Introduction to Software Engineering

Structural Testing

Daniel Sinnig, PhD
d_sinnig@cs.concordia.ca

Department for Computer Science
and Software Engineering

29-July-14

Introduction

" Testing is simple – all a tester needs to do is *find a graph and cover it* "

[Beizer, Software Testing Techniques book]

Directed Graphs

- A graph $G = (V, E)$ is a finite (nonempty) set V of nodes and a set E of unordered pairs of nodes, with:
  - $V = \{n_1, n_2, \dots, n_m\}$ and
  - $E = \{e_1, e_2, \dots, e_p\}$ where each (directed) edge $\boldsymbol{e_k = \langle n_i, n_j \rangle}$ **is an ordered pair of nodes** $\boldsymbol{n_i, n_j \in V}$ (with n_i as the initial or start node and n_j as the terminal node)
  - Indegree (indeg) of a node is the number of distinct edges that have the node as terminal node.
  - Outdegree (outdeg) of the node is the number of distinct edges that have the node as start node.

Control flow graph (flow graph, Program graph)

- Given a program written in an imperative programming language, its control flow graph (CFG) is a directed graph in which nodes are statements (or statement fragments) and edges represent flow of control.
- Formally, a CFG is a quadrupel (V, E, s, t) where:
  - $V = \{n_1, n_2, \dots, n_m\}$ and
  - $E = \{e_1, e_2, \dots, e_p\}$ where each (directed) edge $e_k = \langle n_i, n_j \rangle$ is an ordered pair of nodes $n_i, n_j \in V$
  - $s \in V$ is the start node with $indeg(s) = 0$
  - $t \in V$ is the terminal node with $outdeg(t) = 0$
  - Procedure nodes: Nodes $n \in V$ with $outdeg(n) = 1$
  - Decision nodes: Nodes $n \in V$ with $outdeg(n) > 1$

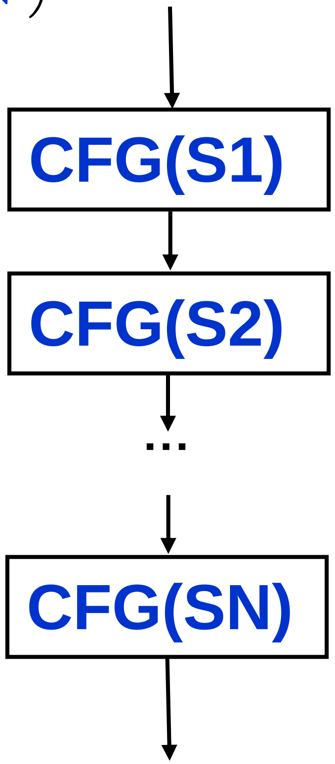
Exercise (5-10min):

- Draw the CFG for the following program

```
Program triangle2
Dim a,b,c As Integer
Dim IsATrinagle As Boolean
Output("Enter 3 integers which are sides of a triangle")
Input(a,b,c)
Output("Side A is", a)
Output("Side B is", b)
Output("Side C is", c)
If (a < b + c) AND (b < a + c) AND (c < a + b)
   Then IsATriangle = True
   Else IsATriangle = False
EndIf
If IsATriangle
   Then  If (a = b) AND (b = c)
           Then Output ("Equilateral")
           Else  If (a≠b) AND (a≠c) AND (b≠c)
                     Then Output ("Scalene")
                     Else Output ("Isosceles")
                 EndIf
           EndIf
  Else  Output("Nota a Triangle")
EndIf
End triangle2
```

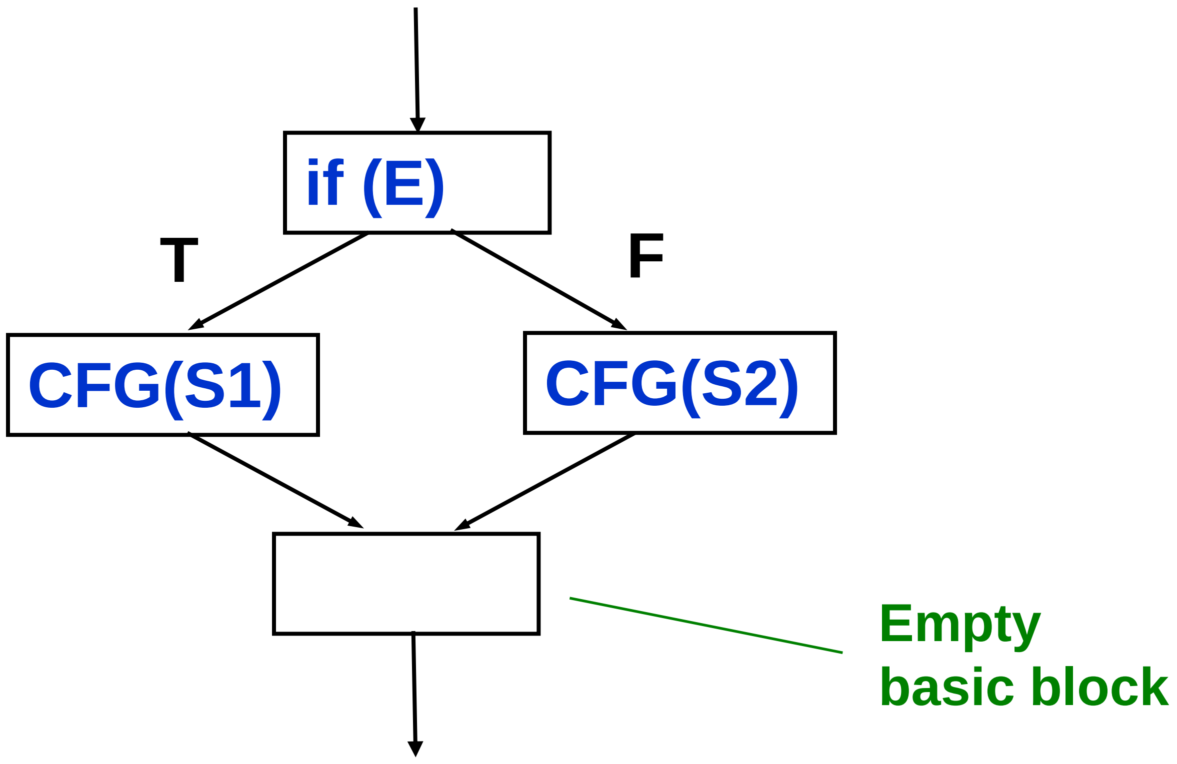
CFG for Block Statement

- CFG(S1; S2; …; SN) =



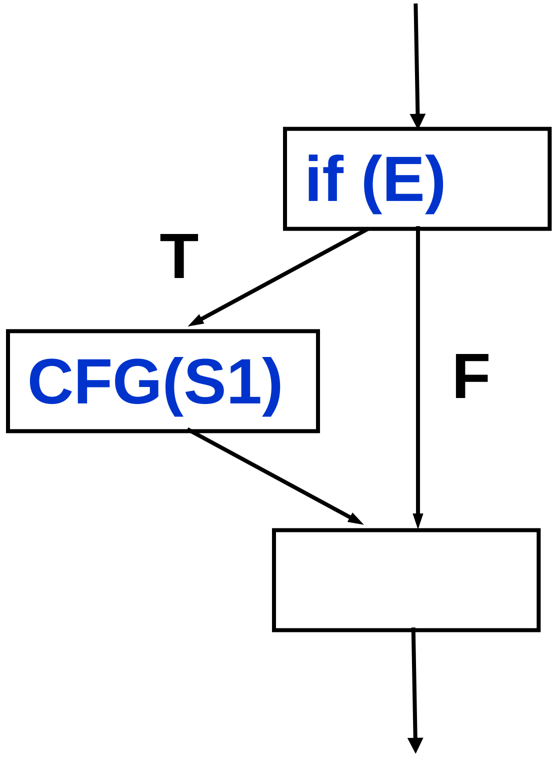
CFG for If-then-else Statement

- CFG (if (E) S1 else S2)



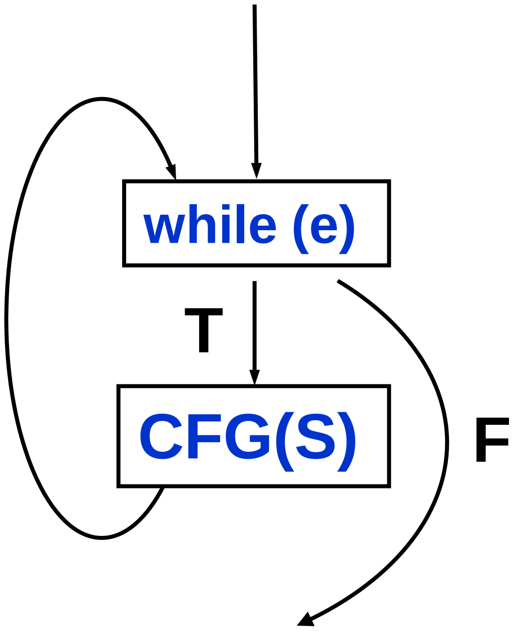
CFG for If-then Statement

- CFG(if (E) S)



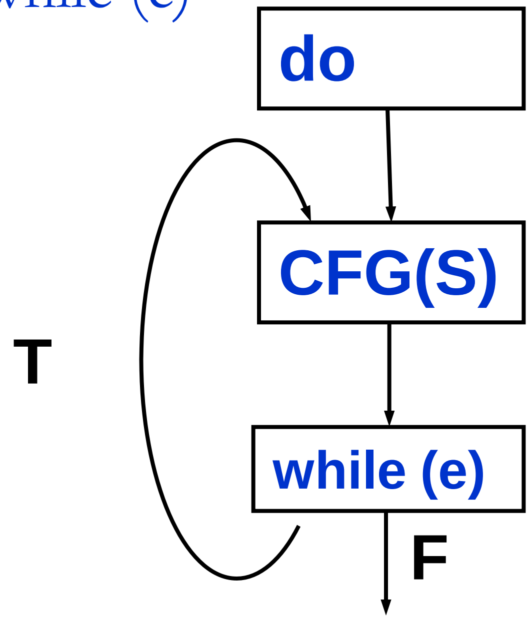
CFG for While Statement

- CFG for: while (e) S



CFG for do while Statement

- CFG for: do S while (e)



Recursive CFG Construction

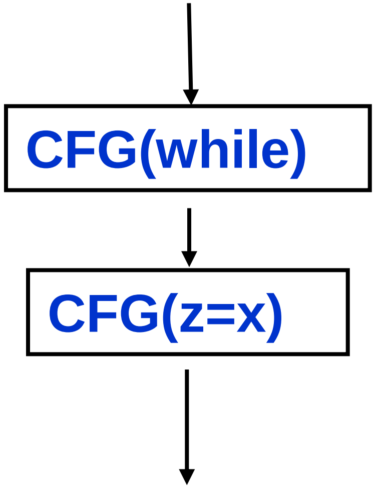
- Nested statements: recursively construct CFG
- Example:

```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```

Recursive CFG Construction

- Nested statements: recursively construct CFG

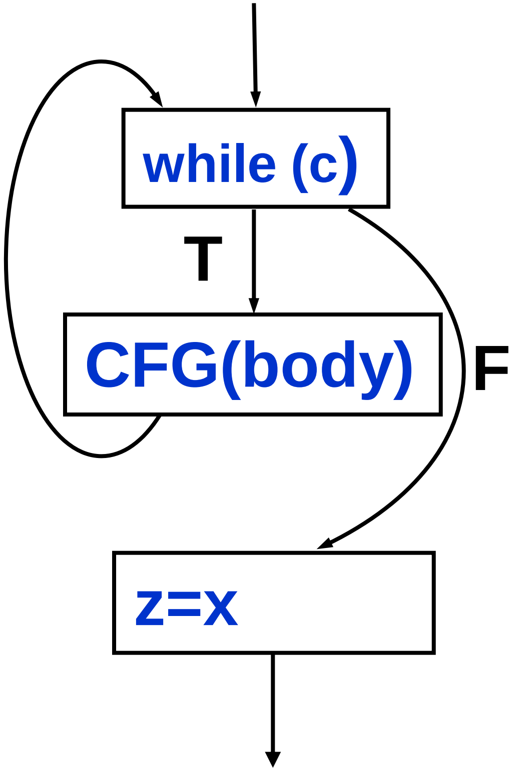
```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```



Recursive CFG Construction

- Nested statements: recursively construct CFG

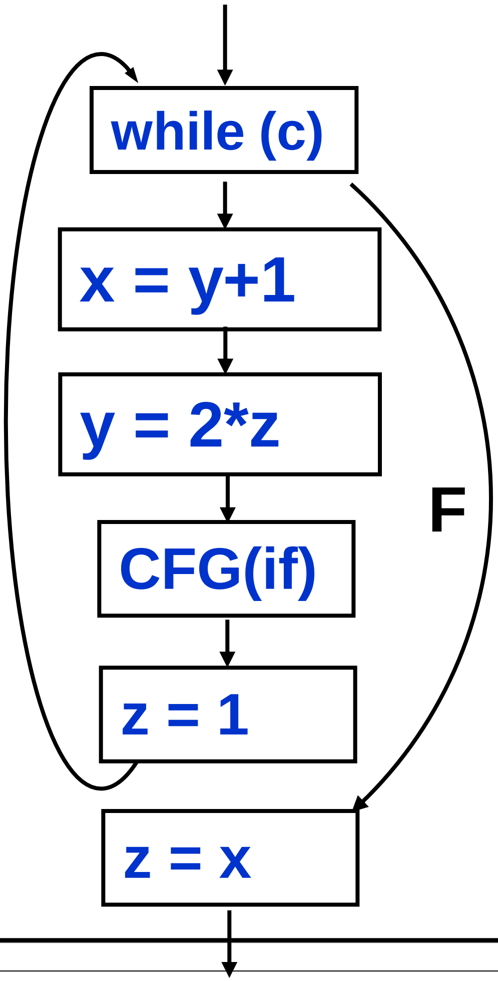
```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```



Recursive CFG Construction

- Nested statements: recursively construct CFG

```
while (c) {
        x = y + 1;
        y = 2 * z;
        if (d)  x = y+z;
        z = 1;
}
z = x;
```



Cyclomatic complexity of CFGs

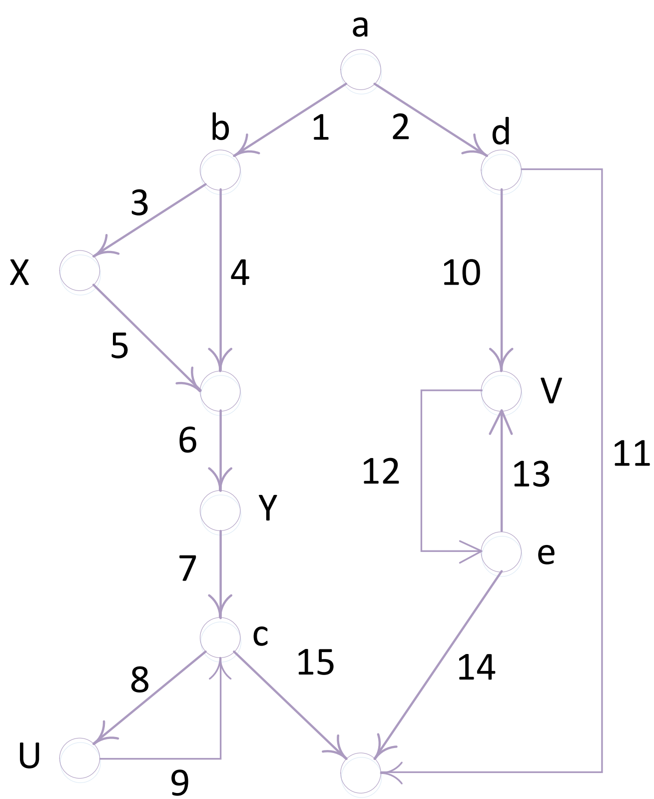
- Number of independent paths for CFGs can be calculated by using the cyclomatic number:
- $V(CFG) = V(V, E, s, t) = e - n + 2$

Coverage Criteria

- **Test coverage criteria:** is a rule or collection of rules that impose test requirements on a test set.
- (Important) white box test criteria:
  - **Statement coverage:** Every statement is executed at least ones (node coverage)
  - **Branch coverage:** Every decision is executed at least ones (edge coverage)
  - **Simple paths coverage:** All simple paths are executed
  - **Visit each loop coverage:** Simple paths coverage + all loops are skipped and executed ones.
  - **All paths coverage:** All paths are executed
  - **Basis path coverage:** All linear independent path are executed
  - **Multi-condition coverage:** All elements of a compound condition are evaluated
  - **Data flow coverage:** Takes into account data definitions and uses

Running Example

```
if (a) {
   if (b) {
     ... //statements X
   }
   ... //statements Y
   while (c) {
     ... //statements U
   }
} else {
   if (d) {
     do {
        ... //statements V
     }
   while (e);
   }
}
```



Statement coverage

- **Strategy:**
  - Every program statement is executed at least once
  - In terms of flowgraph: find a set of paths such that every node lies on at least one path

Statement coverage (cont.)

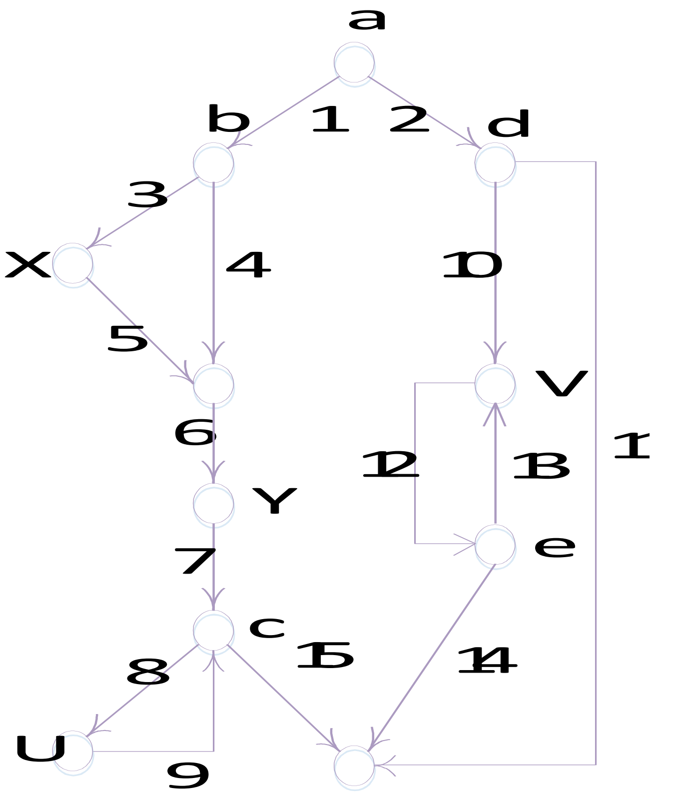
- Select a test set T such that each statement of program P is executed at least once.

```
read(x); read(y);
    if x > 0 then write("1");
    else                 write("2");
    if y > 0 then  write("3");
    else                  write("4");
```

T: {<x = -13, y = 51>, <x = 2, y = -3>}

Statement coverage

```
if (a) {
   if (b) {
     ... //statements X
   }
   ... //statements Y
   while (c) {
     ... //statements U
   }
} else {
   if (d) {
     do {
        ... //statements V
     }
   while (e);
   }
}
```



Minimum number of test cases: 2

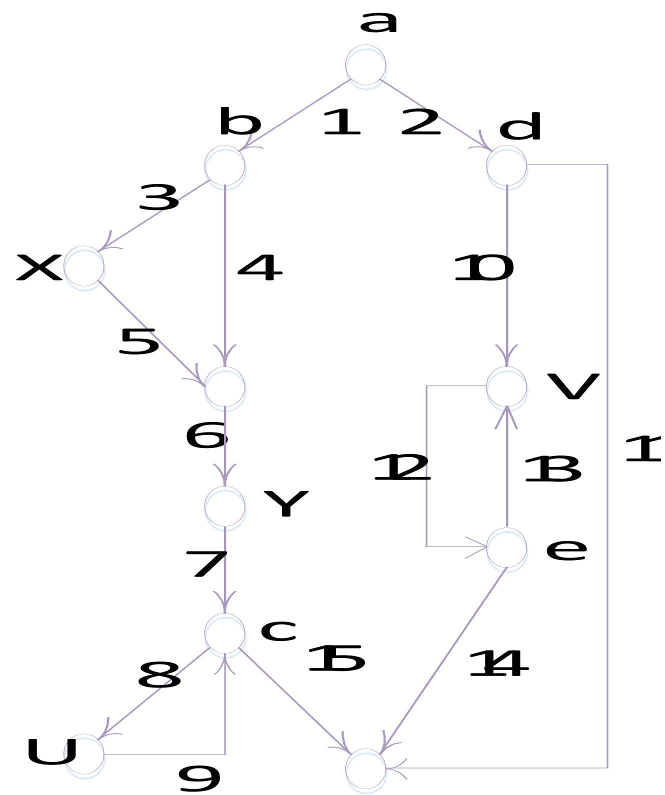
Test cases: {<1,3,5,6,7,8,9,15>, <2,10,12,14>}

Branch coverage

- **Strategy:**
  - Every program branch is executed at least once
  - In terms of flowgraph: find a set of paths such that every edge lies on at least one path

Branch coverage

```
if (a) {
   if (b) {
     ... //statements X
   }
   ... //statements Y
   while (c) {
     ... //statements U
   }
} else {
   if (d) {
     do {
        ... //statements V
     }
   while (e);
   }
}
```



Minimum number of test cases: 4

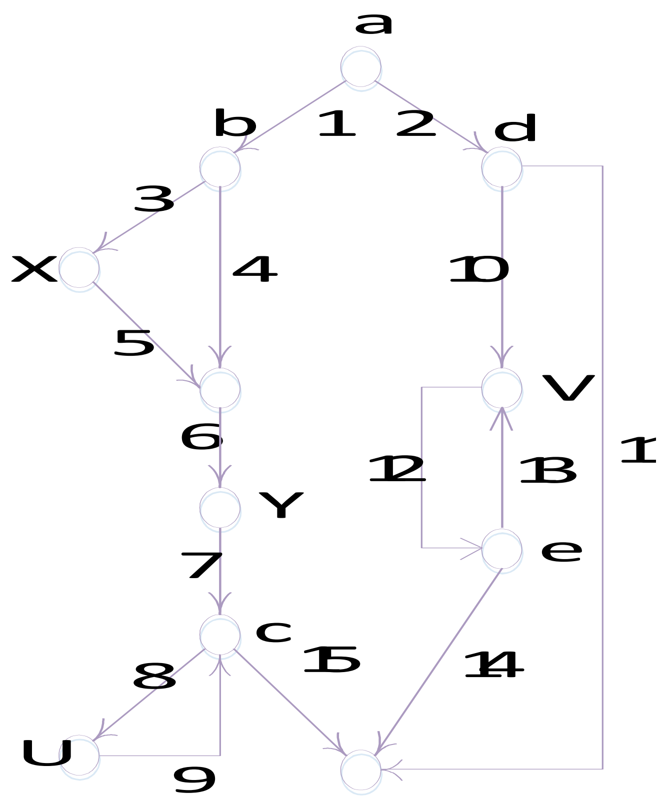
Test cases: {<1,3,5,6,7,8,9,15>, <1,4,6,7,15>, <2,10,12,13,12,14>,<2,11>}

Simple path coverage

- **Strategy:**
  - Every simple path (which does not contain the same edge more than once) is executed once

Simple path coverage

```
if (a) {
   if (b) {
     ... //statements X
   }
   ... //statements Y
   while (c) {
     ... //statements U
   }
} else {
   if (d) {
     do {
        ... //statements V
     }
   while (e);
   }
}
```



Minimum number of test cases: 6

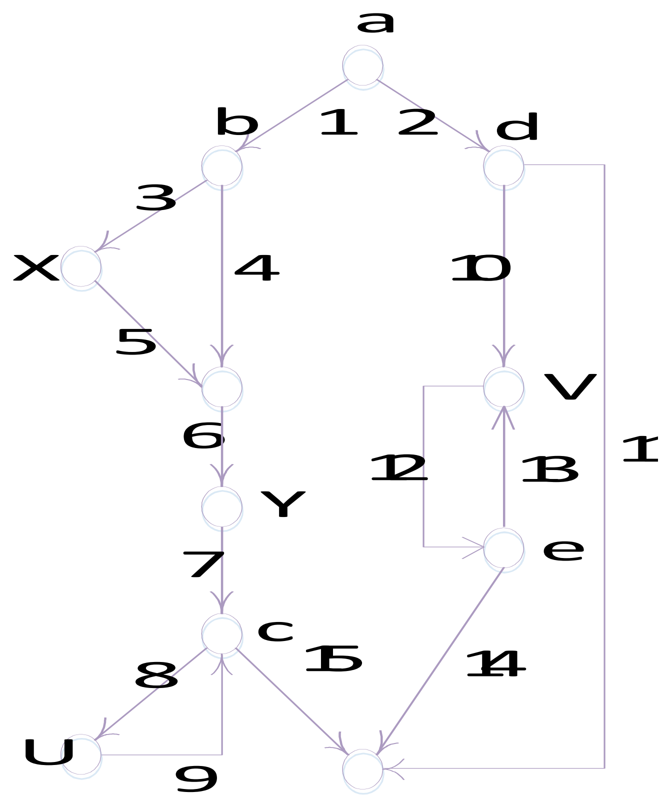
Test cases: {<1,3,5,6,7,8,9,15>, <1,3,5,6,7,15><1,4,6,7,8,9,15>, <1,4,5,6,7,15>
<2,10,12,14>,<2,11>}

Visit each loop coverage

- **Strategy:**
  - Simple path coverage + additional test cases such that for each loop there is one test case that:
    - skips the loop entirely
    - only makes one pass through the loop
  - If code one contains while – do loop then simple path coverage and visit each loop coverage are identical. Why?

Visit each loop coverage

```
if (a) {
   if (b) {
     ... //statements X
   }
   ... //statements Y
   while (c) {
     ... //statements U
   }
} else {
   if (d) {
     do {
        ... //statements V
     }
   while (e);
   }
}
```



Minimum number of test cases: 7

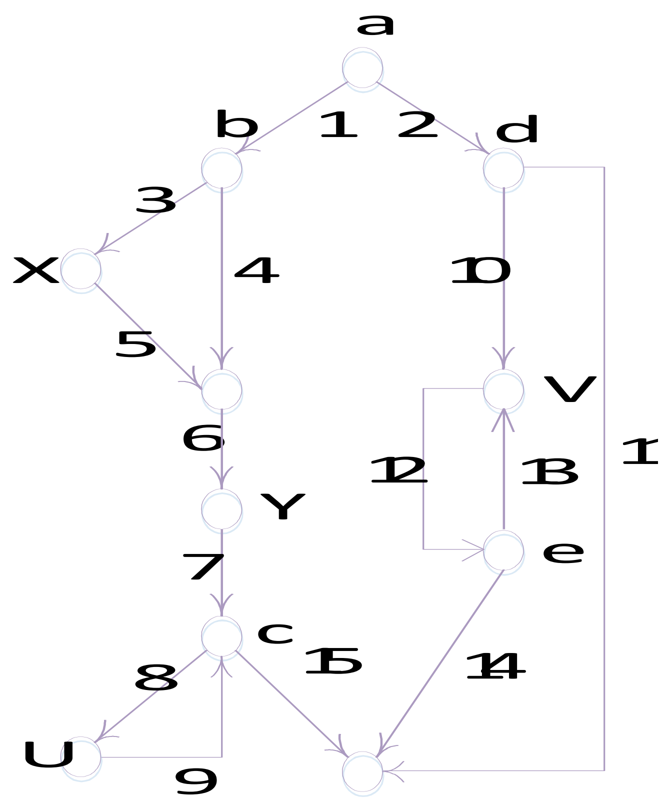
Test cases: {<1,3,5,6,7,8,9,15>, <1,3,5,6,7,15><1,4,6,7,8,9,15>, <1,4,5,6,7,15>
<2,10,12,14>,<2,11>, <**2,10,12,13,12,14**>}

All path coverage

- **Strategy:**
  - Every possible program path is executed at least once
  - In terms of flowgraph: find all paths through the flowgraph
  - Only feasible for if no loops are present

All paths coverage

```
if (a) {
   if (b) {
     ... //statements X
   }
   ... //statements Y
   while (c) {
     ... //statements U
   }
} else {
   if (d) {
     do {
        ... //statements V
     }
   while (e);
   }
}
```



Minimum number of test cases: Infinite

Test cases: {<1,3,5,6,7,15>, <1,4,5,6,7,15>, {<1,3,5,6,7,(8,9)^n,15>, <1,4,6,7,(8,9)^n,15>
<2,11>, <2,10,12,14>, <2,10,12,(13,12)^n,14>} for any n>0

Basis Paths Coverage

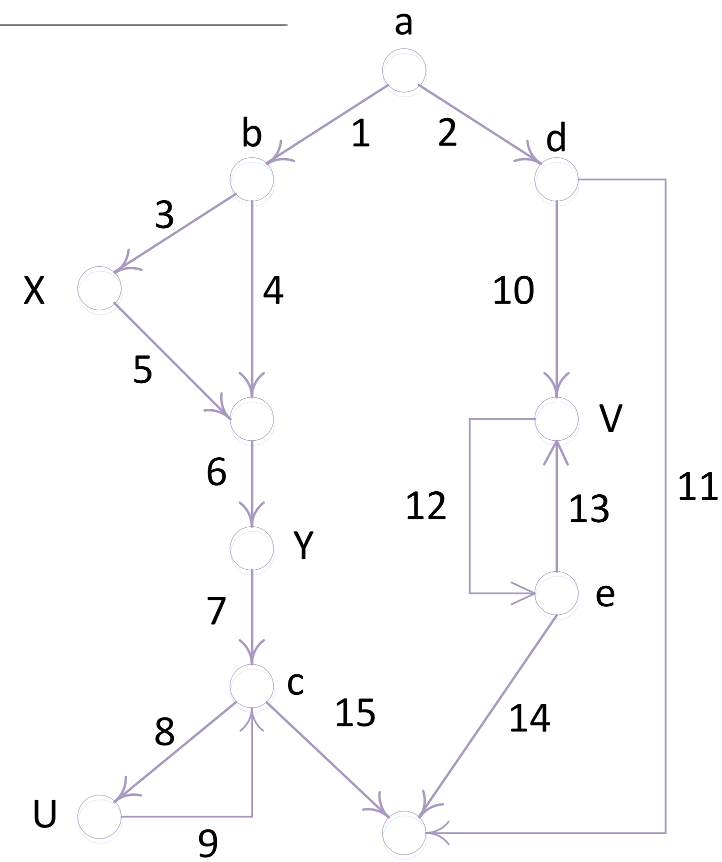
- **Strategy:**
  - every linearly independent path is executed at least once
  - Minimum number of test cases (McCabe Cyclomatic Complexity):

    $V(G) = e – n + 2$

    $V(G)=$ # decision points + 1 (if graph entails binary decision points only)

Basis path coverage

```
if (a) {
   if (b) {
     ... //statements X
   }
   ... //statements Y
   while (c) {
     ... //statements U
   }
} else {
   if (d) {
     do {
        ... //statements V
     }
   while (e);
   }
}
```



Minimum number of test cases: 6

Test cases: {<2,11>,<2,10,12,14>, <2,10,12,13,12,14>, <1,3,5,6,7,15>, <1,3,5,6,7,8,9,15>, <~~1,4,6,7,15~~>, <1,4,6,7,8,9,15> }

<1,4,6,7,15> is not linearly independent

The baseline method

- Developed by McCabe (1987)
- Systematic approach to determine the set of basis paths.
- The method will return a minimal set of basis paths
- However, depending on the choice of the first 'baseline' path, this set may not be unique.
- Mathematical background:
  - A path p is a linear combination of paths $p_1, \ldots, p_n$ iff there are integers $a_1, \ldots, a_n$ such that $p = \sum_{i=1}^{n} a_i p_i$ (in the vector representation)
  - A set of paths is linearly independent iff no path in the set is a linear combination of any other paths in the set.

The baseline method (cont.)

Algorithm:

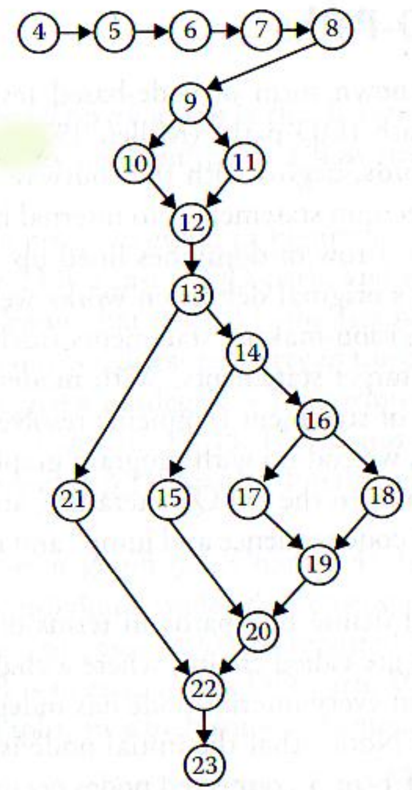
- Step 0: Initialize set of baseline paths B {}
- Step 1: Pick a functional "baseline" path (p1) through the program (a typical run through the program).
- Step 2: Add p1 to B
- Step 3: While there are 'unflipped' (binary) decision nodes do
  - Step 3.1: Pick path p from B
  - Step 3.2: Generate the next baseline path p_{next} by "flipping" the first decision node (n_d) of p. Should p_{next} rejoin p, it must follow it until the end.
  - Step 3.3: Add p_{next} to the set of basis paths. B
  - Step 3.4: Mark n_d as flipped

The baseline method (cont.)

- **Remarks**
  - Multi-way decisions (e.g., switch nodes) must be "flipped" to each of their decision outcomes
  - If the CFG only contains binary decision then the minimal number of basis paths can also be calculated as: Number of decision nodes + 1
  - Criticism:
    - May return infeasible paths due to data dependencies which conflict with the independency assumption of basis paths

Exercise (5-10min): Determine the set of basis paths for the following CFG. Are all paths feasible?

```
1  Program triangle2
2  Dim a,b,c As Integer
3  Dim IsATrinagle As Boolean
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)
9  If (a < b + c) AND (b < a + c) AND (c < a + b)
10   Then IsATriangle = True
11   Else IsATriangle = False
12 EndIf
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("Nota a Triangle")
22 EndIf
23 End triangle2
```



Path Testing Process

- Input:
  - Source code and a path selection criterion
- Process:
  - Generation of a CFG
  - Selection of Paths
  - Generation of Test Input Data
  - Feasibility Test of a Path
  - Evaluation of Program's Output for the Selected Test Cases

White Box Testing Advantages

- Structural testing methods are very amenable to:
  - Rigorous definitions
    - control flow, objectives, coverage criteria, relation to programming language semantics
  - Mathematical analysis
    - Graphs, path analysis
  - Precise measurement
    - Metrics, coverage analysis

Problems with White-Box Testing

- Infeasible paths:
  program paths that cannot be executed for any input
- No white-box strategy on its own can guarantee adequate software testing
- Knowing the set of paths that satisfies a particular strategy doesn't tell you how to create test cases to match the paths.