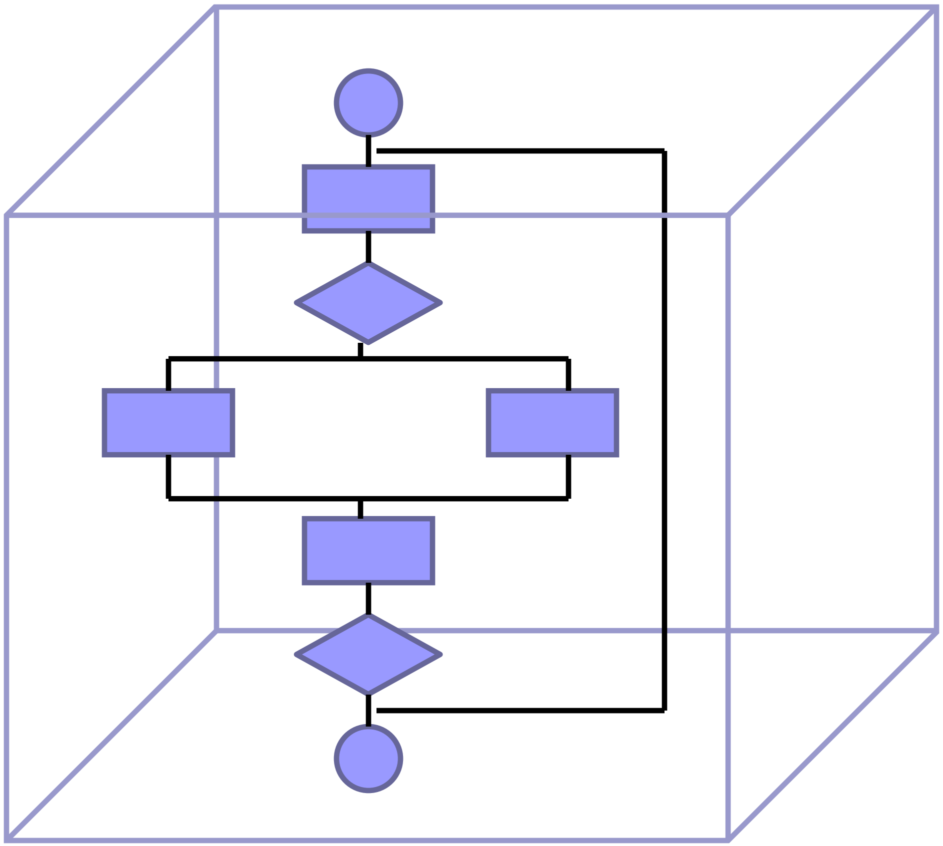
White Box Testing (I)

Chapter 7

Introduction

- Black box testing → you know the functionality
  - From the *outside*, you are testing its functionality against the specifications
- White box testing → you know the code
  - Given knowledge of the internal workings, you thoroughly test what is happening inside

White box testing



Also called structural/clear box /glass box testing

Impossible to exercise all paths…
CANNOT BE EXHAUSTIVE!!!

White Box Testing

- It is a testing technique that examines the implementation details of the code as part of the coding phase itself
- Examining the logic and structure of code
- Employed by unit testing

Why bother with white box testing?

- Black box testing:
  - Requirements fulfilled and
  - Interfaces available and working
- Question: Why white box testing?

Logical errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed

We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis

Typographical errors are random

White Box vs. Black Box Testing

Black Box Testing

Test Focus: Requirements

Validation Audience: User

Code Coverage: Supplemental

Testing things in your specs that you already know about

White Box Testing

Test Focus: Implementation

Validation Audience: Code

Code Coverage: Primary

Testing things that may not be in your specs but in the implementation

Main categories of white box testing

Control flow coverage testing

Data flow coverage testing

Control flow/Coverage testing

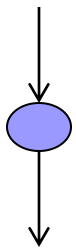
- Control flow of a program is represented by a control flow graph
- Various aspects of the flow graph provide insights for developing an adequate set of test cases
- Coverage
  - A metric for measuring the adequacy and completeness of the test cases
  - For each type of coverage, 100% coverage is the goal

Control flow graph (CFG)

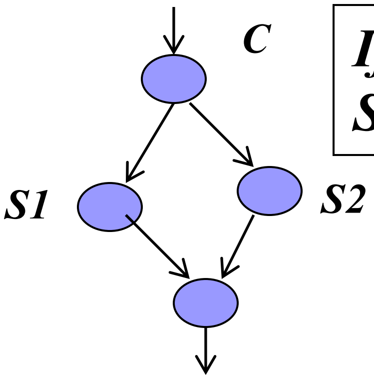
Definition:

- graph representation of control flow of the program
- shows order of executions of statements in the program
  - depends on the semantics of the programming language
- nodes represent statements
- directed edges connect two nodes *a* and *b* iff (if and only if) *b* can be executed right after *a* during the program execution
- edges are labeled with truth values (T or F), which reflect the Boolean decision made at node "a"

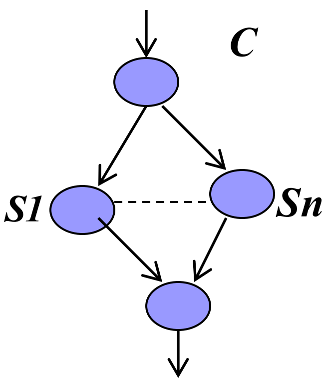
Control Flow Graphs –constructs(1)



Sequential statement block

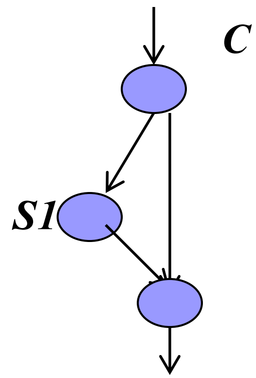


If C then S1 else S2;



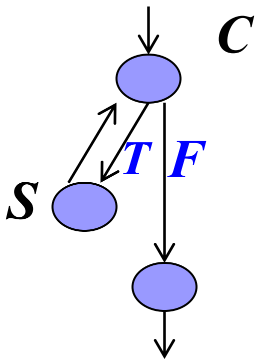
Case C of

L1: S1;
L2: S2;
…
Ln: Sn;
end;

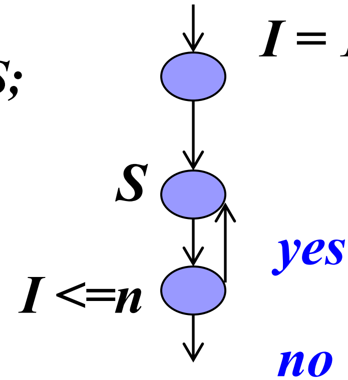


If C then S1;

Control Flow Graphs –constructs(2)

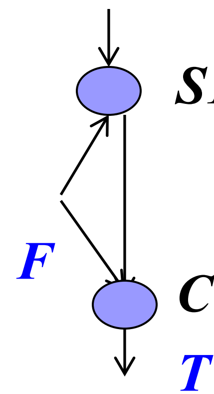


While C do S;



For loop:

for I = 1 to n do S;

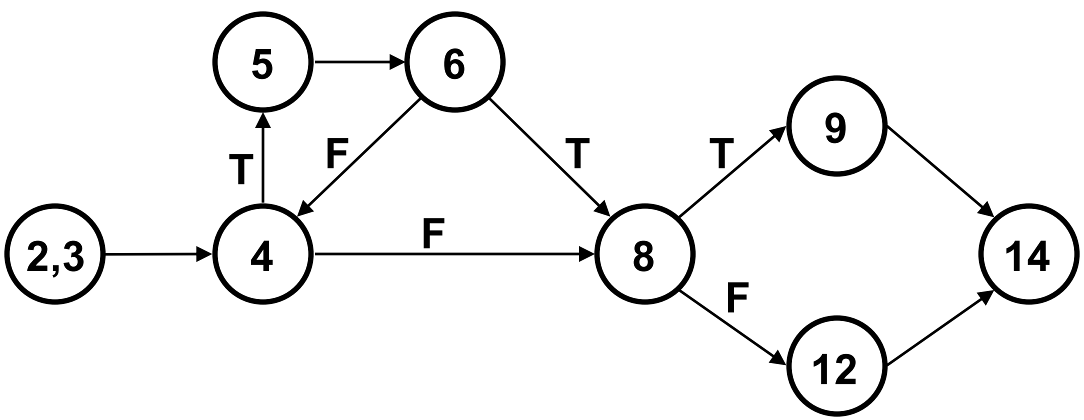


Do loop:

do S1 until C;

Control flow graph: example

statements 1,7,10,11,13, and 15 are ignored

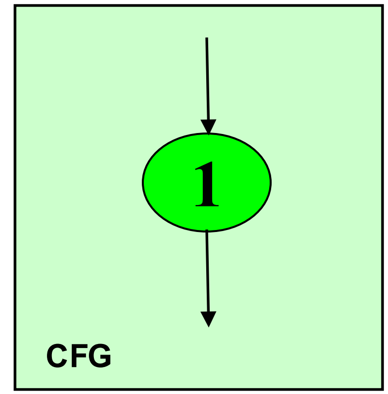



```
1     int Program() {
2         int x,y;
3         cin >> x >> y;
4         while (x>y) {
5             x = x-10;
6             if (x<=0) break;
7             }
8         if (x>=0 && y>0) {
9             y=y-x;
10            }
11          else {
12            y=0;
13            }
14        return y;
15      }
```

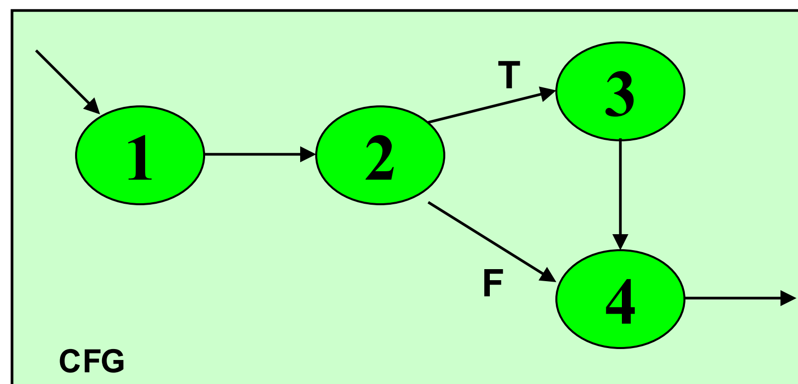
Examples

```
Statement1;
Statement2;
Statement3;
Statement4;
```

can be represented as **one** node as there is no branch.



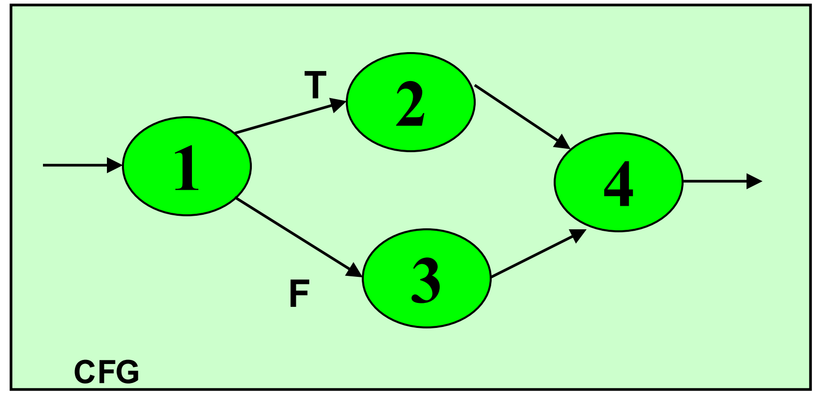

```
Statement1;      1
Statement2;

if X < 10 then   2
   Statement3;   3

Statement4;      4
```



Examples

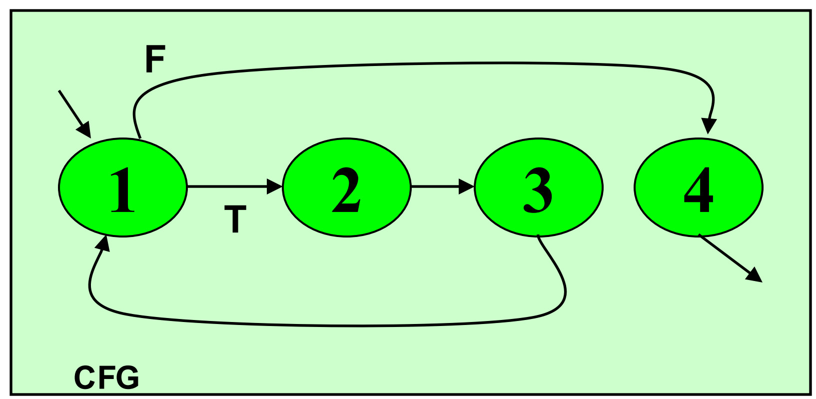
```
if X > 0 then
    Statement1;
else
    Statement2;
```

1
2
3




```
while X < 10 {
    Statement1;
    X++; }
```

1
2
3



a node **4** in both CFGs?

CFG Notation

- A CFG should have:
  - 1 entry edge (directed edge).
  - 1 exit edge.
- All nodes should have:
  - At least 1 entry edge.
  - At least 1 exit edge.
- **A logic node** that does not represent any actual statements can be added as a joining point for several incoming edges.
  - Represents a logical closure.
  - Example:
    - Node 4 in the `if-then-else` example from previous slide.

Example

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

{<x = 2, y = 3>, <x = - 13, y = 51>, <x = 97, y = 17>, <x = - 1, y = - 1>} cover all statements

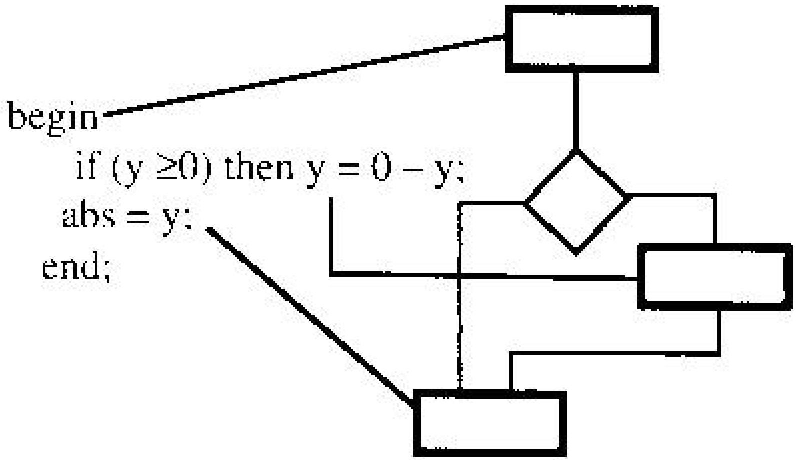
{<x = - 13, y = 51>, <x = 2, y = - 3>} is minimal

Statement coverage criterion

Select a test set T such that every elementary statement in P is executed at least once by some input datum d in T

- an input datum executes many statements → try to minimize the number of test cases still preserving the desired coverage

Statement coverage



Statement coverage

Every <u>statement</u> in the code has to be executed at least once

```
begin
if (y >=0) then y=0-y;
abs =y;
end;
```

Test case y =0 - all statements executed

Statement coverage: A coverage tool

Standard coverage Unix tool tcov (Python coverage.py)

Compile program under test with a special option

Run a number of test cases

A listing indicates how often each statement was executed

and percentage of statements executed

Branch (decision) coverage

Every <u>branch</u> (edge of the graph) in the code has to be executed at least once

This requires that each decision box is evaluated to *true* and *false* at least once

We strive for 100% coverage

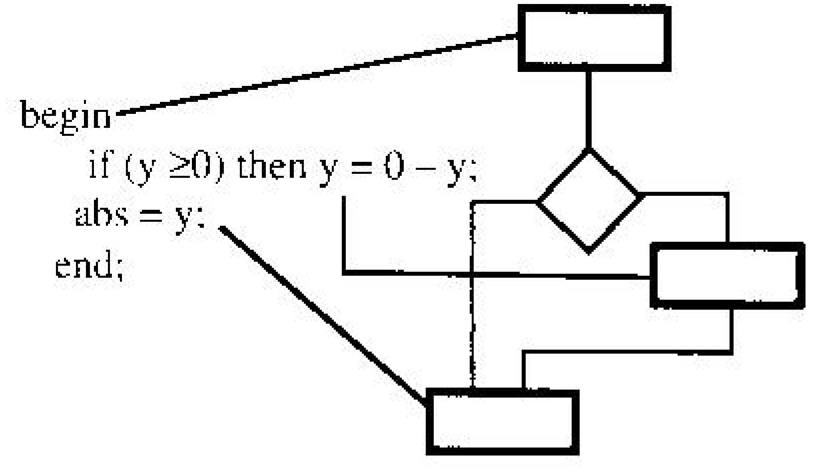
Branch (decision) coverage

Every <u>branch</u> in the code has to be executed at least once

<u>Back to the previous example</u>

Two test cases: y=0 and y=-5

Fault detected



Branch coverage- example (1)

If((x <level_2) && (y > level_1))
{z = compute(x, y); else z = compute_altern(x,y);
}

Consider that we assumed the values **level_2 =- 5** **level_1 =11**

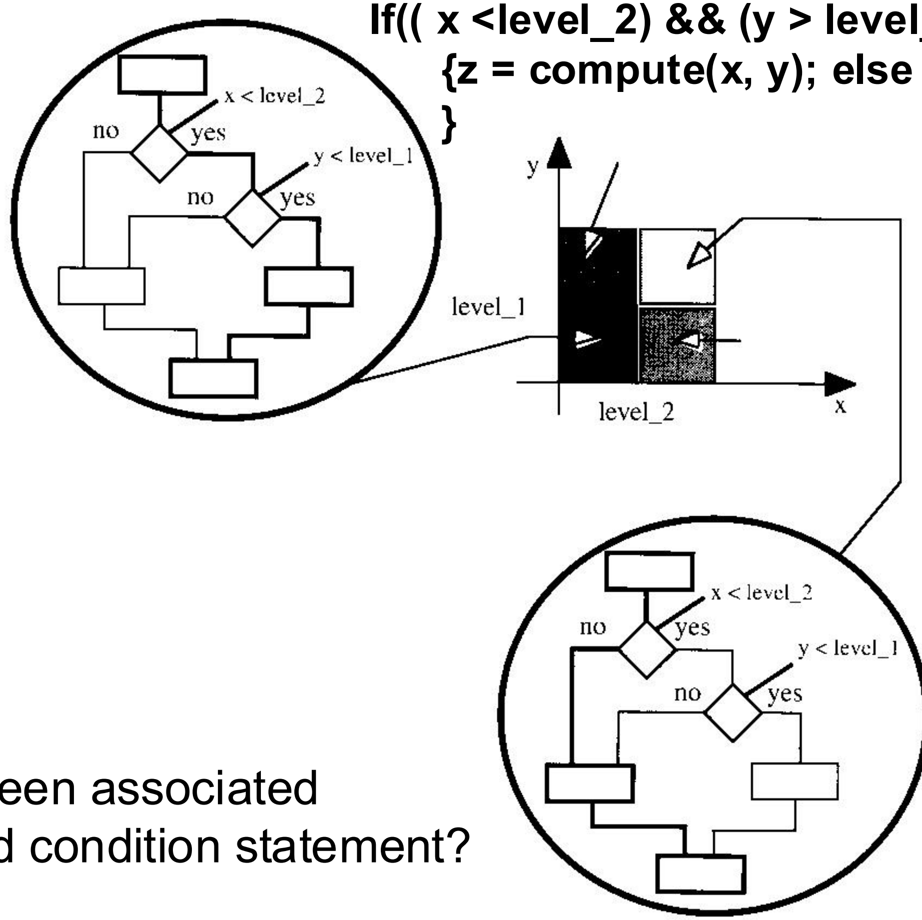
Test cases:

x =-4, y =10

x =-6, y =12

Branch coverage- example (2)

```
If(( x <level_2) && (y > level_1) )
    {z = compute(x, y); else z = compute_altern(x,y);
    }
```



Level_2=-5
Level_1=11

x=-4, y=10
x=-6, y=12

What if fault has been associated
with the compound condition statement?

Figure 12.20 Testing cases for complete branch conditions

True-false test case

If((x <level_2) && (y > level_1))
{z = compute(x, y); else z = compute_altern(x,y);
}

Level_2=-5
Level_1=11

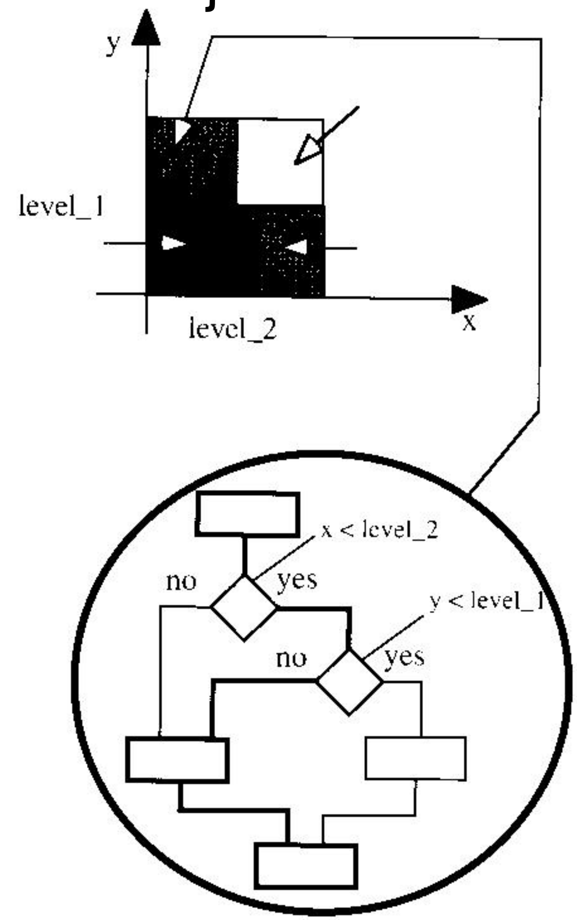


Figure 12.21 True–false test case

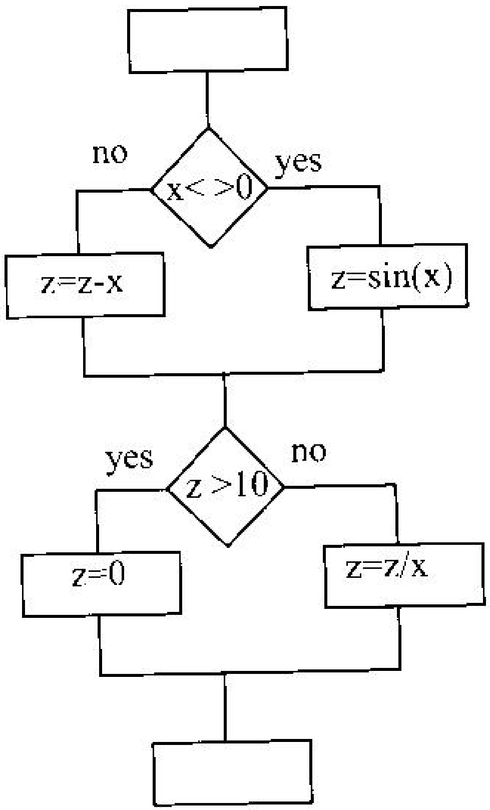
Condition/branch coverage

Every branch in the code has to be executed at least once and all possible combinations of conditions in compound decisions must be exercised.

Could be quite challenging- if the number of subconditions is high; n-subconditions gives rise to 2^n test cases

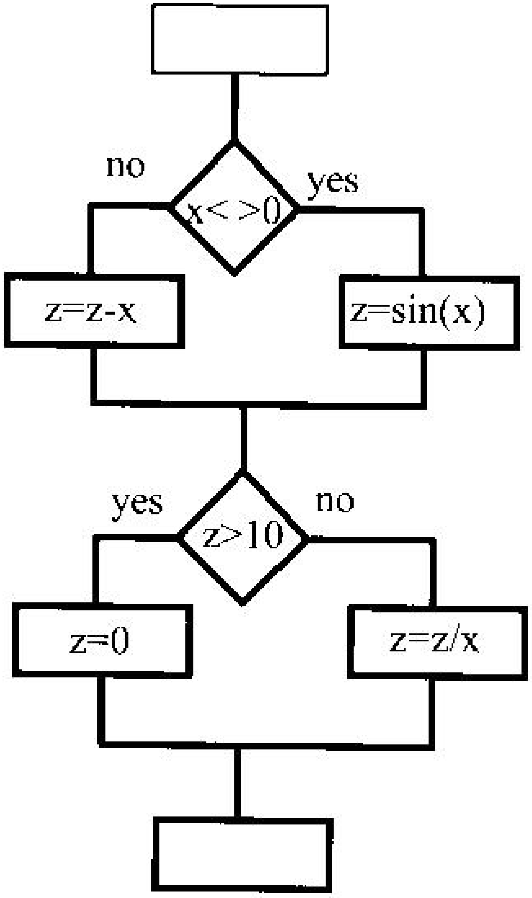
(real-time systems/software)

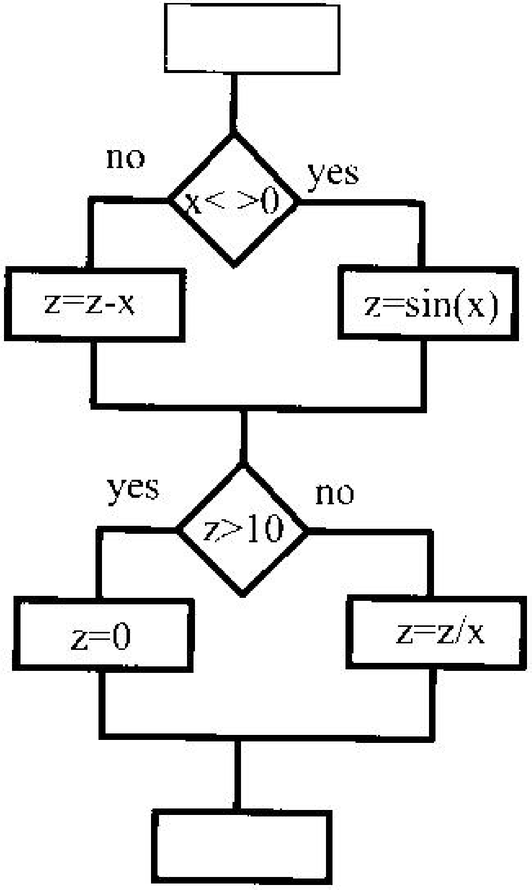
Branch coverage: Example 2



Branch coverage test cases

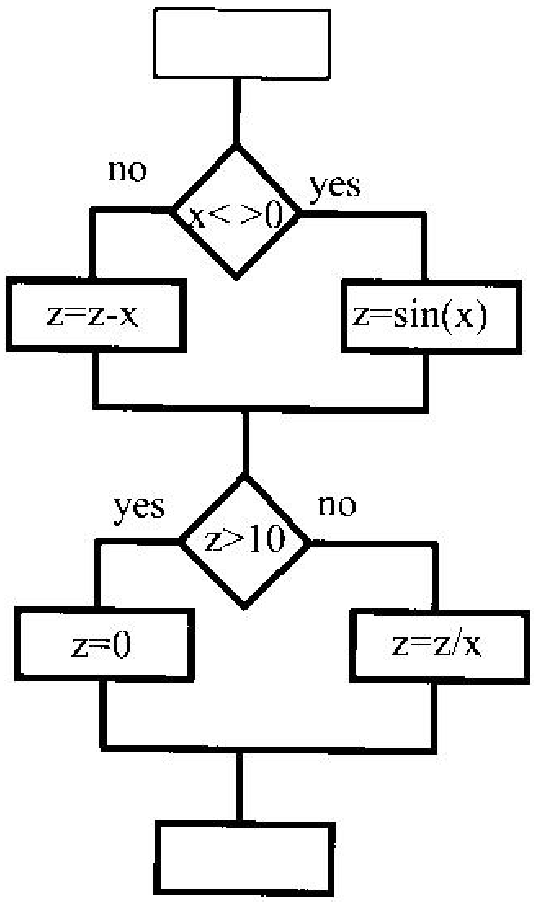
Test cases: { x=2, z =6} {x=0, z= 12}

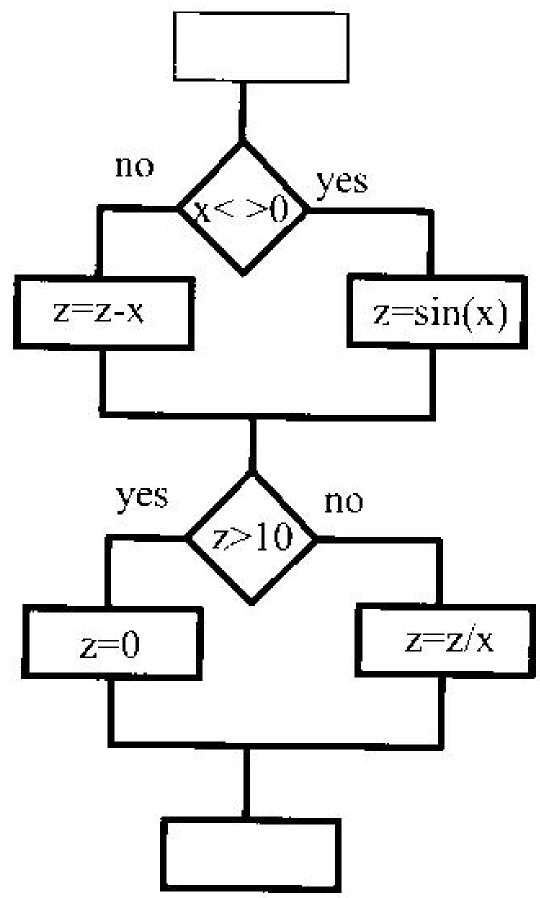




Branch coverage test cases

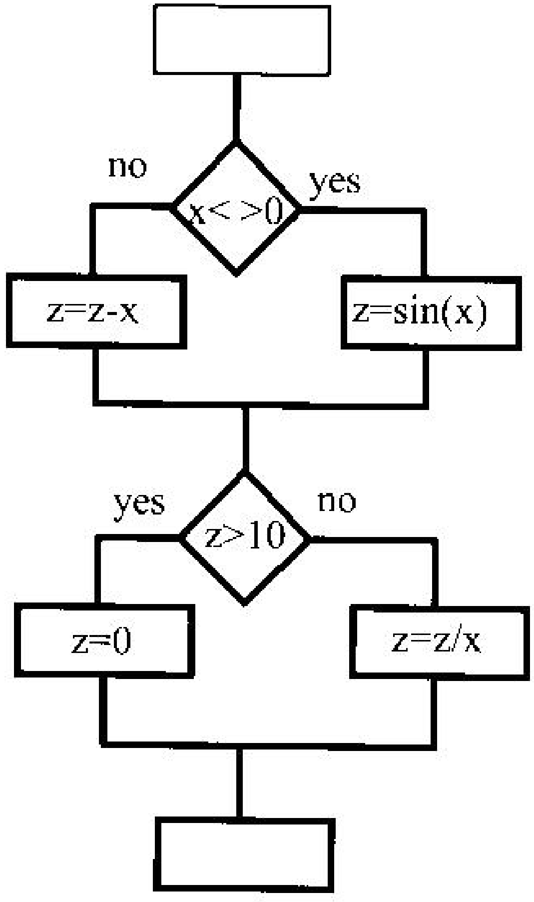
What if at some point x assumes zero → z/x produces a failure; additional test cases?

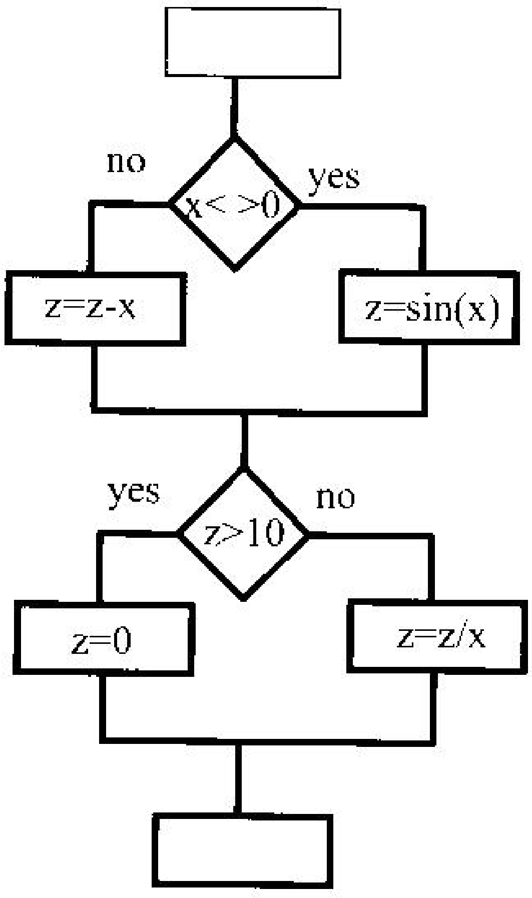




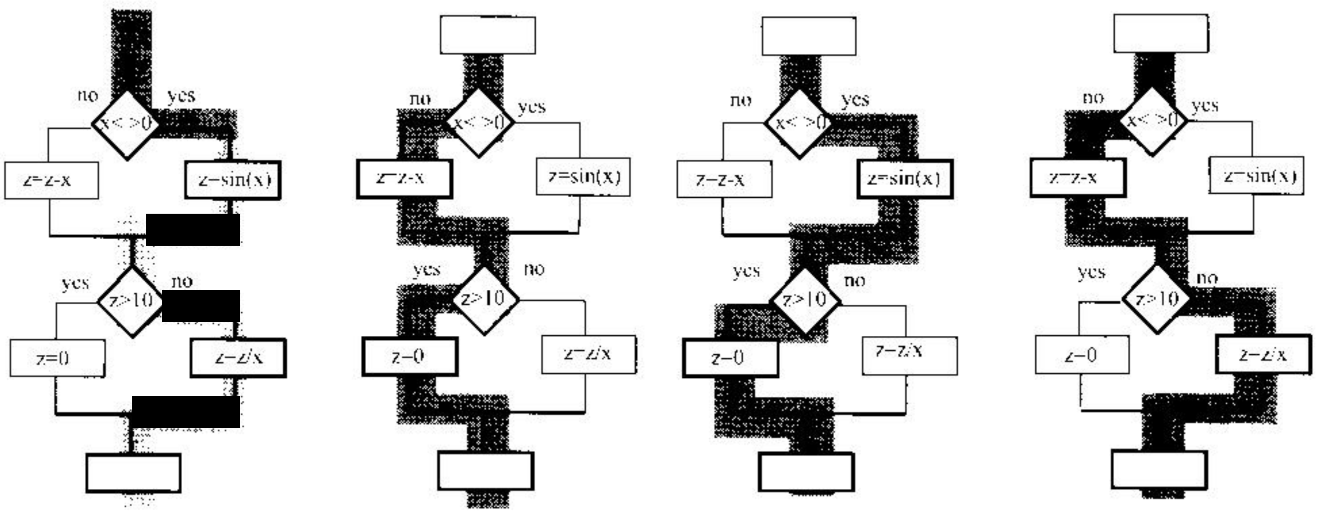
Branch coverage test cases

{x=1, z =5}   {x=2, z =15}   {x=0, z =7}   {x=0, z =13}





Path traversal

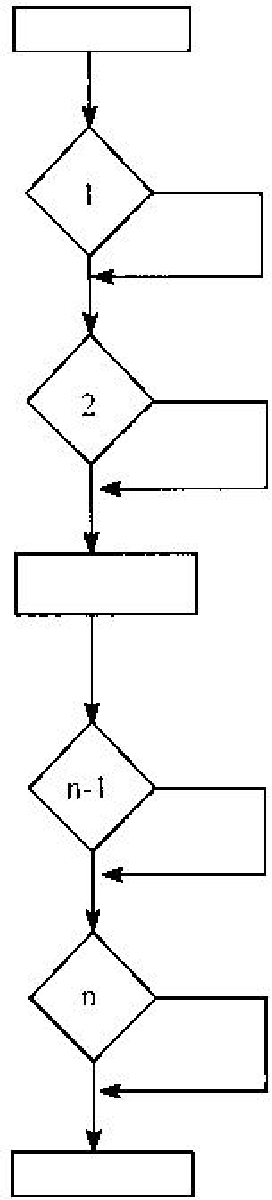


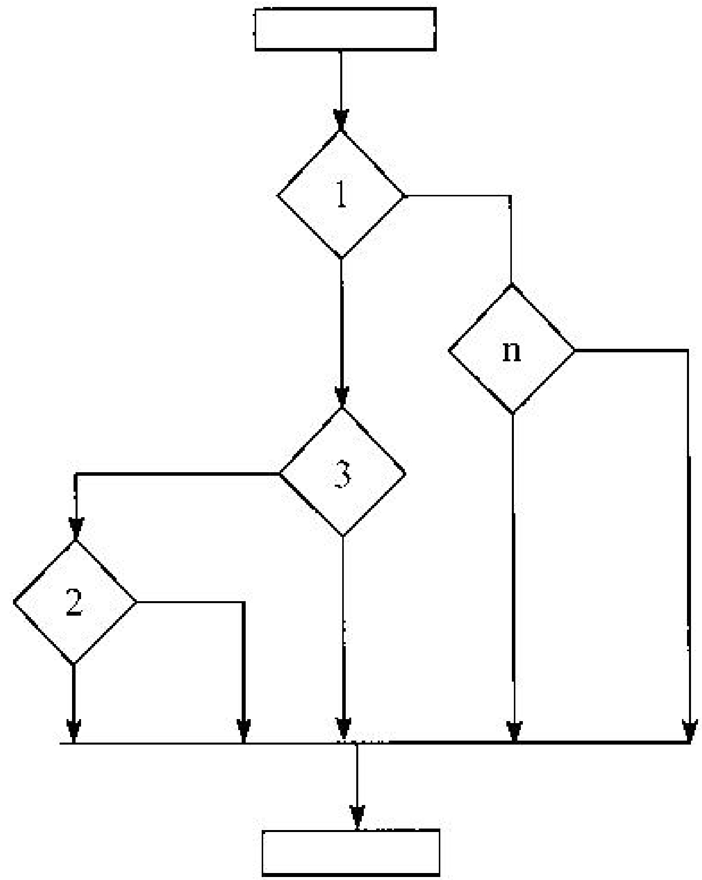
Path coverage

All possible logical paths must be exercised

Usually a high number
Could we achieve 100% coverage?

Number of paths





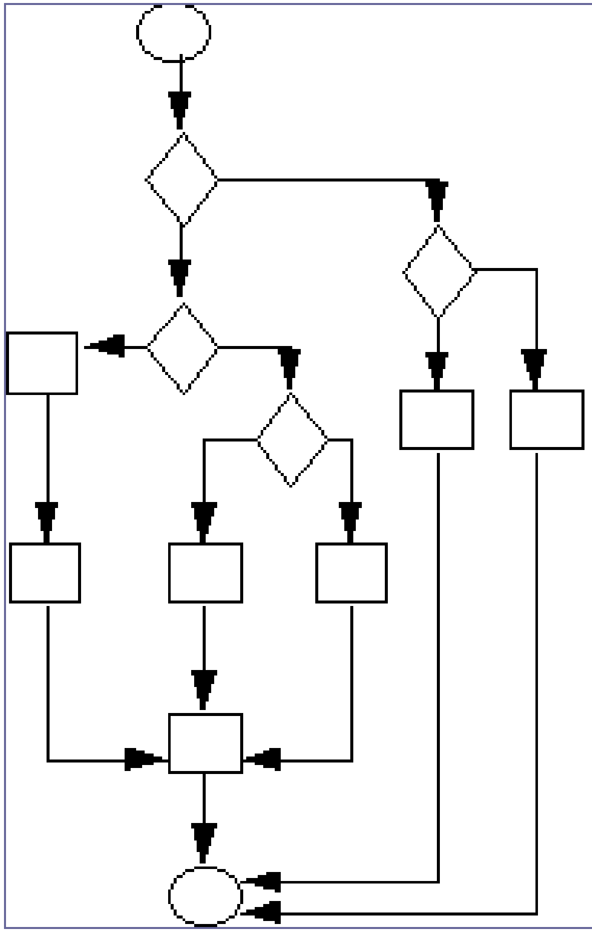
Branch-no merge: n+1 paths

Branch-merge: 2^n paths

Path coverage: bounds on the number of paths

$n+1 \quad <= \text{number of paths} <= 2^n$

McCabe complexity measure



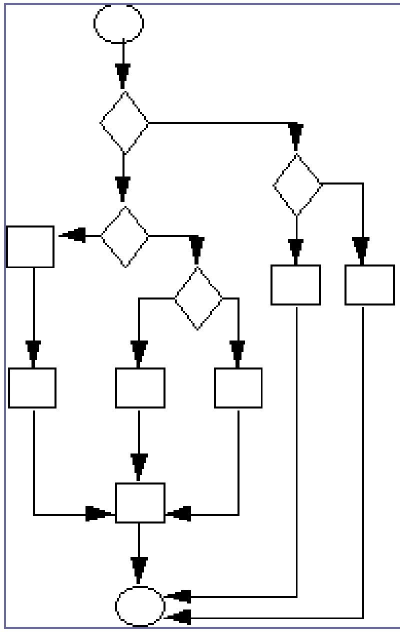
If G is the control flowgraph of program P and G has e edges (arcs) and n nodes

$$\boxed{v(P) = e - n + 2}$$

v (P) is the number of linearly independent paths in G

here e = 16 n = 13 v (P) = 5

McCabe complexity measure



If G is the control flowgraph of program P and G has e edges (arcs) and n nodes

$$\boxed{v(P) = e - n + 2}$$

v (P) is the number of linearly independent paths in G

here e = 16 n =13 v (P) = 5

Linearly independent path is one that contains at least one new (viz. previously unvisited) node and starts and finishes on the same node

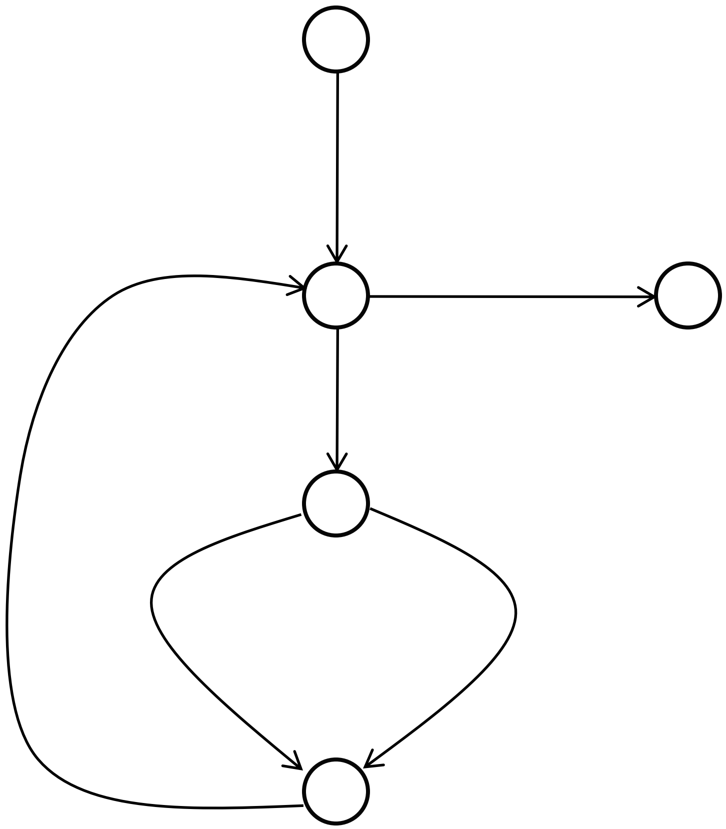
Strongly connected graphs

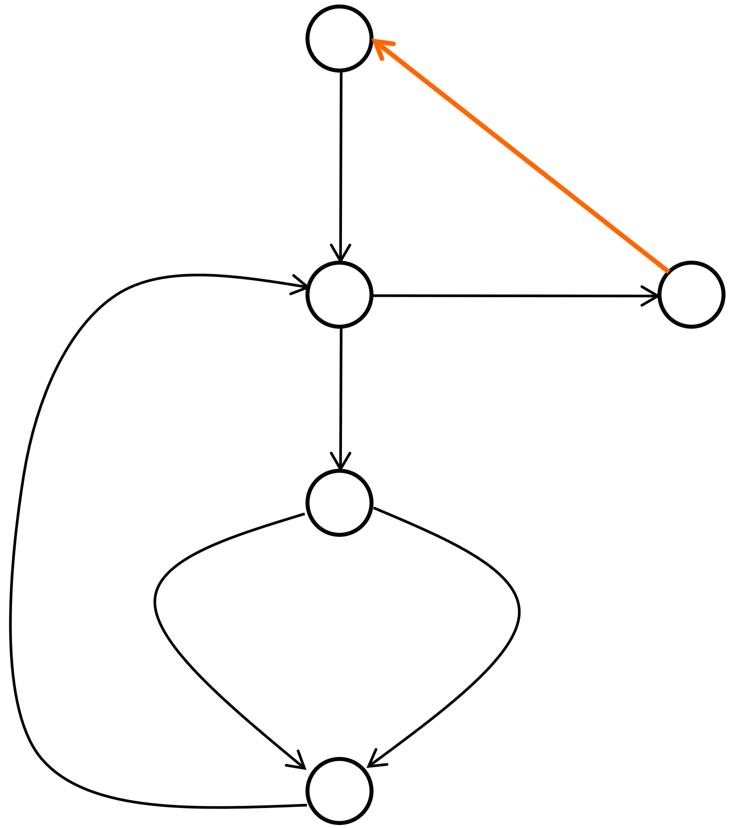
A graph is said to be ***strongly connected*** if there is a path from any node in the graph to any other node. In this case

$$v(P)= e-n+1.$$

In general, the graph is not strongly connected but can be made by adding an edge.

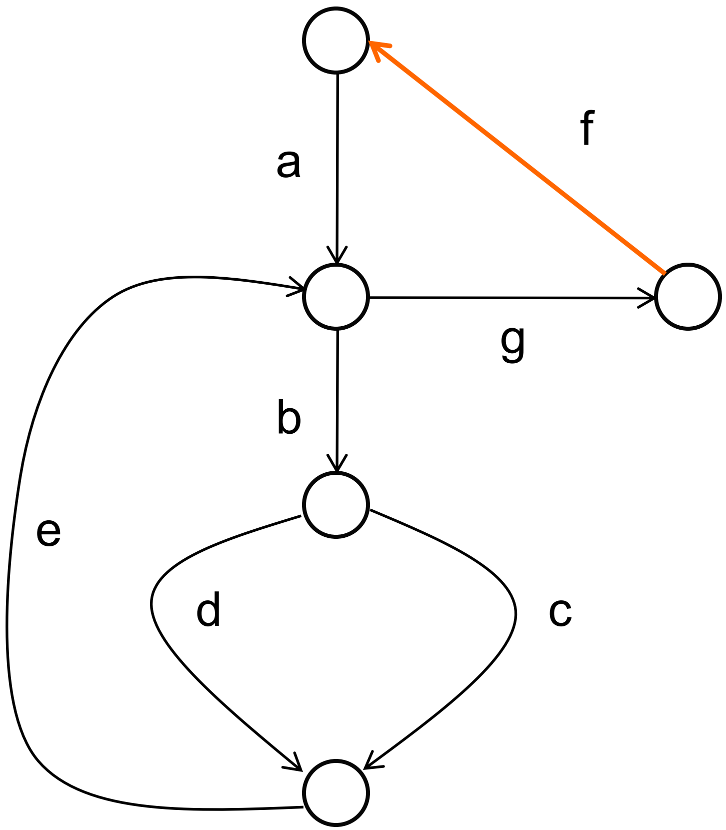
Strongly connected graphs





Strongly connected graph – added edge

Vector representation of paths



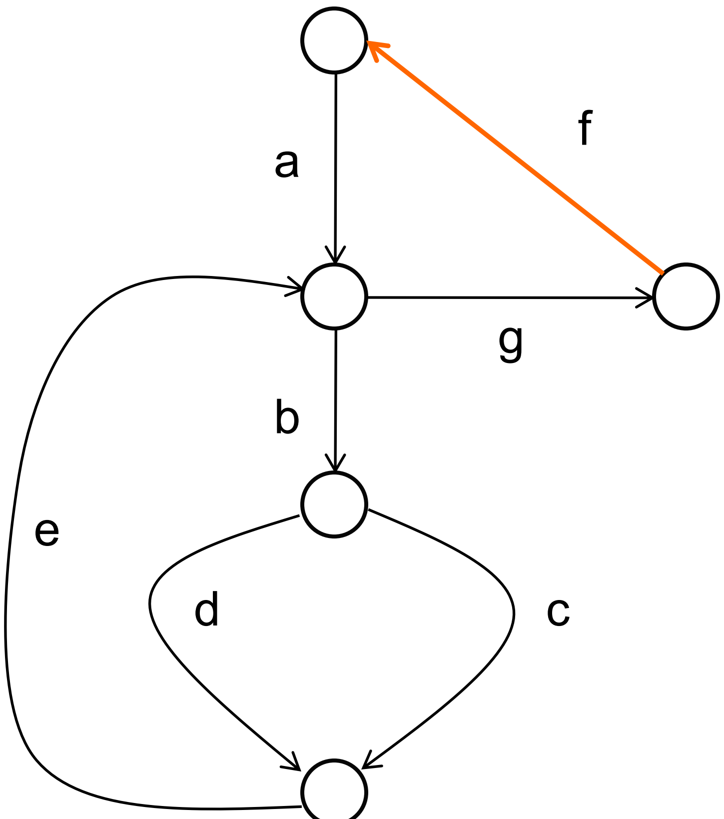
With each path associated is a vector of integers representing the number of times the given edge is used in the path- vector representation of paths

Arrange the edges as a b c d e f g

Path **abdeg** given as [1 1 0 1 1 0 1]

Path **bcebce** is given as [0 2 2 0 2 0 0]

Vector representation of paths



A path is said to be a linear combination of other paths is equal to that formed by a *linear combination* of their vector representation

bdeg is a linear combination of

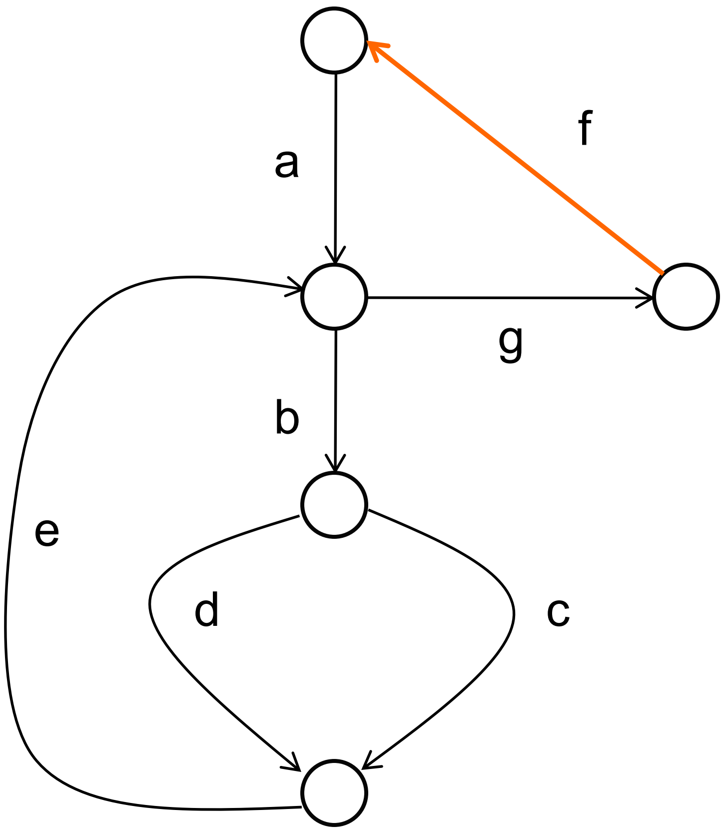
bd and **eg** because

[0101101] = [0101000] +[0000101]

ag is a linear combination of **abceg** and **bce** :

[1000001]= [1110101]-[0110100]

Vector representation of paths



A set of paths is ***linearly independent*** if no path in the set is a linear combination of any other paths in the set

linearly dependent paths:

abdeg [1101101]
abdebdeg [1202201]
ag [1000001]

abdebdeg= abdeg +abdeg –ag

[2202202] –[1000001] = [1202201]

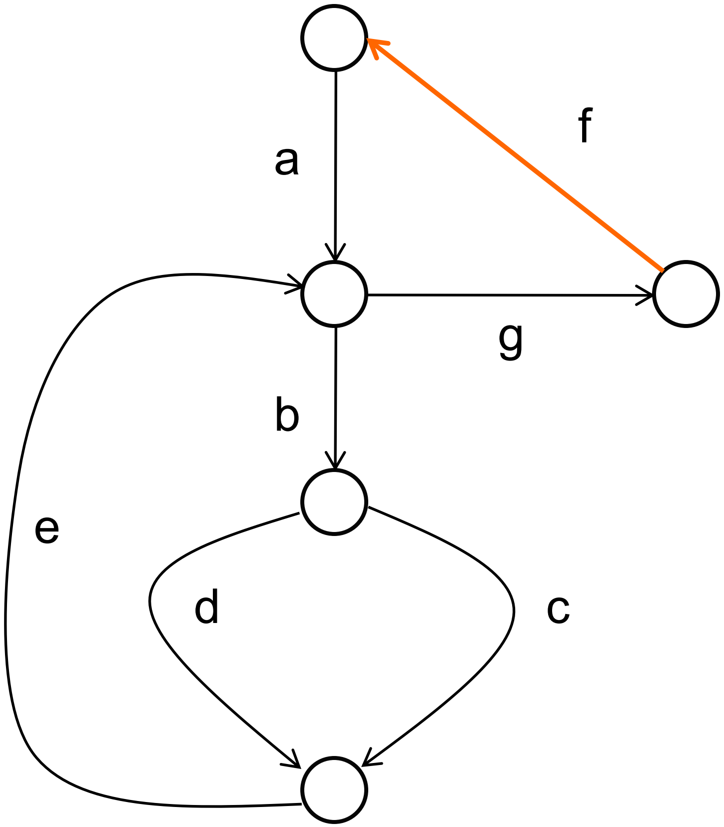
ag=abdeg +abdeg –abdebdeg

[1101101]+[1101101]-[1202201]=
[2202202] -[1202201]= [1000001]

abdeg =1/2 abdebdeg +1/2ag

[1/2 10110 1/2]+[1/2 00000 1/2]= [1101101]

Vector representation of paths



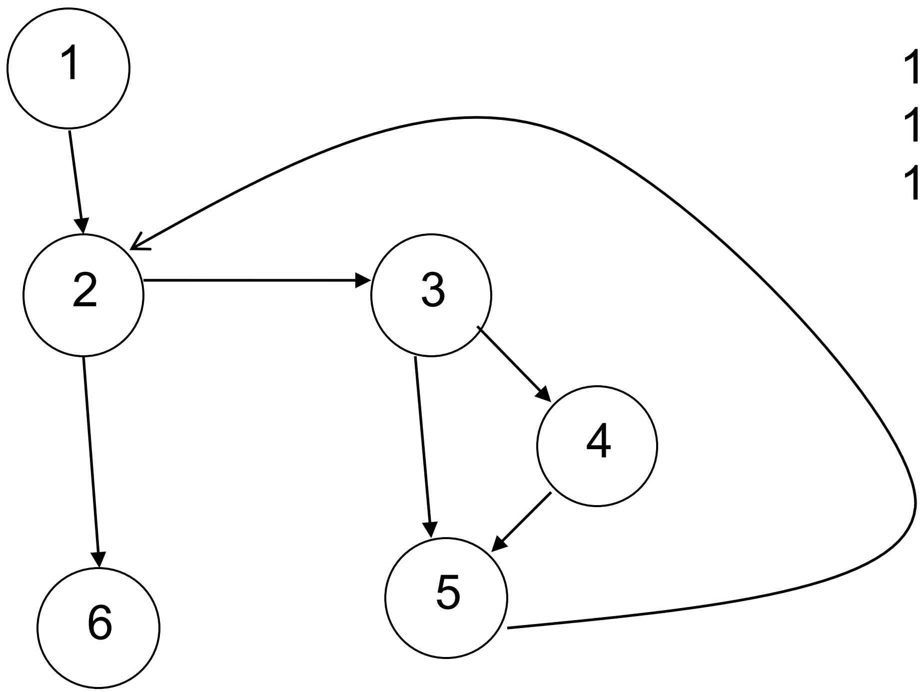
A set of paths is ***linearly independent***
if no path in the set is a linear combination
of any other paths in the set

Linearly independent paths
ag
abceg
abdeg

McCabe complexity measure

Linearly independent path

Any path that introduces at least one new node- must traverse any edge not previously covered



1-2-6
1 - 2 - 3 -5 -2 - 6
1 – 2 – 3 - 4 – 5 – 2 - 6

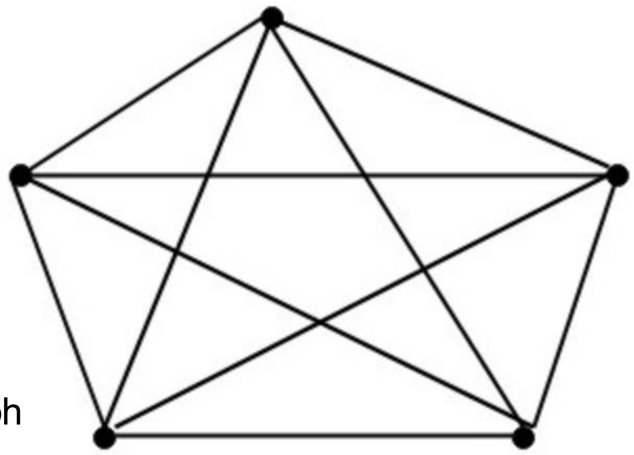
V(.) provides an *upper bound* for the number of paths necessary to achieve **branch coverage**

Complexity measure and its determination

Counting based on a graph

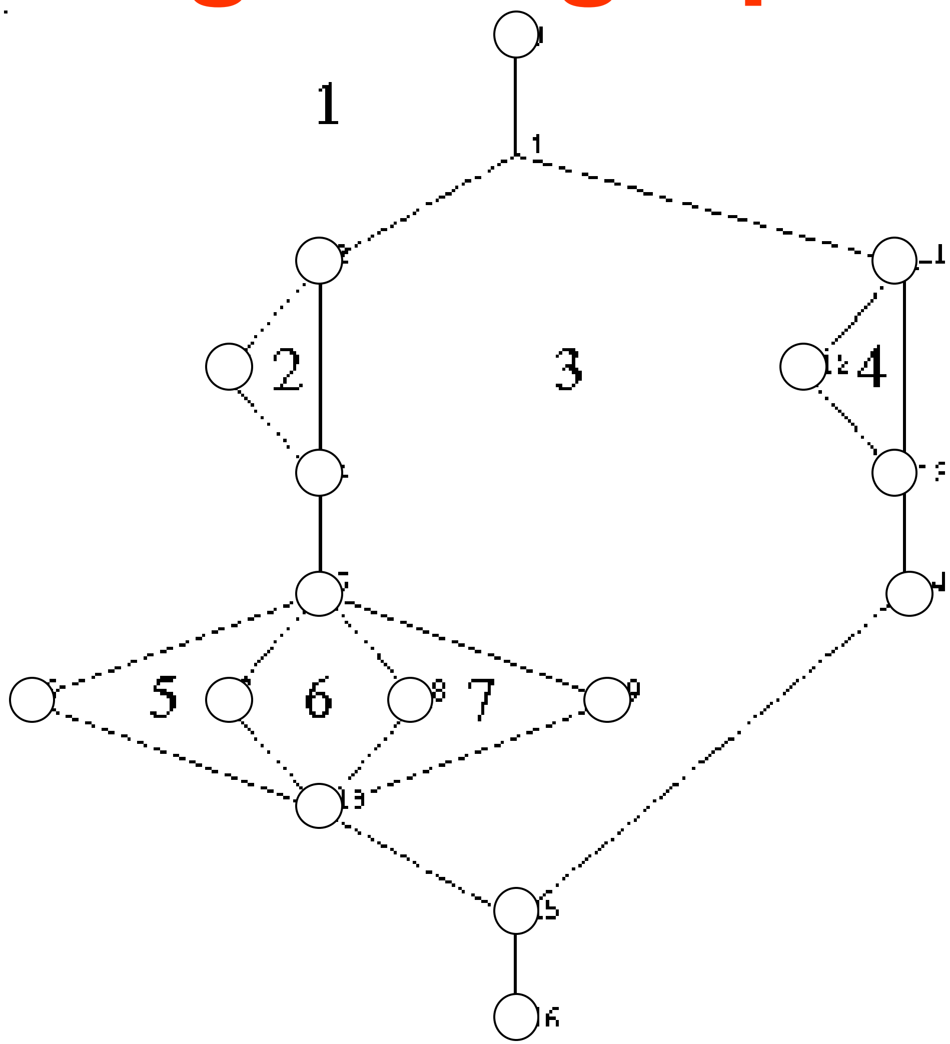
No of *binary* decision blocks +1

Counting flow graph regions (if the flow graph is planar, that is no edges cross)



non-planar graph

Counting flow graph regions



Complexity and its impact

Cyclomatic Complexity & Reliability Risk

- 1 – 10 Simple procedure, little risk
- 11- 20 More Complex, moderate risk
- 21 – 50 Complex , high risk
- >50 Untestable, VERY HIGH RISK

Cyclomatic Complexity & Bad Fix Probability

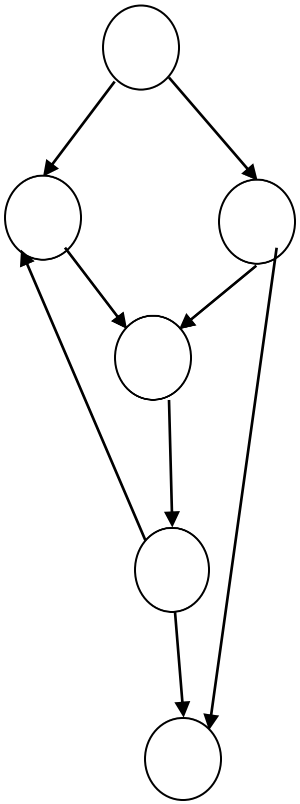
- 1 – 10 5%
- 20 –30 20%
- > 50 40%
- Approaching 100 60%

Essential Complexity (Unstructuredness) & Maintainability (future Reliability) Risk

- 1 – 4 Structured, little risk
- > 4 Unstructured, High Risk

Cyclomatic complexity: example

```
If  (s==1.0)
{
…..
label1: …..
…….
}
else
          {
          ….
          If(c==s) goto label2;
          ….
          }
…..
If  (q<=0.0) goto label1;
…..

label2:  a:=powf(w, 2.5);
```



From code to CFG: python (1)

Library **pycfg** tkinter (development of GUI)

https://pypi.org/project/pycfg/

```python
a= 10
while(a <= 0):
    if a == 5:
        print(a)
    a += 1
print("exited")
```

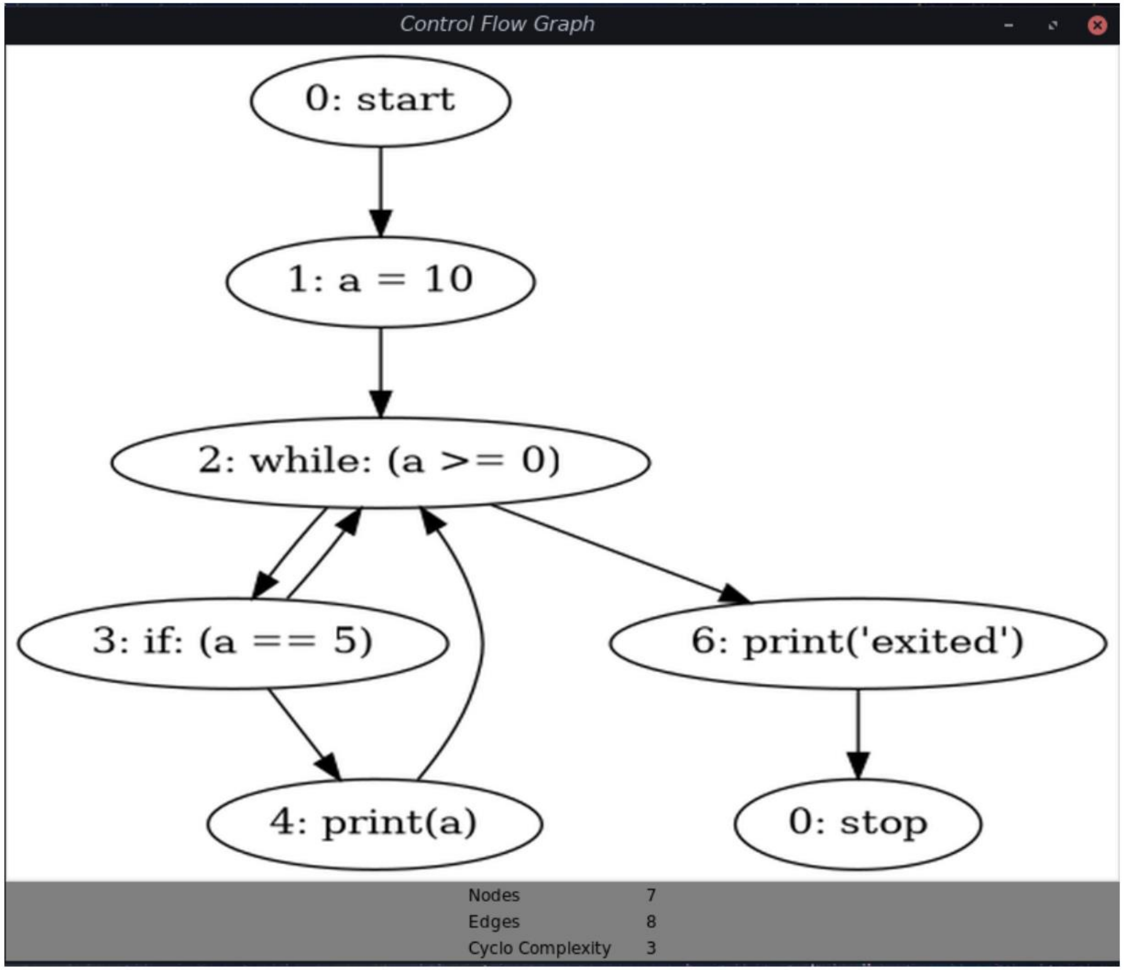
Output:

```
strict digraph "" {
        node [label="\N"];
        0       [label="0: start"];
        1       [label="1: a = 10"];
        0 -> 1;
        2       [label="2: while: (a >= 0)"]
;
        1 -> 2;
        3       [label="3: if: (a == 5)"];
        2 -> 3;
        5       [label="6: print('exited')"]
;
        2 -> 5;
        3 -> 2;
        4       [label="4: print(a)"];
        3 -> 4;
        6       [label="0: stop"];
        5 -> 6;
        4 -> 2;
}
```

From code to CFG: python (2)

Library pycfg **tkinter (development of GUI)**

```python
a= 10
while(a <= 0):
    if a == 5:
        print(a)
    a += 1
print("exited")
```



radon 5.1.0

```
pip install radon
```

- **cc**: compute Cyclomatic Complexity
- **raw**: compute raw metrics
- **mi**: compute Maintainability Index
- **hal**: compute Halstead complexity metrics

CC score	Rank	Risk
1 - 5	A	low - simple block
6 - 10	B	low - well structured and stable block
11 - 20	C	moderate - slightly complex block
21 - 30	D	more than moderate - more complex block
31 - 40	E	high - complex block, alarming
41+	F	very high - error-prone, unstable block

Block type	Letter
Function	F
Method	M
Class	C

```
(venv) (base) Witolds-iMac-Pro:pythonProject8 witold$ radon --h
usage: radon [-h] [-v] {cc,raw,mi,hal} ...

positional arguments:
  {cc,raw,mi,hal}
    cc              Analyze the given Python modules and compute Cyclomatic Complexity (CC).
    raw             Analyze the given Python modules and compute raw metrics.
    mi              Analyze the given Python modules and compute the Maintainability Index.
    hal             Analyze the given Python modules and compute their Halstead metrics.

optional arguments:
  -h, --help        show this help message and exit
  -v, --version     show program's version number and exit
(venv) (base) Witolds-iMac-Pro:pythonProject8 witold$
```

main.py

```python
# This is a sample Python script.

# Press ^R to execute it or replace it with your code.
# Press Double ⇧ to search everywhere for classes, files, tool windows, actions, and settings.

# See PyCharm help at https://www.jetbrains.com/help/pycharm/
def fun1(x):
    if x>1:
        x=x*x
    return(x)
def fun2(x):
    return(x)
def fun3(x):
    for i in range(x):
        for j in range(y):
            for k in range(z):
                print(i+j*k)
    return(z+1)

```

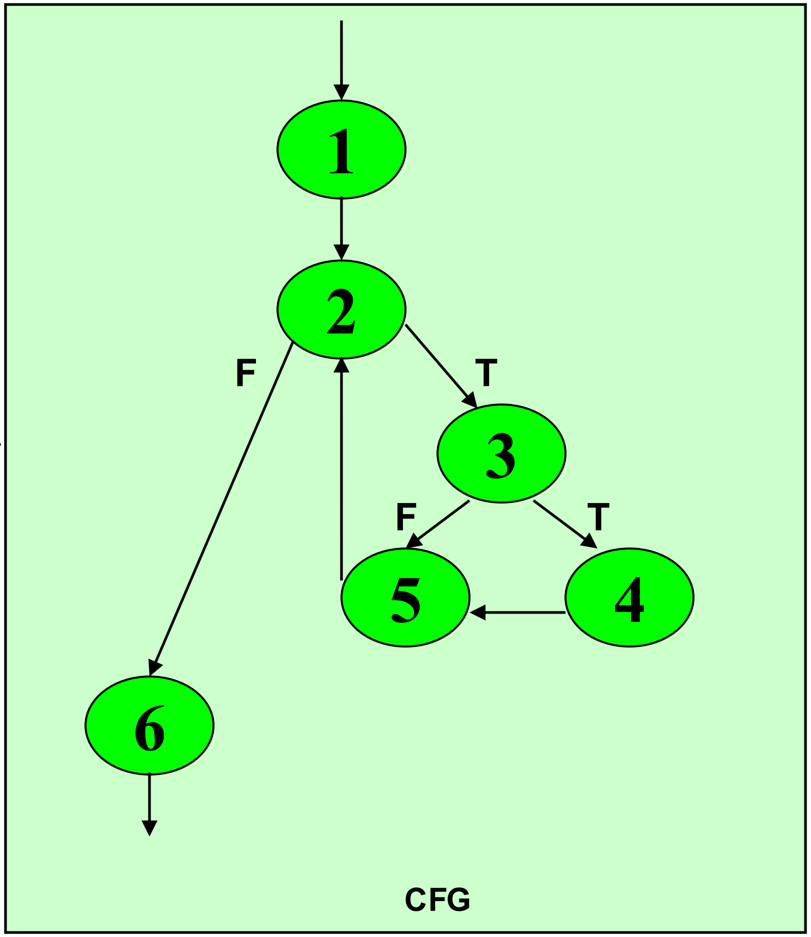
```
(venv) (base) Witolds-iMac-Pro:pythonProject8 witold$ radon cc main.py -s
main.py
    F 13:0 fun3 - A (4)
    F 7:0 fun1 - A (2)
    F 11:0 fun2 - A (1)
(venv) (base) Witolds-iMac-Pro:pythonProject8 witold$
```

```
(venv) (base) Witolds-iMac-Pro:pythonProject8 witold$ radon raw main.py -s
main.py
    LOC: 18
    LLOC: 12
    SLOC: 12
    Comments: 4
    Single comments: 4
    Multi: 0
    Blank: 2
    - Comment Stats
        (C % L): 22%
        (C % S): 33%
        (C + M % L): 22%
** Total **
    LOC: 18
    LLOC: 12
    SLOC: 12
    Comments: 4
    Single comments: 4
    Multi: 0
    Blank: 2
    - Comment Stats
        (C % L): 22%
        (C % S): 33%
        (C + M % L): 22%
```

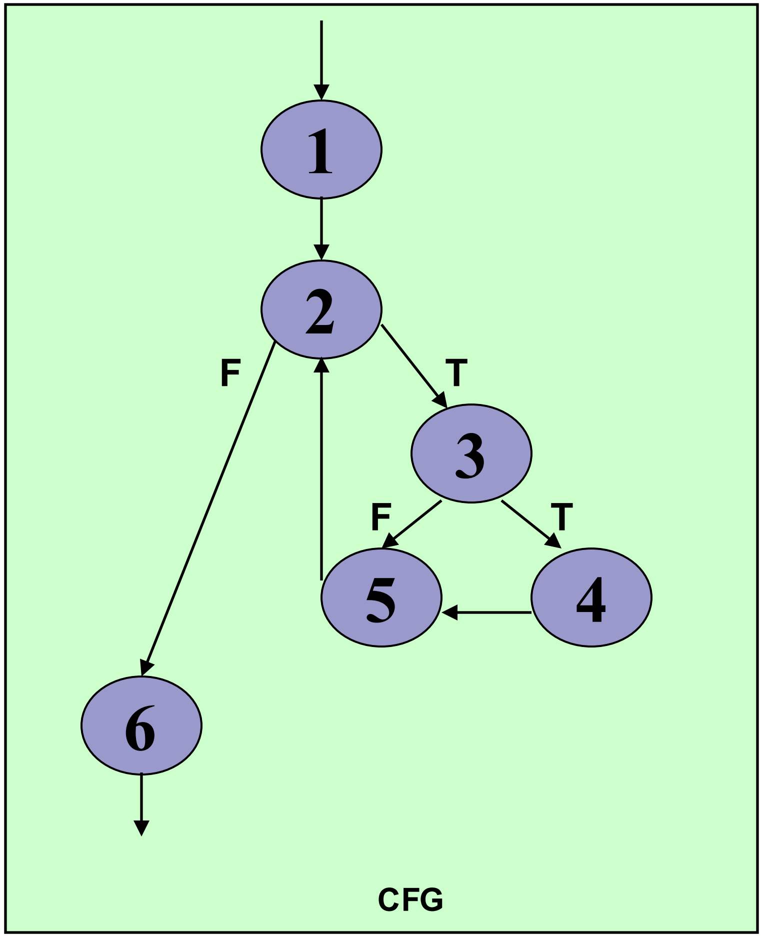
- **LOC**: the total number of lines of code
- **LLOC**: the number of logical lines of code
- **SLOC**: the number of source lines of code - not necessarily corresponding to the **LLOC**
- **comments**: the number of Python comment lines (i.e. only single-line comments `#`)
- **multi**: the number of lines representing multi-line strings
- **blank**: the number of blank lines (or whitespace-only ones)

Path Based Testing (1)

```
min = A[0];
I = 1;

while (I < N) {
      if (A[I] < min)
         min = A[I];
      I = I + 1;
}
print min
```



Path Based Testing(2)



- Cyclomatic complexity:
  - The number of ‘regions’ in the graph; OR
  - The number of predicates + 1.

Path Based Testing (3)

Independent path:

- An **executable** or **realizable path** through the graph from the start node to the end node that has not been traversed before.
- **Must** move along **at least one edge** that has not been yet traversed (an unvisited arc).

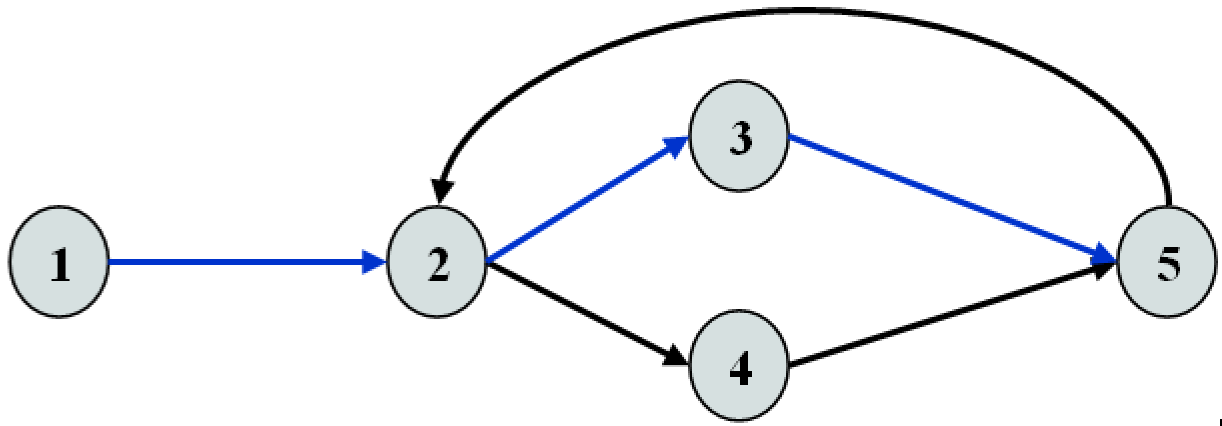
The number of independent paths to discover <= cyclomatic complexity.

Cyclomatic complexity - a maximum number of (linearly) independent paths

Select the Basis Path Set:

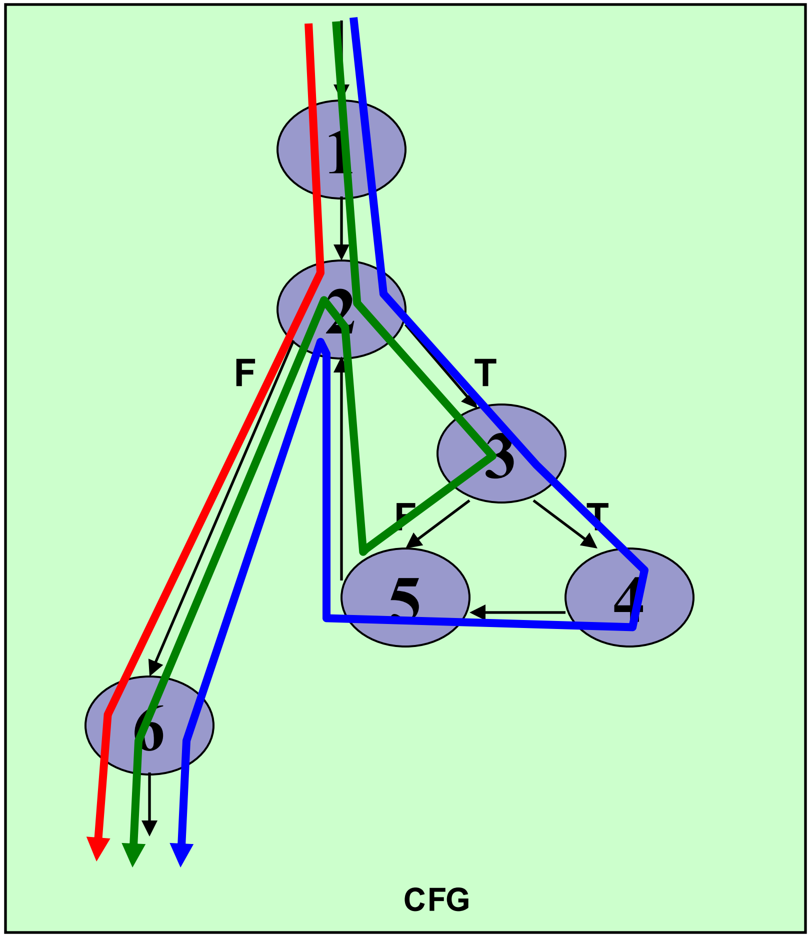
- It is the maximal set of *independent paths* in the flow graph.
- NOT a unique set.

Example



- 1-2-3-5 can be the first independent path; 1-2-4-5 is another; 1-2-3-5-2-4-5 is one more.
- There are only these 3 independent paths. The basis path set is then having 3 paths.
- Alternatively, if we had identified 1-2-3-5-2-4-5 as the first independent path, there would be no more independent paths.
- The number of independent paths therefore can vary according to the order we identify them.

Path Based Testing (3)

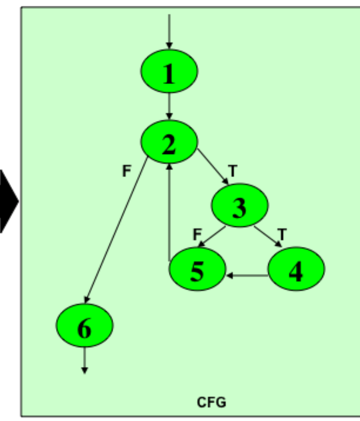


- Cyclomatic complexity = 3.
- Need at most **3** independent paths to cover the CFG.
- In this example:
  - [1 – 2 – 6]
  - [1 – 2 – 3 – 5 – 2 – 6]
  - [1 – 2 – 3 – 4 – 5 – 2 – 6]

Path Based Testing (4)

- Preparation of a test case for each independent path.
- In this example:
  - Path: [1 – 2 – 6]
    - Test Case: A = { 5, …}, N = 1
    - Expected Output: 5
  - Path: [1 – 2 – 3 – 5 – 2 – 6]
    - Test Case: A = { 5, 9, … }, N = 2
    - Expected Output: 5
  - Path: [1 – 2 – 3 – 4 – 5 – 2 – 6]
    - Test Case: A = { 8, 6, … }, N = 2
    - Expected Output: 6

```
min = A[0];
I = 1;

while (I < N) {
      if (A[I] < min)
         min = A[I];
      I = I + 1;
}
print min
```



- These tests will result a complete edge (branch) and statement coverage of the code.

Example

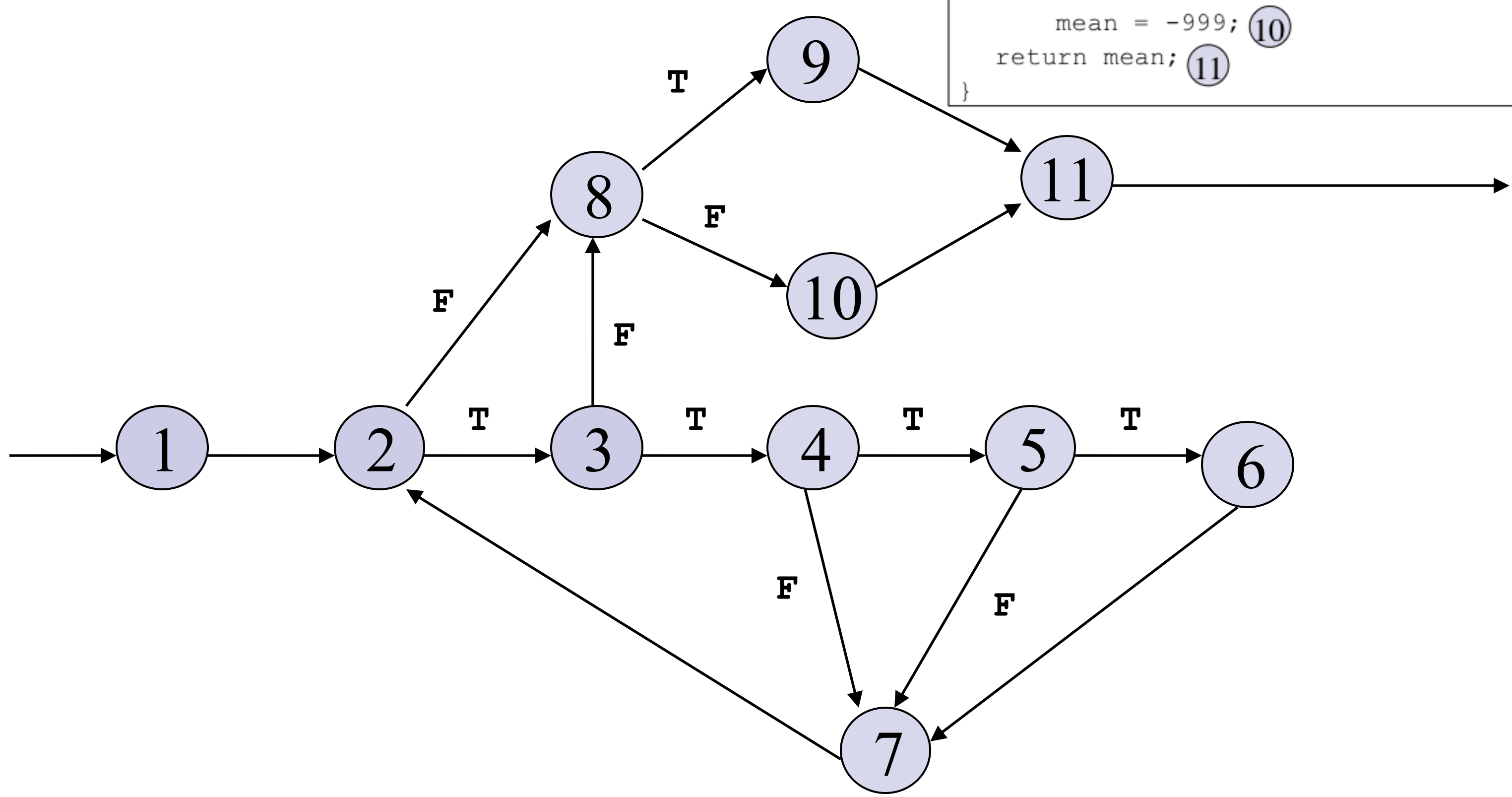
```
int average (int[ ] value, int min, int max, int N) {
   int i, totalValid, sum, mean;
   i = totalValid = sum = 0;
   while ( i < N && value[i] != -999 ) {
       if (value[i] >= min && value[i] <= max){
              totalValid += 1; sum += value[i];
       }
       i += 1;
   }
   if (totalValid > 0)
       mean = sum / totalValid;
   else
       mean = -999;
   return mean;
}
```

CFG

```
int average (int[ ] value, int min, int max, int N) {
   int i, totalValid, sum, mean;                  1
   i = totalValid = sum = 0;
   while ( i < N && value[i] != -999 ) {
                2     4          3           5
        if (value[i] >= min && value[i] <= max){
               totalValid += 1; sum += value[i];   6
        }
        i += 1;   7
   }
   if (totalValid > 0)   8
        mean = sum / totalValid;   9
   else
        mean = -999;   10
   return mean;   11
}
```

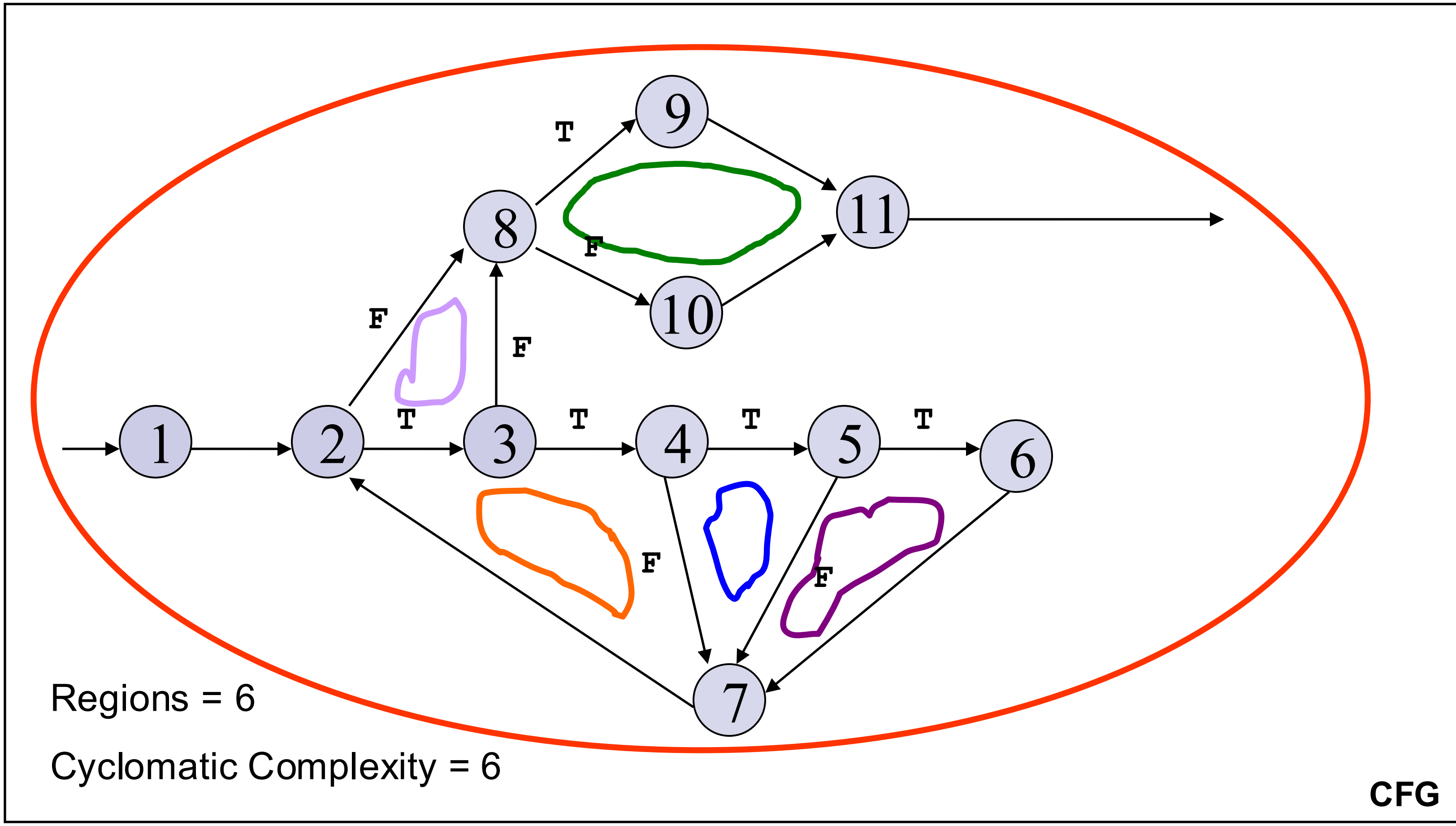
CFG

```
int average (int[ ] value, int min, int max, int N) {
   int i, totalValid, sum, mean;         } 1
   i = totalValid = sum = 0;
   while ( i < N && value[i] != -999 ) {
          2    4        3              5
       if (value[i] >= min && value[i] <= max){
               totalValid += 1; sum += value[i];  6
       }
       i += 1;  7
   }
   if (totalValid > 0)  8
       mean = sum / totalValid;  9
   else
       mean = -999;  10
   return mean;  11
}
```

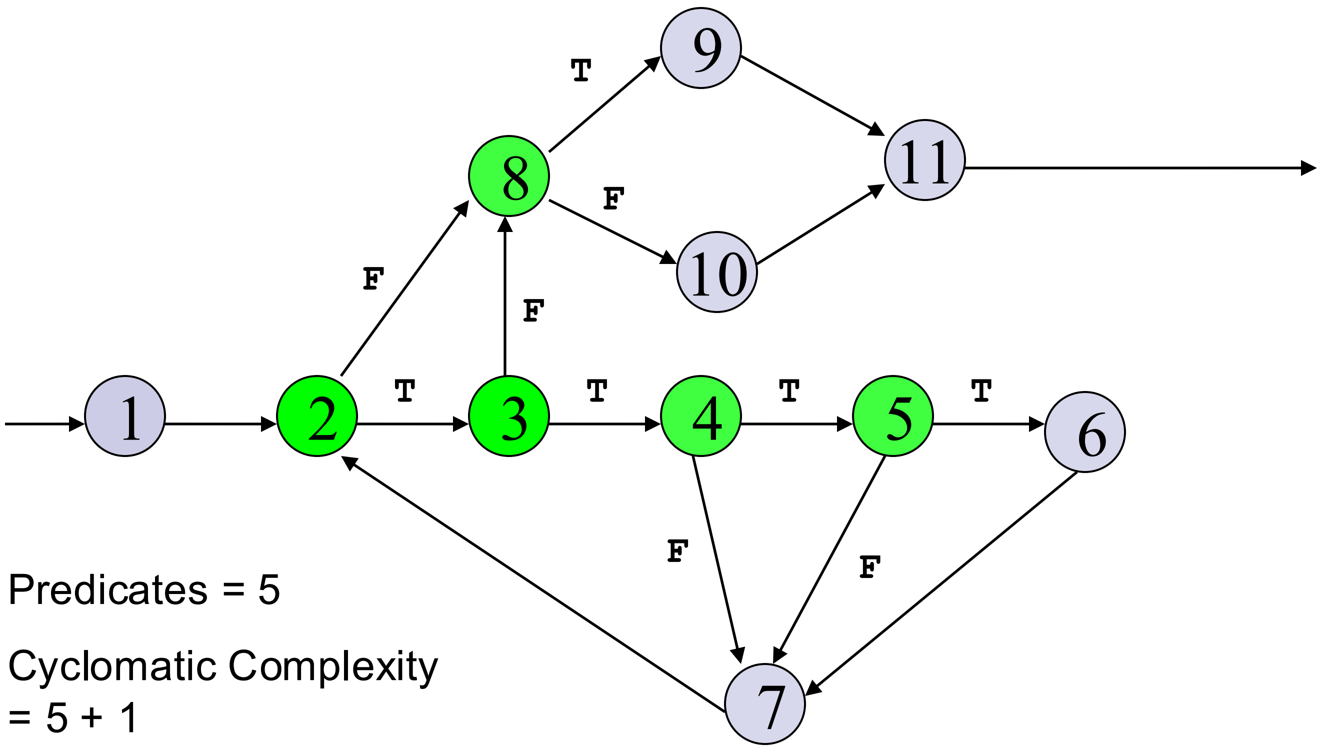


CFG

Cyclomatic Complexity



Cyclomatic Complexity



Predicates = 5

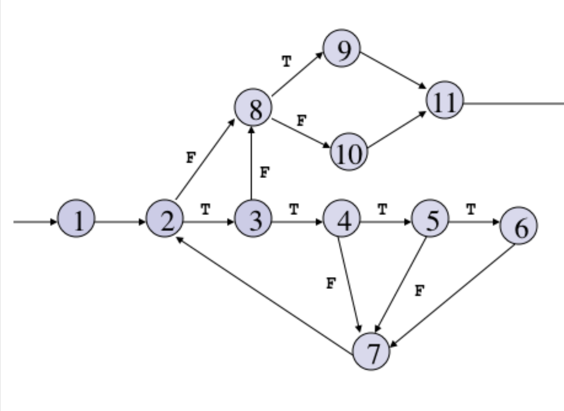
Cyclomatic Complexity
= 5 + 1
= 6

CFG

Basic Path Set

- Find at most 6 independent paths.
- Usually, simpler path == easier to develop a test case.
- However, some of the simpler paths are not possible (not realizable):
  - Example: [1 – 2 – 8 – 9 – 11].
    - Not Realizable (i.e., impossible in execution).
    - Verify this by tracing the code.
- Basic Path Set:
  - [1 – 2 – 8 – 10 – 11].
  - [1 – 2 – 3 – 8 – 10 – 11].
  - [1 – 2 – 3 – 4 – 7 – 2 – 8 – 10 – 11].
  - [1 – 2 – 3 – 4 – 5 – 7 – 2 – 8 – 10 – 11].
  - [1 – (2 – 3 – 4 – 5 – 6 – 7) – 2 – 8 – 9 – 11].
- In the last case, (…) represents possible repetition.

```
int average (int[ ] value, int min, int max, int N) {
   int i, totalValid, sum, mean;              } 1
   i = totalValid = sum = 0;
   while ( i < N && value[i] != -999 ) {
            2     4          3             5
        if (value[i] >= min && value[i] <= max){
                totalValid += 1; sum += value[i];  6
        }
        i += 1;  7
   }
   if (totalValid > 0)  8
        mean = sum / totalValid;  9
   else
        mean = -999;  10
   return mean;  11
}
```



Construction of Test Cases

- Path:
  - [1 – 2 – 8 – 10 – 11]
- Test Case:
  - `value = {…}` irrelevant.
  - `N = 0`
  - `min, max` irrelevant.
- Expected Output:
  - `average = -999`

```
... i = 0;                            1

while (i < N &&                       2
         value[i] != -999) {
      ......
}
if (totalValid > 0)                   8
      ......
else
      mean = -999;                    10

return mean;                          11
```

Construction of Test Cases

- Path:
  - [1 – 2 – 3 – 8 – 10 – 11]
- Test Case:
  - `value = {-999}`
  - `N = 1`
  - `min, max` irrelevant
- Expected Output:
  - `average = -999`

```
... i = 0;                          1

while (i < N &&                     2
       value[i] != -999)            3
     ......
}
if (totalValid > 0)                 8
     ......
else                                10
     mean = -999;
                                    11
return mean;
```

Construction of Test Cases

- Path:
  - [1 – 2 – 3 – 4 – 7 – 2 – 8 – 10 – 11]
- Test Case:
  - A single value in the `value[ ]` array which is smaller than *min.*
  - `value = { 25 }, N = 1, min = 30, max` irrelevant.
- Expected Output:
  - `average = -999`

- Path:
  - [1 – 2 – 3 – 4 – 5 – 7 – 2 – 8 – 10 – 11]
- Test Case:
  - A single value in the `value[ ]` array which is larger than *max.*
  - `value = { 99 }, N = 1, max = 90, min` irrelevant.
- Expected Output:
  - `average = -999`

Construction of Test Cases

- Path:
  - [1 – 2 – 3 – 4 – 5 – 6 – 7 – 2 – 8 – 9 – 11]
- Test Case:
  - A single valid value in the `value[ ]` array.
  - `value = { 25 }, N = 1, min = 0, max = 100`
- Expected Output:
  - `average = 25`

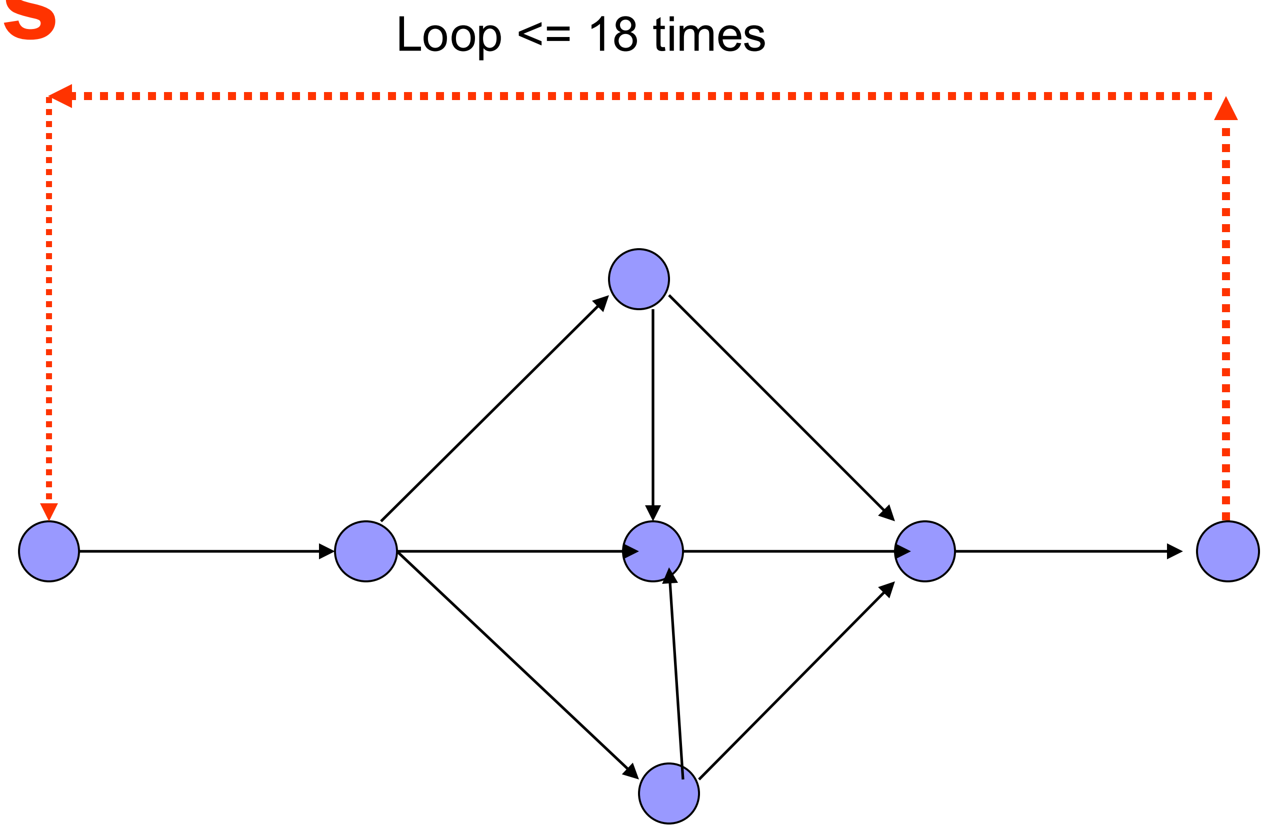
OR

- Path:
  - [1 – 2 – 3 – 4 – 5 – 6 – 7 – 2 – 3 – 4 – 5 – 6 – 7 – 2 – 8 – 9 – 11]
- Test Case:
  - Multiple valid values in the `value[ ]` array.
  - `value = { 25, 75 }, N = 2, min = 0, max = 100`
- Expected Output:
  - `average = 50`

Path Based Testing

- A test that:
  - Cover all statements.
  - Exercise all decisions (conditions).
- The cyclomatic complexity is an ***upper bound*** of the independent paths needed to cover the CFG.
  - If more paths are needed, then either cyclomatic complexity is wrong, or the paths chosen are incorrect.
- Although picking a complicated path that covers more than one unvisited edge is possible all times, it is not encouraged:
  - May be hard to design the test case.

Path coverage: graphs with loops



Number of paths:

$$5^1 + 5^2 + \ldots + 5^{18} = 4.77\ 10^{12}$$

Graphs with loops – practical considerations

- select a few critical paths instead of all possible paths
  - bounded/unbounded loops
    - zero times (skipping the loop)- could test (detect) loop initialization problems
    - Going through loop once – loop initialization and set-up problems
    - Going through loop twice – problems preventing from loop repetition
    - maximum (large) number of times
    - an average number of times (according to some statistics)

Deterministic and nondeterministic loops

Deterministic and nondeterministic loops

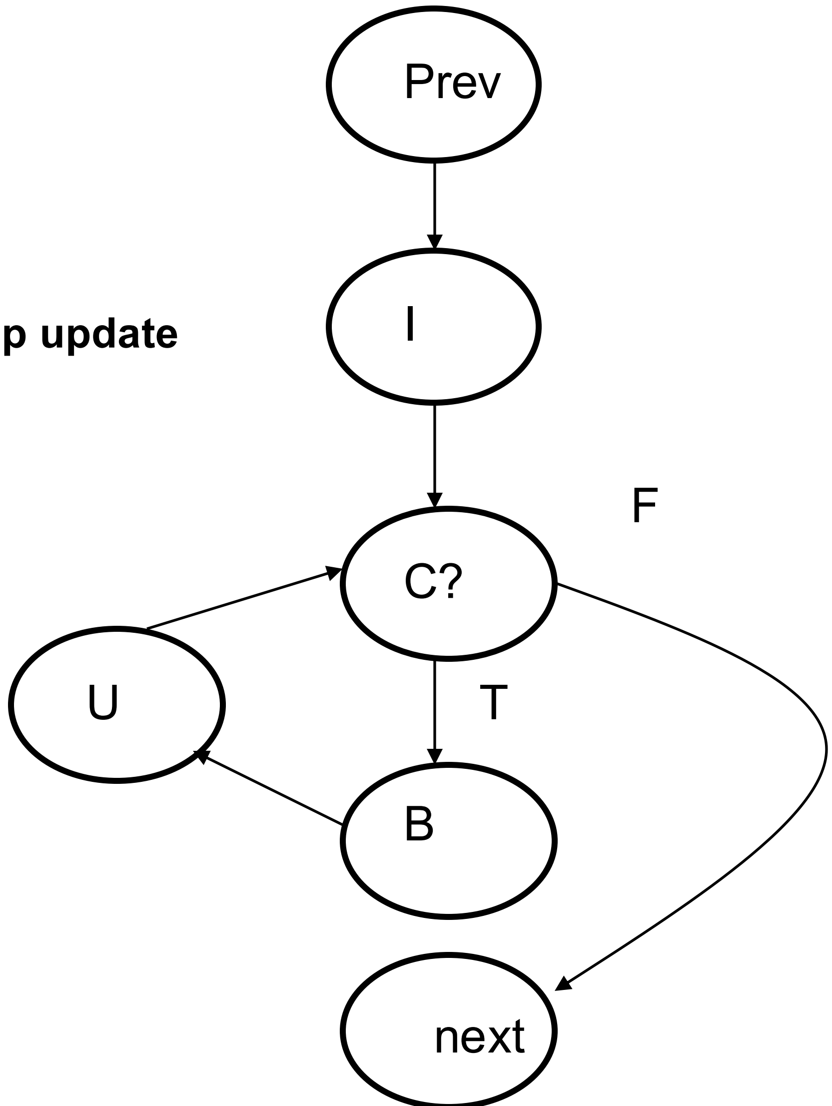
Loop initialization

for (I; C; U) {B}

Loop update

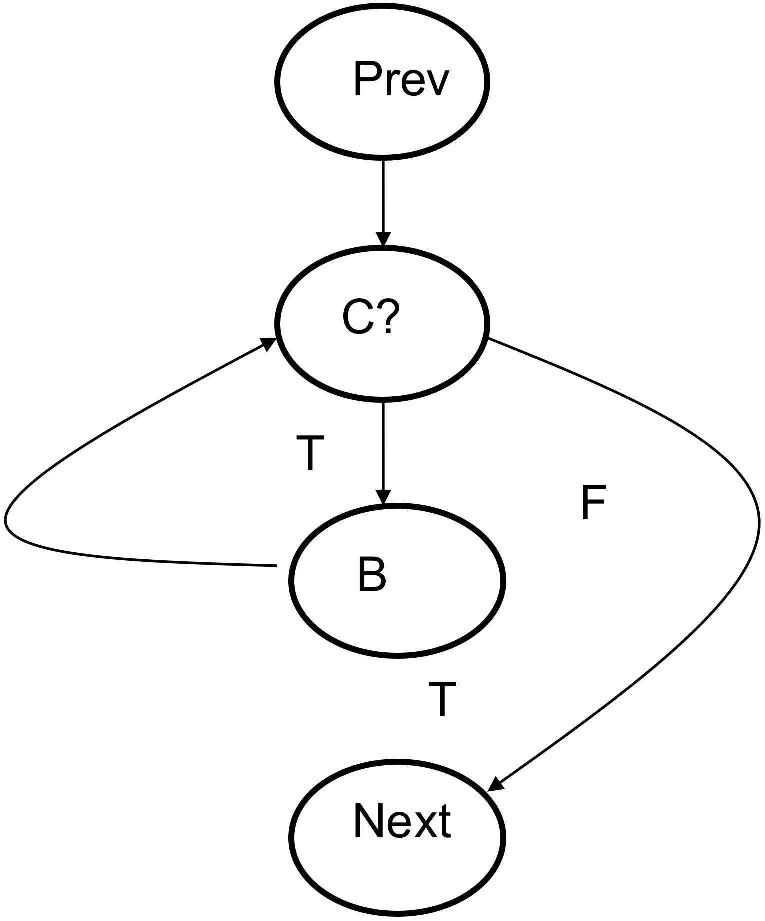
Loop condition

do (n) {B}



Deterministic and nondeterministic loops

while (C) do {B}



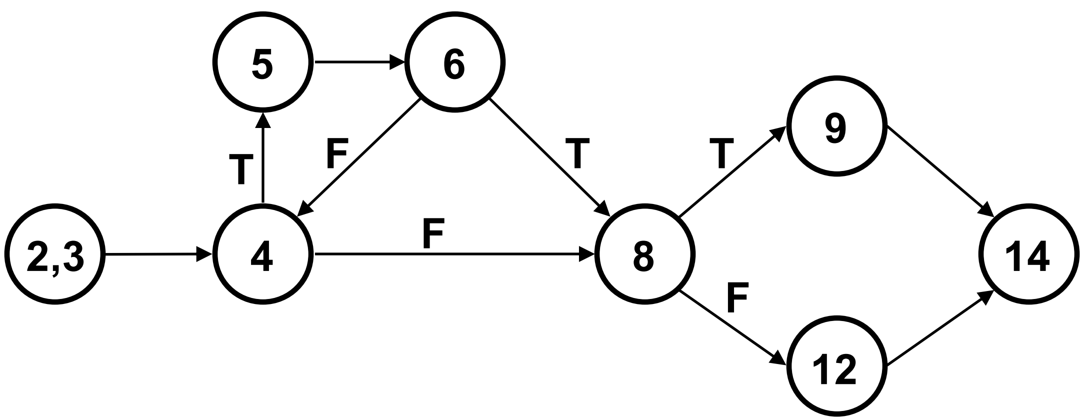
Graphs with loops- practical considerations

Exhaustive testing

100% coverage --impractical

EXAMPLES

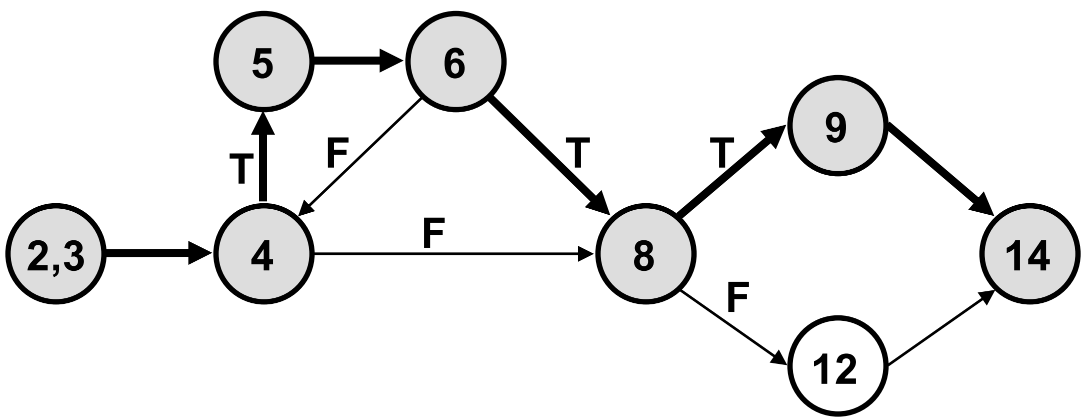
statements 1,7,10,11,13, and 15 are ignored



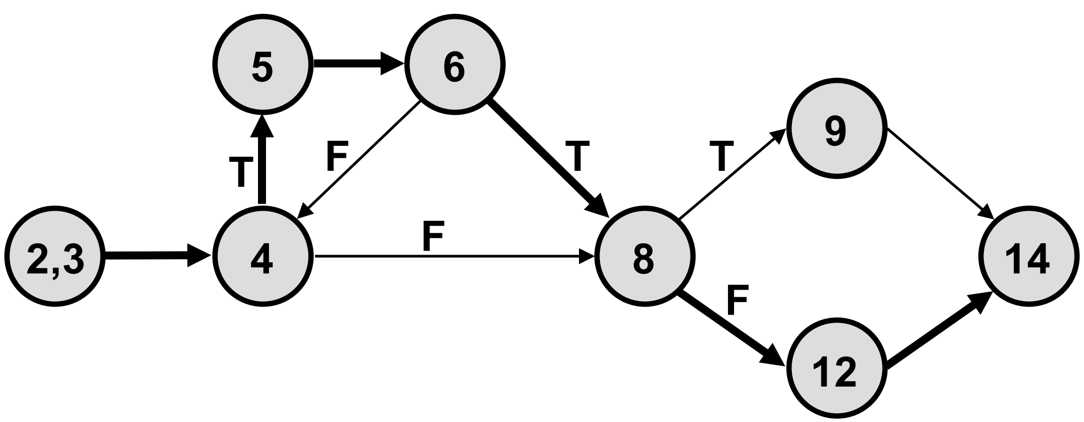

```
1    int Program() {
2        int x,y;
3        cin >> x >> y;
4        while (x>y) {
5            x = x-10;
6            if (x<=0) break;
7            }
8        if (x>=0 && y>0) {
9            y=y-x;
10           }
11         else {
12           y=0;
13           }
14       return y;
15   }
```

Statement coverage

- Statement coverage criterion using CFG
  - test case (10, 1)



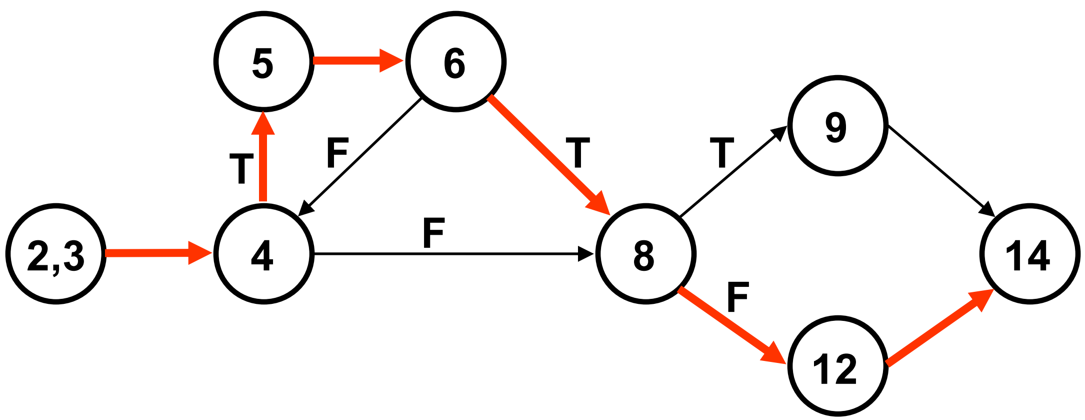
  - test case (5, 1)



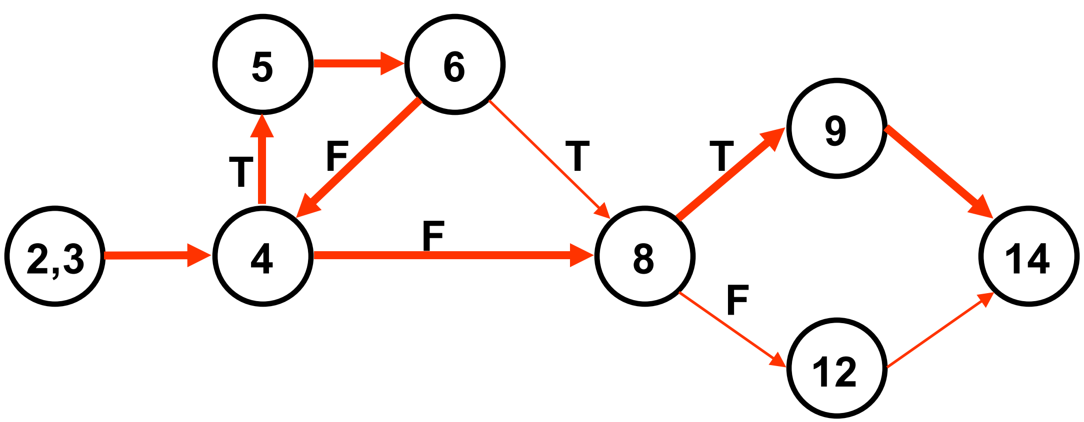

```
1    int Program() {
2        int x,y;
3        cin >> x >> y;
4        while (x>y) {
5            x = x-10;
6            if (x<=0) break;
7            }
8        if (x>=0 && y>0) {
9            y=y-x;
10           }
11         else {
12           y=0;
13           }
14       return y;
15     }
```

Branch coverage

- test case (5, 1)



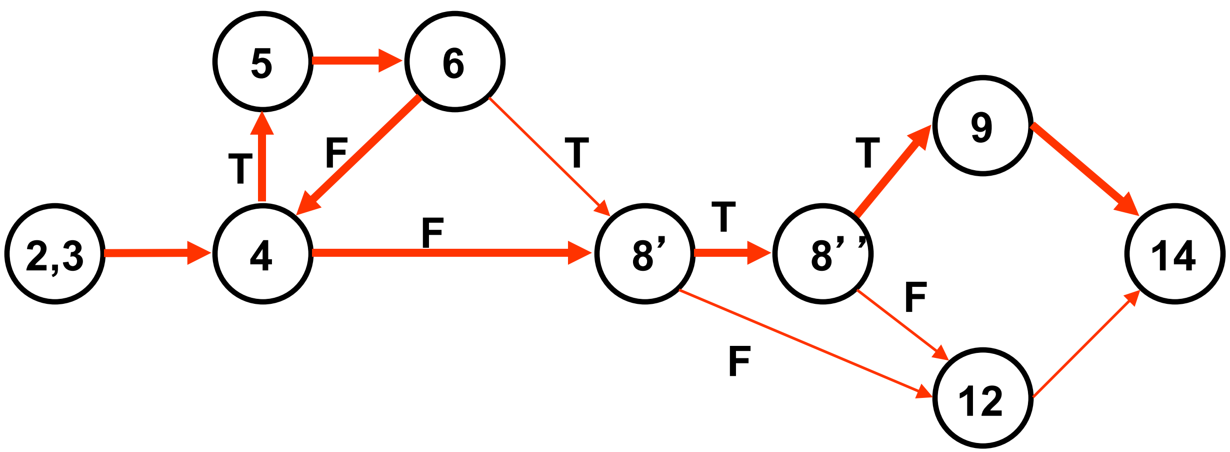
- test case (11, 1)




```
1    int Program() {
2        int x,y;
3        cin >> x >> y;
4        while (x>y) {
5            x = x-10;
6            if (x<=0) break;
7            }
8        if (x>=0 && y>0) {
9            y=y-x;
10           }
11         else {
12           y=0;
13           }
14       return y;
15     }
```

Condition/branch coverage

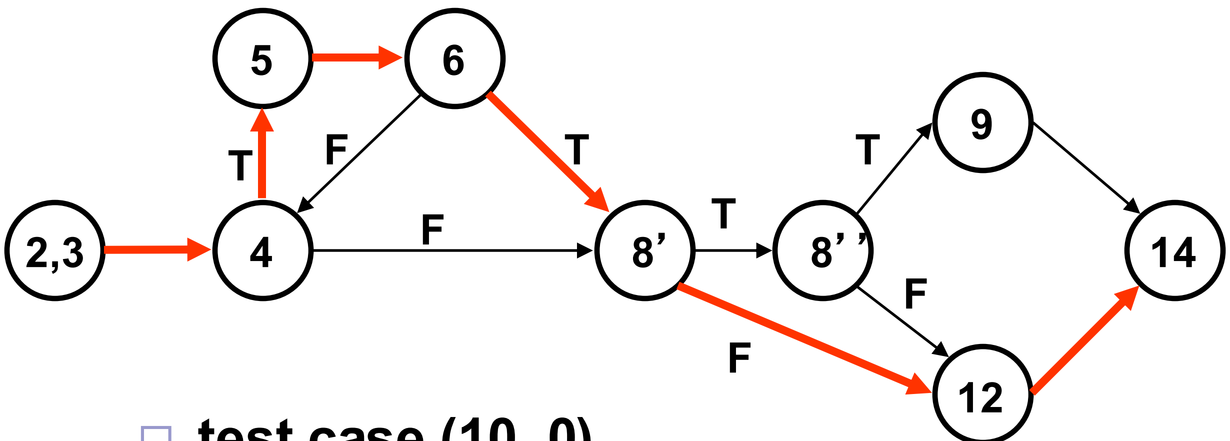
test case (11, 1)



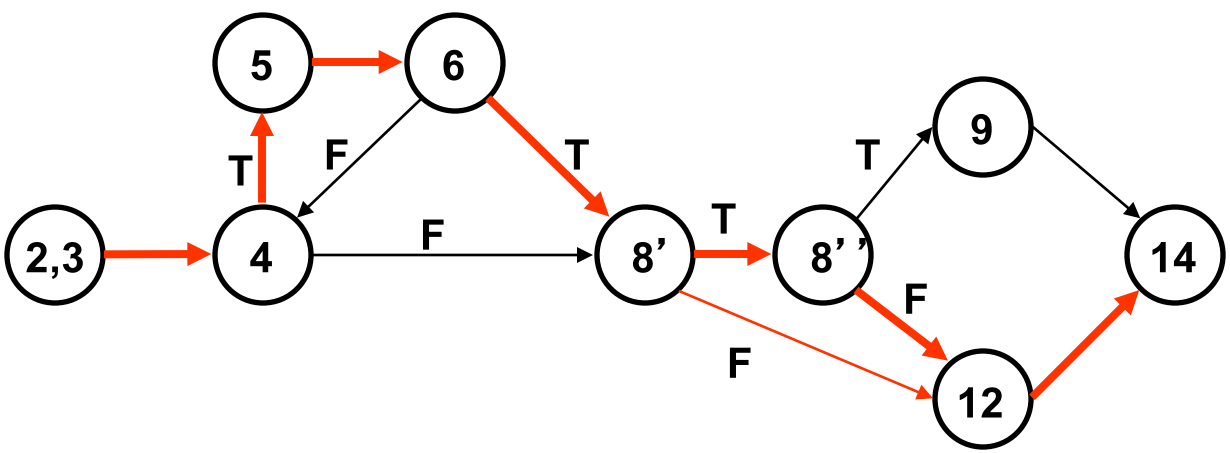

```
1    int Program() {
2        int x,y;
3        cin >> x >> y;
4        while (x>y) {
5            x = x-10;
6            if (x<=0) break;
7            }
8        if (x>=0 && y>0) {
9            y=y-x;
10           }
11         else {
12           y=0;
13           }
14       return y;
15   }
```

Condition/branch coverage

- test case (5, 1)



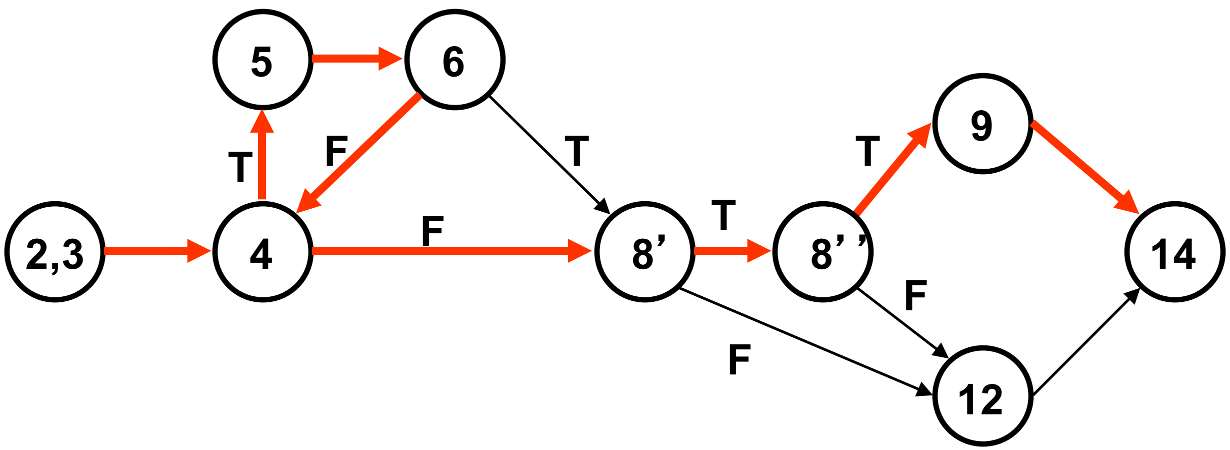
- test case (10, 0)




```
int Program() {
    int x,y;
    cin >> x >> y;
    while (x>y) {
        x = x-10;
        if (x<=0) break;
    }
    if (x>=0 && y>0) {
        y=y-x;
    }
    else {
        y=0;
    }
    return y;
}
```

Path coverage

- test case (11, 1)
- test case (21, 1)
- test case (31, 1)
- ….




```
1     int Program() {
2         int x,y;
3         cin >> x >> y;
4         while (x>y) {
5             x = x-10;
6             if (x<=0)
      break;
7             }
8         if (x>=0 && y>0) {
9             y=y-x;
10            }
11          else {
12            y=0;
13            }
14        return y;
15      }
```

Achieving coverage requirements?

condition/branch coverage (multiple condition coverage)

<u>every</u> <u>branch</u> in the code has to be executed at least once and <u>all possible combinations</u> of conditions in compound decisions must be exercised.

Testing all possible permutations of conditions (predicates)

All possible permutations of conditions (predicates)

Compilers do short-circuit predicate evaluation (from left to right) (efficiency)

See some related discussion in Chapter 8

Modified condition/branch criterion

Required by FAA (Federal Avionics Administration)

Avionics system:
Airborne systems written in Ada

	Number of Conditions, *n*									
	1	**2**	**3**	**4**	**5**	**6-10**	**11-15**	**16-20**	**21-35**	**36-76**
Number of Boolean expressions with *n* conditions	16491	2262	685	391	131	219	35	36	4	2

Modified condition/branch criterion

Required by FAA (Federal Avionics Administration)
FAA DO -178C standard, level A of criticality
https://en.wikipedia.org/wiki/DO-178C

Each condition should affect the decision independently:

Select test cases when the condition changes its truth value producing different values of output
-all other conditions are the same

n variables entails (n+1) test cases

Modified condition/branch criterion: example (1)

a && b && c

Test #	a	b	c	outcome
1	T	T	T	T
2	T	T	F	F
3	T	F	T	F
4	T	F	F	F
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

a: 1-5

b: 1-3

c: 1-2

Modified condition/branch criterion: example (2)

a || b

Test #	a	b	outcome
1	T	T	T
2	T	F	T
3	F	T	T
4	F	F	F

a: 4-2

b: 4-3

Modified condition/branch criterion: example (3)

z=(a || b)&&(c||d)

	a	*b*	*c*	*d*	*z*
1	F	F	F	F	F
2	F	F	F	T	F
3	F	F	T	F	F
4	F	F	T	T	F
5	F	T	F	F	F
6	F	T	F	T	T
7	F	T	T	F	T
8	F	T	T	T	T
9	T	F	F	F	F
10	T	F	F	T	T
11	T	F	T	F	T
12	T	F	T	T	T
13	T	T	F	F	F
14	T	T	F	T	T
15	T	T	T	F	T
16	T	T	T	T	T

Modified condition/branch criterion: example (3, cont)

z=(a || b)&&(c||d)

	a	*b*	*c*	*d*	*z*
1	F	F	F	F	F
2	F	F	F	T	F
3	F	F	T	F	F
4	F	F	T	T	F
5	F	T	F	F	F
6	F	T	F	T	T
7	F	T	T	F	T
8	F	T	T	T	T
9	T	F	F	F	F
10	T	F	F	T	T
11	T	F	T	F	T
12	T	F	T	T	T
13	T	T	F	F	F
14	T	T	F	T	T
15	T	T	T	F	T
16	T	T	T	T	T

For *a* 2-10, 3-11, 4-12
For *b*: 2-6, 3-7, 4-8
For *c*: 5-7 9-11, 13-15
For *d*: 5-6, 9-10, 13-14

Short circuiting(1)

All possible permutations of conditions (predicates) exercised

Short circuiting

Compilers do short-circuit predicate evaluation (from left to right) (efficiency)

and A&& B && C… short circuit when the first term is FALSE

or A|| B|| C short circuit when the first term is TRUE

Instead of A || B is A || funcB()

and funcB() initializes a data structure

Short circuiting (2)

Short circuiting: C++ JAVA C (some)

No short circuiting PASCAL

(A && B && (C || (D && E))

Test	A	B	C	D	E	Decision
1	F					F
2	T	F				F
3	T	T	F	F		F
4	T	T	F	T	F	F
5	T	T	F	T	T	T
6	T	T	T			T

Short-circuiting and Coupling

Multiple occurrences of a condition in an expression

A || (!A && B)

A and !A are *coupled*

Interpretation of term condition: <u>uncoupled condition</u>

Symbolic execution

Uses *symbolic* values, instead of concrete values, as inputs

Represents the values of program variables as symbolic expressions

During symbolic execution, the state of executed program includes

*symbolic values of program variables,

*path constraint on the symbolic values to reach the point, and

*program counter

Symbolic execution

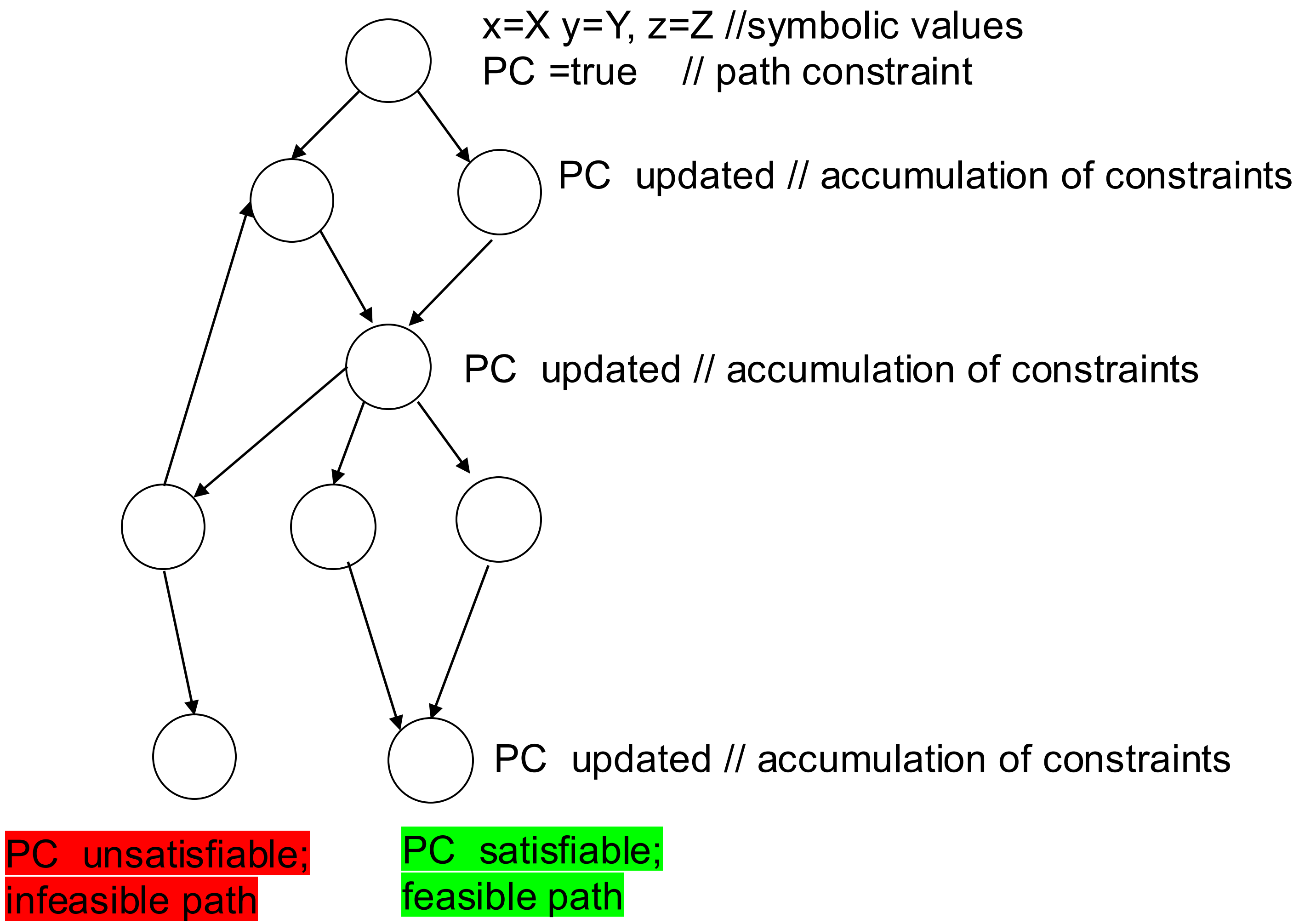
Path constraint (PC) – Boolean formula over the symbolic values; accumulation of constraints the input must satisfy for an execution to follow that path

at each branch, the PC is updated with constraints on the input:

-if PC becomes unsatisfiable, the program path is *infeasible*
-
-If PC satisfiable, any solution of the PC is a program input that executes the corresponding path

Program counter – identifies next statement to be executed

Symbolic execution: propagating constraints



Symbolic execution - example

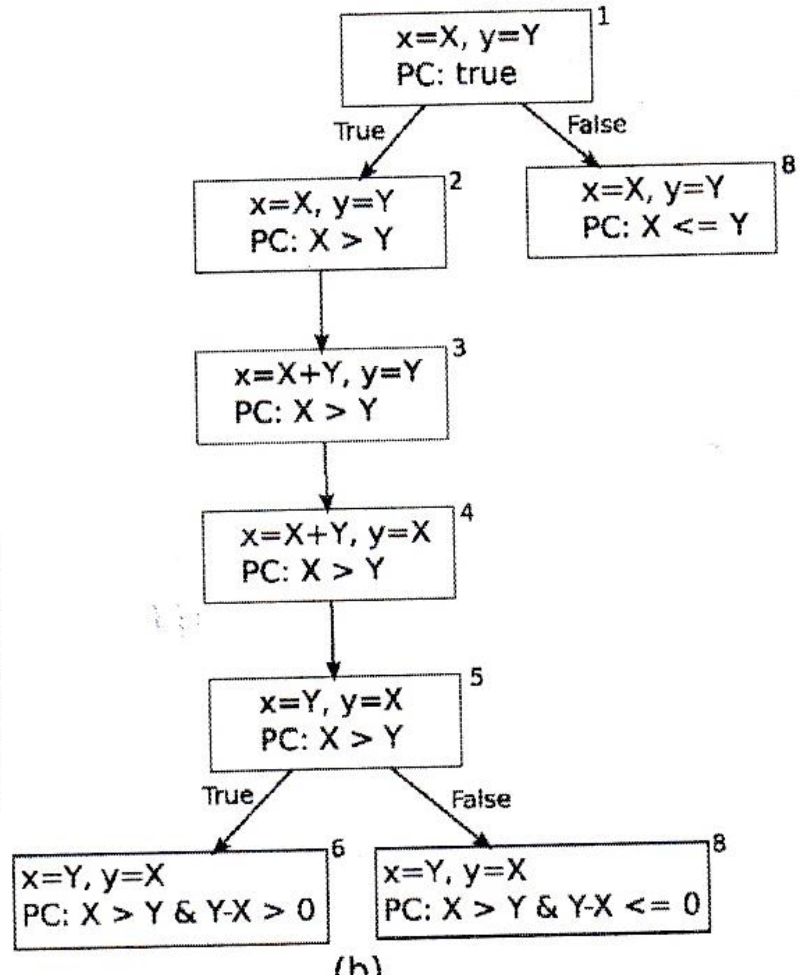
Swapping two integer values x, y (if x is greater than y)

```
int x, y;
if(x > y){
   x = x+y;
   y = x-y;
   x = x-y;
   if(x - y > 0)
      assert false;
}
print(x, y)
```

Symbolic execution – example(1)

```
int x, y;
if(x > y){
   x = x+y;
   y = x-y;
   x = x-y;
   if(x - y > 0)
      assert false;
}
print(x, y)
```

Symbolic execution tree

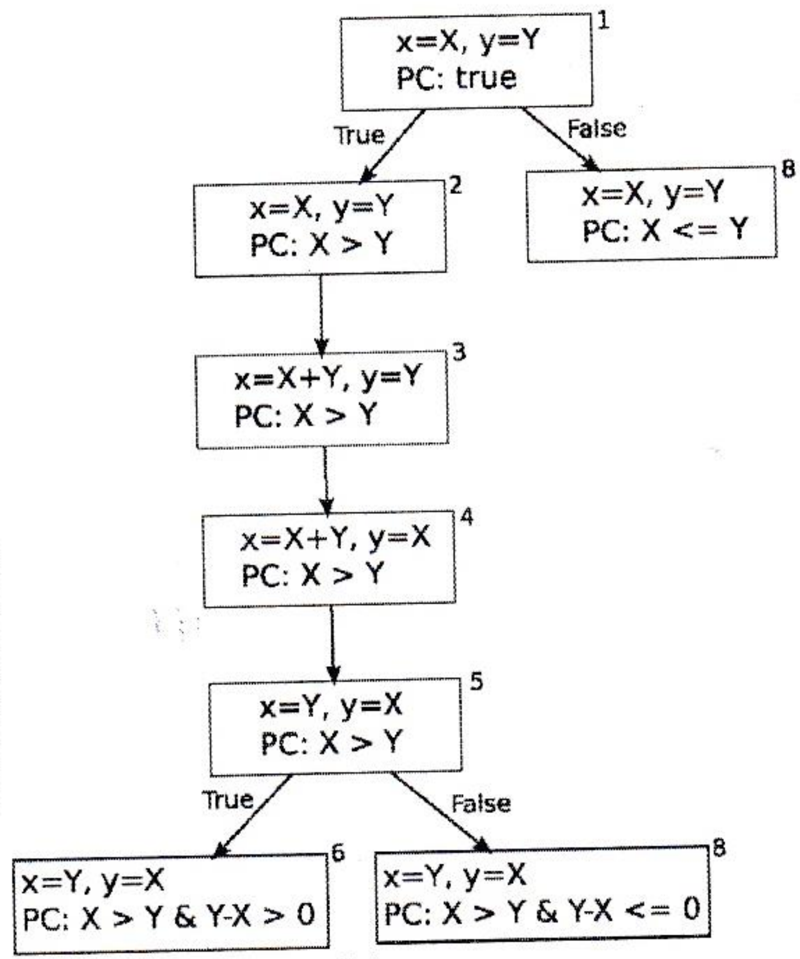


Symbolic execution – example (2)

```
   int x, y;
1  if(x > y){
2     x = x+y;
3     y = x-y;
4     x = x-y;
5     if(x - y > 0)
6        assert false;
7  }
8  print(x, y)
```

Path constraints

Path	PC	Program Input
1,8	X <= Y	X=1 Y=1
1,2,3,4,5,8	X>Y & Y-X<=0	X=2 Y=1
1,2,3,4,5,6	X>Y & Y-X>0	none



(b)

Symbolic execution – challenges(1)

Symbolic execution requires solving complex constraints (constraint solvers)

Path explosion – difficult to symbolically execute a large subset of program paths:

(i) extremely large number of paths,

(ii) symbolic execution can incur high computational overhead

Path divergence- computing precise constraints (given multiple programming languages, possible different initialization, side effects) requires additional effort;

the path that program takes may diverge from the path for which the test data were generated

Symbolic execution – challenges(2)

Complex constraints
(nonlinear operations, multiplication, division, sin, log…)

Path constraint: Boolean expression describing the conditions satisfied for explored path
SAT – Satisfiability Theorem
is there an assignment to Boolean variables making the formula true?

literals

clauses

$$\omega_1 = (b \vee c)$$

$$\omega_2 = (\neg a \vee \neg d)$$

$$\omega_3 = (\neg b \vee d)$$

$$\varphi = \omega_1 \wedge \omega_2 \wedge \omega_3$$

$$A = \{a=0, b=1, c=0, d=1\}$$

SATisfying assignment!

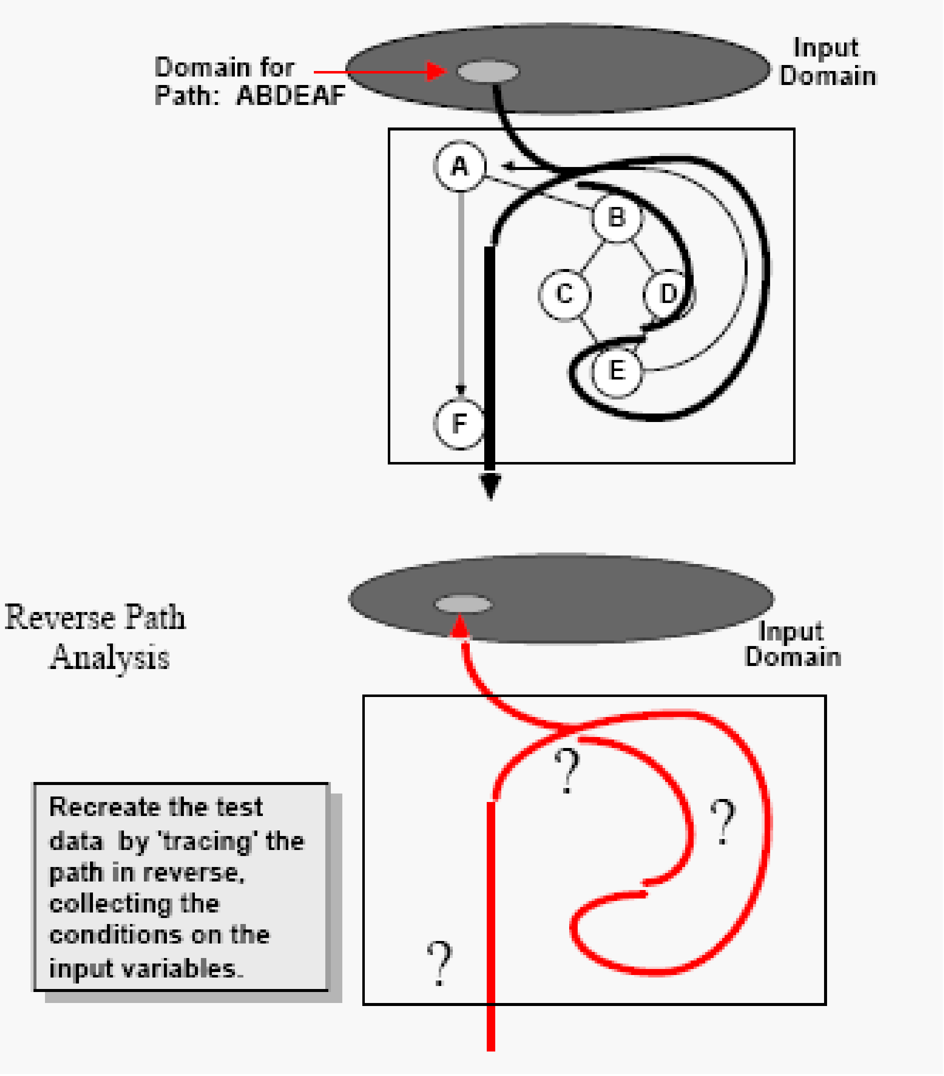
SMT – Satisfiability Modulo Theorem/theory; replacement of predicates by variables (arithmetic, arrays, bit vectors); NP hard

Concolic testing

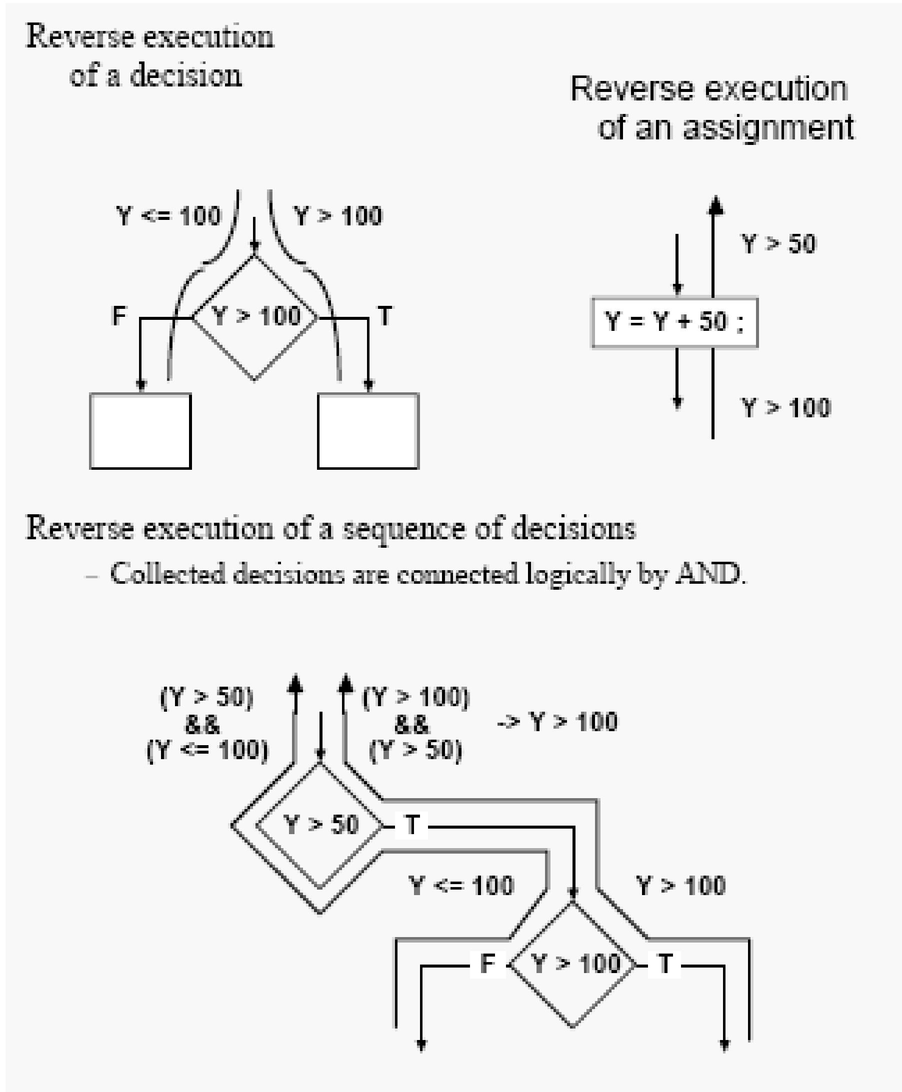
Hybrid approach combining concrete and symbolic testing

Concrete testing + symb**olic** testing = concolic

Reverse path analysis (1)



Reverse path analysis (2)



Control flow testing: Guidelines

- **Statement coverage** - to be accepted as the minimum (paths, not statements alone transform input into output); weak coverage measure, 100% coverage

- **Branch coverage** – most effective for control flow coverage completed at some level; required level of coverage of 85%

- **Path coverage** – highly demanding, should be limited to critical modules and limited to a few functions with life criticality features

Control flow testing: Summary

- Focus on basic decision and control flow problems (control flow graphs)
- Execution paths
- Process-oriented (step-by-step path)

The infeasibility problem

- Syntactically indicated behaviors (statements, edges, etc.) are often *impossible*
  - unreachable code, infeasible edges, paths, etc.
- Adequacy criteria may be impossible to satisfy
  - manual justification for omitting each impossible test case
  - adequacy “scores” based on coverage
    - example: 95% statement coverage

Further problem

- What if the code omits the implementation of some part of the specification?
- White box test cases derived from the code will ignore that part of the specification