11:** one choice is the **path <27,28,29,30,31,32,33>,** and the other is the **path <27,28,29,34>.** The remaining du-paths for sales are:

P13 = <27, 28, 29, 34>
P14 = <27, 28, 29, 34, 35, 36, 37>
P15 = <27, 28, 29, 34, 38, 39>

✓ du-paths for Commission

- In **statements 29 through 41**, the calculation of **commission** is controlled by ranges of the variable **sales.** Statements **31 to 33** build up the value of commission by using the memory location to hold intermediate values.

Define/Use paths for Commission

Variable	Path(Beginning,End) Nodes	Feasible?	Definition-Clear?
commission	31,32	Yes	Yes
commission	31,33	Yes	No
commission	31,37	No	n/a
commission	31,42	Yes	No
commission	32,32	Yes	Yes
commission	32,33	Yes	Yes
commission	32,37	No	n/a
commission	32,42	Yes	No
commission	33,32	No	n/a
commission	33,33	Yes	Yes
commission	33,37	No	n/a
commission	33,42	Yes	Yes
commission	36,32	No	n/a
commission	36,33	No	n/a
commission	36,37	Yes	Yes
commission	36,42	Yes	No
commission	37,32	No	n/a
commission	37,33	No	n/a
commission	37,37	Yes	Yes
commission	37,42	Yes	Yes
commission	39,32	No	n/a
commission	39,33	No	n/a
commission	39,37	No	n/a
commission	39,42	Yes	Yes

- Now, it is decided to **disallow du-paths** from assignmet Statements like **31, 32** and **36**, so we will just consider du-paths that begin with the **three "real"** defining nodes: **DEF(commission, 33), DEF(commission, 37)** and **DEF(commission, 39).** Only one usage node is: **USE(commission, 42).**

✓ Du-path Test Coverage Metrics

- The whole point of analyzing a program with **definition-use paths** is to define a **set of test coverage metrics** known as the **Rapps-Weyuker data flow metrics** (Rapps and Weyuker, 1985).
- The first three of these are equivalent to **All-Paths, All-Edges, and All-Nodes.**
- The others presume that define and usage nodes have been identified for all program variables, and that du-paths have been identified with respect to each variable.

In the following definitions, **T is a set of paths** in the **program graph G(P)** of a **program P**, with the **set V of variables**. In the next definitions, we assume that the define/use paths are **all feasible**.

- **Definition**

  The set **T** satisfies the **All-Defs** criterion for the program **P** iff for every variable **v** ∈ **V, T** contains **definition-clear** paths from every defining **node of v** to a **use of v**.

- **Definition**

  The set **T** satisfies the **All-Uses** criterion for the program **P** iff for every variable **v** ∈ **V, T** contains **definition-clear** paths from every defining **node of v** to every **use of v**, and to the successor node of each **USE(v,n).**

- **Definition**

  The set **T** satisfies the **All-P-Uses /Some C-Uses** criterion for the program **P** iff for every variable **v** ∈ **V**, **T** contains **definition-clear** paths from every defining **node of v** to every predicate **use of v**, and if a definition of **v** has **no P-uses**, there is a **definition-clear** path to at least **one computation use**.

- **Definition**

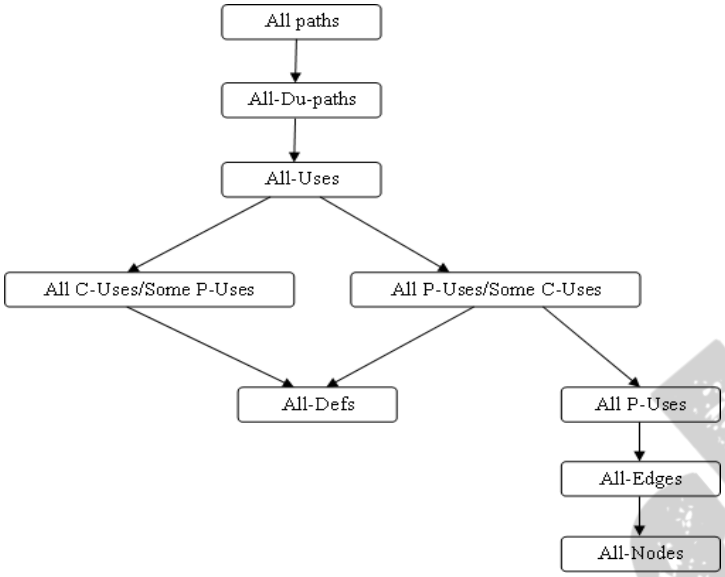
  The set **T** satisfies the **All-C-Uses /Some P-Uses** criterion for the program **P** iff for every variable **v** ∈ **V, T** contains **definition-clear** paths from every defining node of **v** to every computation use of **v**, and if a definition of **v has no C-uses**, there is a **definition-clear** path to at least **one predicate use**.

- **Definition**

  The set **T** satisfies the **All-DU-paths** criterion for the program **P** iff for every variable **v** ∈ **V**, **T** contains **definition-clear** paths from every defining node of **v** to every use of **v,** and to the **successor node** of each **USE(v, n),** and that these paths are either single loop traversals or they are cycle free.

  These **test coverage metrics** have several **set-theory** based relationships, which are referred to as **subsumption.** These relationships are shown in **Figure below.**

Figure: Rapps-Weyuker hierarchy of dataflow coverage metrics.



➢ Slice-Based Testing

- **Program slices** have **surfaced** and **submerged** in software engineering literature since the early 1980s. Informally, a **program slice** is **a set of program statements** that **contribute to, or affect** a **value for a variable** at **some point** in the program.
- We continue with the notation we used for define-use paths: a program **P** that has a program graph **G(P),** and a set of program variables **V.**

- **Definition**
  Given a program **P**, and a set **V** of variables in **P,** a **slice on the variable set V** at statement **n**, written **S(V,n),** is the **set of all statements** in **P** that contribute to the values of variables in **V** at node **n.**
  Listing elements of a slice **S(V,n)** will be cumbersome, because the elements are program statement fragments. Since it is much simpler to list fragment numbers in P(G), we make the following **trivial change:**
- **Definition**
  Given a program **P,** and a program graph **G(P)** in which statements and statement fragments are numbered, and a set **V** of variables in **P,** the **slice on the variable set V** at statement fragment **n**, written **S(V,n),** is the set of node numbers of all statement fragments in **P prior to n** that contribute to the values of variables in **V** at statement fragment **n**.
- The idea of slices is to **separate a program** into **components** that have some **useful meaning.**
- Slice captures the **execution time behavior** of a program with respect to the **variable(s)** in the **slice.**
- Eventually, we will develop a **lattice** (a directed, acyclic graph) of **slices**, in which **nodes are slices**, and **edges correspond** to the **subset relationship**.

- **Declarative statements** have an **effect** on the **value of a variable**. For now, we simply **exclude** all **non-executable** statements. The notion of contribution is partially clarified by the **predicate (P-use)** and **computation (C-use)** usage distinction, but we need to refine these forms of variable usage. Specifically, the **USE relationship** pertains to **five forms of usage:**

  P-use used in a predicate (decision)

  C-use used in computation

  O-use used for output

  L-use used for location (pointers, subscripts)

  I-use iteration (internal counters, loop indices)

  We identify two forms of definition nodes:

  I-def defined by input

  A-def defined by assignment

For now, assume that the slice S(V, n) is a slice on one variable, that is, the set V consists of a single variable, v.

- If statement fragment **n** is a **defining node for v**, then **n** is **included in the slice**.
- If statement fragment **n** is a **usage node for v**, then **n is not included in the slice**.
- **P-uses** and **C-uses** of other variables (**not the v** in the slice set V) are **included to the extent** that **their execution affects** the **value** of the **variable v.**
- If the **value of v** is the **same** whether a statement fragment is **included or excluded**, then **exclude that statement fragment.**
- **L-use** and **I-use** variables are typically **invisible** outside their modules
- **O-use, L-use,** and **I-use** nodes are **excluded** from slices.

✓ Example

- The **commission problem** is used because it contains **interesting** data flow properties.
- **Follow** these examples **while looking at the source code** for the commission problem that we used to **analyze** in terms of **define-use paths.**
- variable show why it is potentially **fault-prone**. It has a **P-use** at node 14 and a **C-use** at node 16, and has two definitions, the **I-defs** at nodes **13 and 19.**

  S1: S(locks, 13) = {13}
  S2: S(locks, 14) = {13, 14, 19, 20}
  S3: S(locks, 16) = {13, 14, 19, 20}
  S4: S(locks, 19) = {19}

- The slices for **stocks** and **barrels** are dull. They are short, **definition-clear** paths contained entirely within a loop, so they are **not affected by iteration** of the loop.

S5: S(stocks, 15) = {13, 14, 15, 19, 20}
S6: S(stocks, 17) = {13, 14, 15, 19, 20}
S7: S(barrels, 15) = {13, 14, 15, 19, 20}
S8: S(barrels, 18) = {13, 14, 15, 19, 20}

- The next four slices illustrate how **repetition appears** in slices. **Node 10** is an **A-def** for **totalLocks**, and **node 16** contains both an **A-def** and a **C-use**. The remaining nodes in **S10 (**13, 14, 19, and 20**)** pertain to the While-loop **controlled by locks**. Slices **S10** and **S11** are equal because **nodes 21** and **24** are an **O-use** and **a C-use** of **totalLocks** respectively.

S9: S(totalLocks, 10) = {10}
S10: S(totalLocks, 16) = {10, 13, 14, 16, 19, 20}
S11: S(totalLocks, 21) = {10, 13, 14, 16, 19, 20}

- The slices on **total-stocks** and **total-barrels** are quite similar. They are initialized by A**-defs at nodes 11** and **12,** and then are redefined by **A-defs at nodes 17** and **18.** Again, the remaining **nodes** (13, 14, 19 and 20) pertain to the While-loop **controlled by locks.**

S12: S(totalStocks, 11) = {11}
S13: S(totalStocks, 17) = {11, 13, 14, 15, 17, 19, 20}
S14: S(totalStocks, 22) = {11, 13, 14, 15, 17, 19, 20}
S15: S(totalBarrels,12) = {12}
S16: S(totalBarrels, 18) = {12, 13, 14, 15,18,19,20}
S17: S(totalBarrels, 23) = {12, 13, 14, 15,18,19,20}

- The next **six slices** demonstrate our convention regarding values defined by assignment statements **(A-defs).**

S18: S(lockPrice, 24) = {7}
S19: S(stockPrice, 25) = {8}
S20: S(BarrelPrice, 26) = {9}
S21: S(locksales, 24) = {7, 10, 13, 14, 16, 19, 20, 24}
S22: S(stocksales, 25) = {8, 11, 13, 14, 15, 17, 19, 20, 25}
S23: S(barrelsales, 26) = {9, 12, 13, 14, 15, 18, 19, 20, 26}

- The slices on **sales** and **commission** are the interesting ones. There is **only one defining node** for **sales,** the **A-def at node 27**. The remaining slices on sales show the **P-uses, C-uses**, and the **O-use** in **definition-clear** paths.

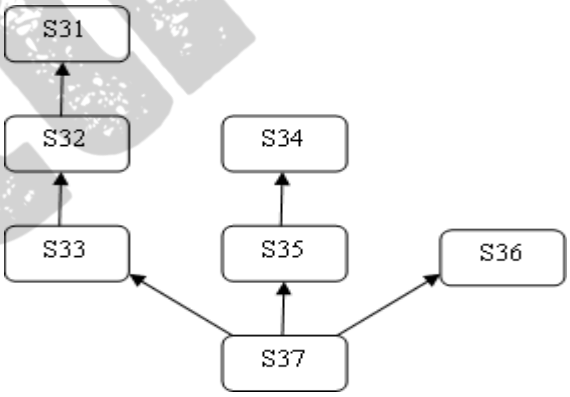
S24: S(sales, 27) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}
S25: S(sales, 28) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}
S26: S(sales, 29) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}
S27: S(sales, 33) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}
S28: S(sales, 34) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}
S29: S(sales, 37) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}
S30: S(sales, 39) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27}

- Think about **slice S24** in terms of its components, which are the slices on the C-use variables. We can write **S24 = S10 U S13 U S16 U S21 U S22 U S23 U {27}.** Notice how the formalism corresponds to our intuition: **if the value of sales is wrong, we first look at how it is computed, and if this is OK, we check how the components are computed.**
- Everything comes together with the slices on commission. There are six A-def nodes for commission (corresponding to the six du-paths we identified earlier). Three computations of commission are controlled by P-uses of sales in the IF, ELSE IF logic. This yields three "paths" of slices that compute commission.

S31: S(commission, 31) = {31}
S32: S(commission, 32) = {31, 32}
S33: S(commission, 33) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31, 32, 33}
S34: S(commission, 36) = {36}
S35: S(commission, 37) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 34, 35, 36, 37}
S36: S(commission, 39) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 34, 38, 39}

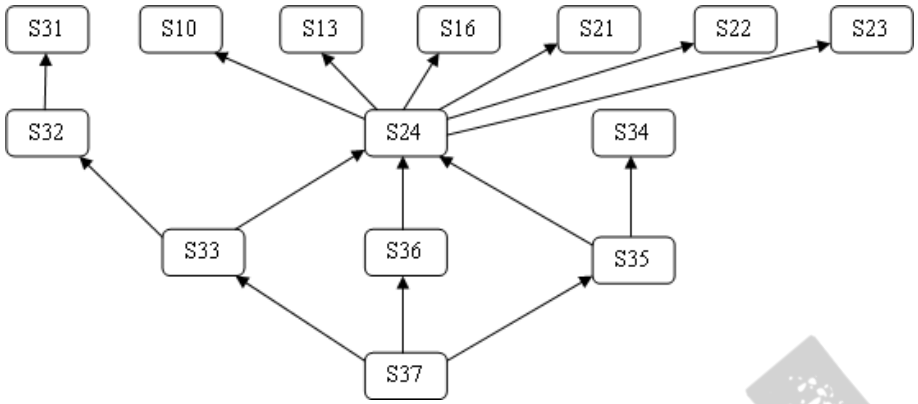
Whichever computation is taken, all come together in the last slice.
S37: S(commission, 42) = {7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 24, 25, 26, 27, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39}

- The slice information improves our insight. Look at the lattice in **Figure below**, it is a directed **acyclic graph** in which slices are nodes, and an edge represents the proper subset relationship.
Figure: Lattice of slices on commission.



This lattice is drawn so that the position of the slice nodes roughly corresponds with their position in the source code. The definition-clear paths <33, 41>, <37, 41>, and <39, 41> correspond to the edges that show slices S33, S35, and S36 are subsets of slice S37.
Figure below shows a lattice of slices for the entire program. Some slices (those that are identical to others) have been deleted for clarity.

Figure: Lattice on sales and commission.



✓ Style and Technique

When we analyze a program in terms of "interesting" slices, we can focus on parts of interest while disregarding unrelated parts.

1. Never make a slice S(V, n) for which variables v of V do not appear in statement fragment n. As an example, suppose we defined a slice on the locks variable at node 27. Defining such slices necessitates tracking the values of all variables at all points in the program.
2. Make slices on one variable. The set V in **slice S(V, n)** can contain several variables, and sometimes such slices are useful. The slice S(V, 26) where **V= { lockSales, stockSales, barrelSales }** contains all the elements of the **slice S({sales}, 27)** except statement 27.
3. Make slices for all A-def nodes. When a variable is computed by an assignment statement, a slice on the variable at that statement will include all du-paths of the variables used in the computation. Slice S({sales}, 36) is a good example of an A-def slice.
4. Make slices for P-use nodes. When a variable is used in a predicate, the slice on that variable at the decision statement shows how the predicate variable got its value. This is very useful in decision-intensive programs like the Triangle program and NextDate.
5. Slices on non-P-use usage nodes are not very interesting. We discussed C-use slices in point 2, where we saw they were very redundant with the A-def slice. Slices on O-use variables can always be expressed as unions of slices on all the A-defs (and I-defs) of the O-use variable. Slices on I-use and O-use variables are useful during debugging, but if they are mandated for all testing, the test effort is dramatically increased.
6. Consider making slices compilable. Nothing in the definition of a slice requires that the set of statements is compilable, but if we make this choice, it means that a set of compiler directive and declarative statements is a subset of every slice.