Software Quality Assurance and Testing

Structural Testing

Outline

- Case Study - Airbus A320
- Program Models and Graphs
- White Box (Structural) Testing
- Control Flow Coverage
- Beyond Branch and Condition Testing
- Path Testing
- Cyclomatic Complexity
- Java Code Coverage Tools
- Summary

Case Study – Airbus A320

Case Study – Airbus A320

- Launched in 1984
- First civilian fly-by-wire computer system so advanced it can land plane virtually unassisted
- No instrument dials – 6 CRTs

Case Study – Airbus A320 – Fatal Accidents

- Air France Flight 296

  Alsace, France, June 26, 1988
  - The airplane software interpreted the low altitude/downed gear as "We're about to land"
  - Would not allow the pilot to control the throttle.
  - 3 people died, 133 survived
- Indian Airline Flight 605

  Bangalore, India, February 14, 1990
  - 92 people died, 56 survived
- Air Inter Flight 148

  Mont Sainte Odile, January 20, 1992
  - 87 people died, 9 survived

Case Study – Airbus A320: What Were the Causes?

- The fly-by-wire system could ignore pilot actions.
- Warning system alerts only seconds before accident.
  - no time to react
- Programmed landing maneuvers with bug in altitude calculation
  - Altimeter showed the plane was higher than its actual altitude
- Flight path angle and vertical speed indicator have the same display format
  - confuses pilots

Note:

In vertical speed mode "-3.3" means a descent rate of 3300 feet/min.

In TRK/FPA (track/flight path angle) mode this would have meant a (correct) -3.3 deg descent angle.

Program Models and Graphs

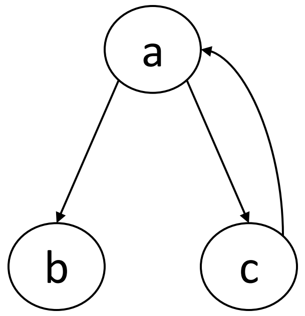
Properties of Models

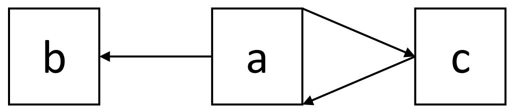
- Compact:
  - representation of a system
- Predictive:
  - represent some salient characteristics
  - well enough to distinguish between *good* and *bad*
  - no single model represents all characteristics
- Semantically meaningful:
  - permits diagnosis of the causes of failure
- Sufficiently general:
  - general enough for practical use

Graph Representations: directed graphs

- Directed graph:
  - N (set of nodes)
  - E (relation on the set of nodes) edges

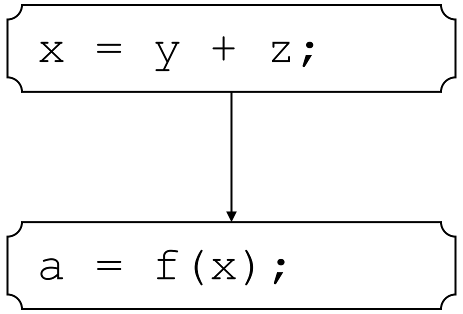
Nodes: {a, b, c}
Edges: {(a,b), (a, c), (c, a)}





Graph Representations: labels and code

- We can label nodes with the names or descriptions of the entities they represent.
  - If nodes a and b represent program regions containing assignment statements, we might draw the two nodes and an edge (a,b) connecting them in this way:

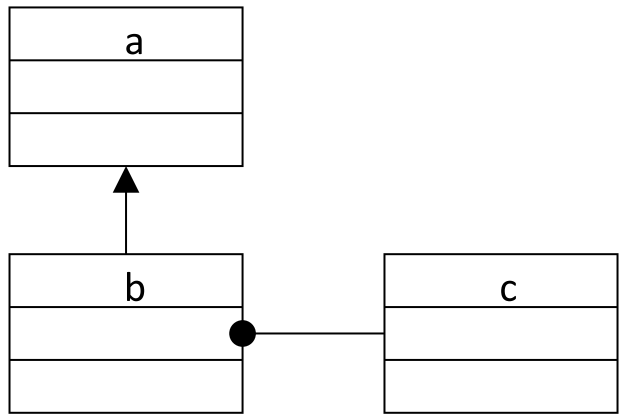


Multidimensional Graph Representations

- Sometimes we draw a single diagram to represent more than one directed graph, drawing the shared nodes only once
  - class B extends (is a subclass of) class A
  - class B has a field that is an object of type C

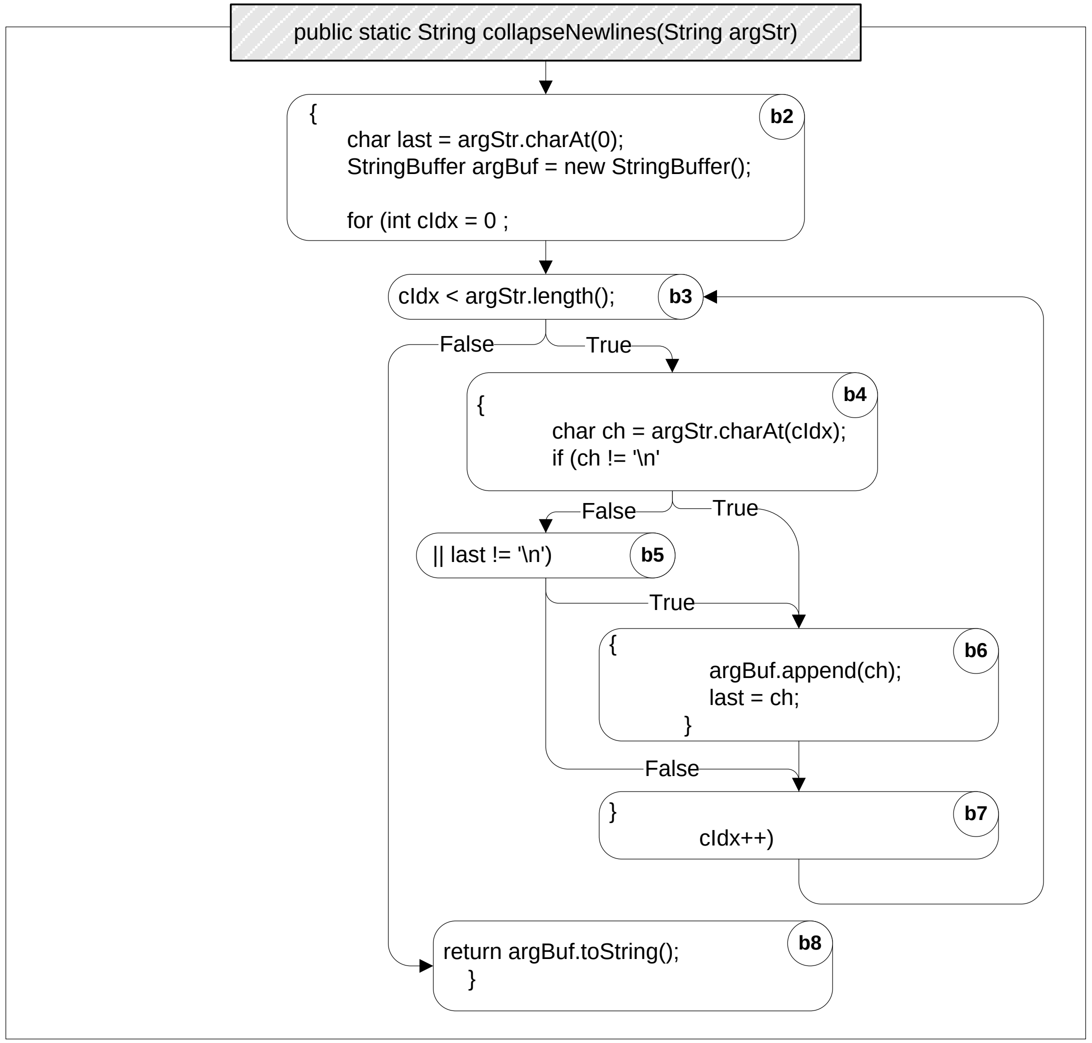
extends relation
  NODES = {A, B, C}
  EDGES = {(A,B)}

includes relation
  NODES = {A, B, C}
  EDGES = {(B,C)}



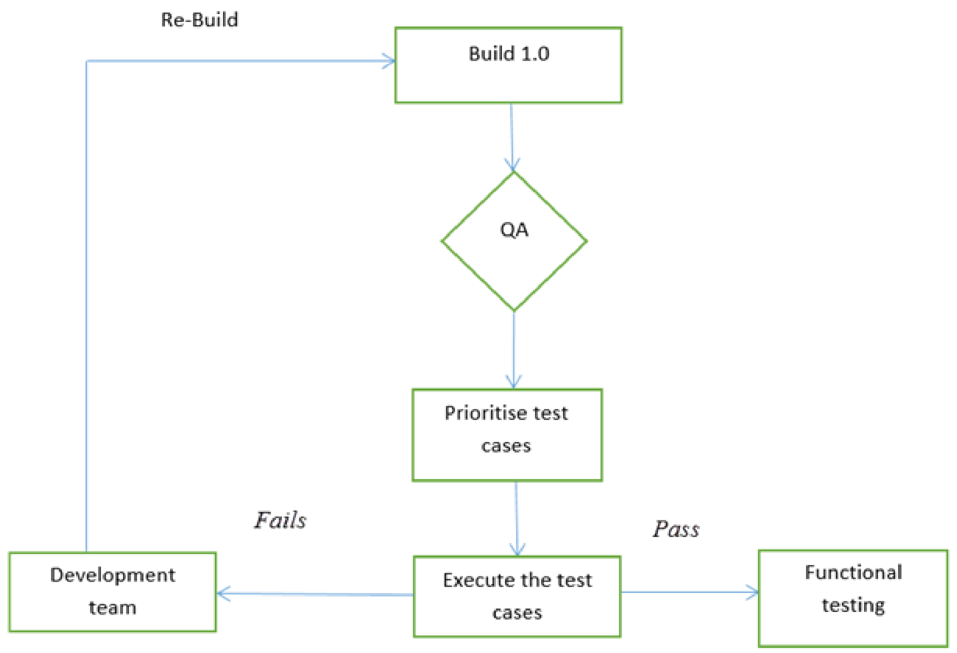
Example of Control Flow Graph

```
public static String collapseNewlines(String argStr)
  {
      char last = argStr.charAt(0);
      StringBuffer argBuf = new StringBuffer();

      for (int cIdx = 0 ; cIdx < argStr.length(); cIdx++)
      {
          char ch = argStr.charAt(cIdx);
          if (ch != '\n' || last != '\n')
          {
              argBuf.append(ch);
              last = ch;
          }
      }

      return argBuf.toString();
  }
```


public static String collapseNewlines(String argStr)
{ char last = argStr.charAt(0); StringBuffer argBuf = new StringBuffer(); for (int cIdx = 0 ;
b2
cIdx < argStr.length();
b3
False
True
{ char ch = argStr.charAt(cIdx); if (ch != '\n'
b4
False
True
|| last != '\n')
b5
True
{ argBuf.append(ch); last = ch; }
b6
False
} cIdx++)
b7
return argBuf.toString(); }
b8

Flowchart

- Testers usually use flowcharts in the test plan, test strategy, requirements artifacts or other process documents.
- Here are two ways we testers use flow charts:
  - Flowcharts for control flow and statistical analysis
  - Flow charts for process illustration



Control Flow Graph (CFG)

- Intra-procedural control flow graph
- Nodes = regions of source code, basic blocks
  - maximal program region with a single entry and single exit
    - Statements are grouped in single block
    - Single statement can also be broken into multiple nodes
- Directed edges = control flow
  - program execution may proceed from one node to another

White Box Testing
a.k.a. Structural Testing

Structural Testing

- Judging the *thoroughness* of a test suite based on the *structure* of the program
- Compare to functional (requirements based, black-box) testing
  - Structural testing is still testing product functionality against its specification.
  - Only the measure of thoroughness has changed.
- Usually done by the programmers as part of *unit testing*

Why Structural Testing?

- "What is *missing* in our test suite?"
- If part of a program is not executed by any test case in the suite, defects in that part cannot be exposed.
- What is a "part"?
  - Typically, a control flow element or combination:
    - Statements (or CFG nodes), branches (or CFG edges)
    - Fragments and combinations: conditions, paths
- Complements functional testing:
  - Another way to recognize cases that are treated *differently*

Structural Testing

- Structural testing techniques serve two purposes:
    - Test coverage measurement
        - We can assess the amount of testing performed by tests derived from e.g. specification-based technique to asses coverage.
    - Structural test case design
        - We can generate additional test cases with the aim of increasing the test coverage.

What is coverage?

- $Coverage = \frac{Number\ of\ coverage\ items\ excercised}{Total\ number\ of\ coverage\ items} \times\ 100$

- A coverage item is whatever we have been able to count and see whether a test has exercised or used this item.
- NB! 100% coverage does not mean that 100% tested!

Structural Coverage Testing

- Idea
  - Code that has never been executed likely has bugs
    - At least the test suite is clearly not complete

- This leads to the notion of code coverage
  - Divide a program into elements (e.g., statements)
  - Define the coverage of a test suite to be

Control Flow Graphs: The One Slide Tutorial

X = 3; if

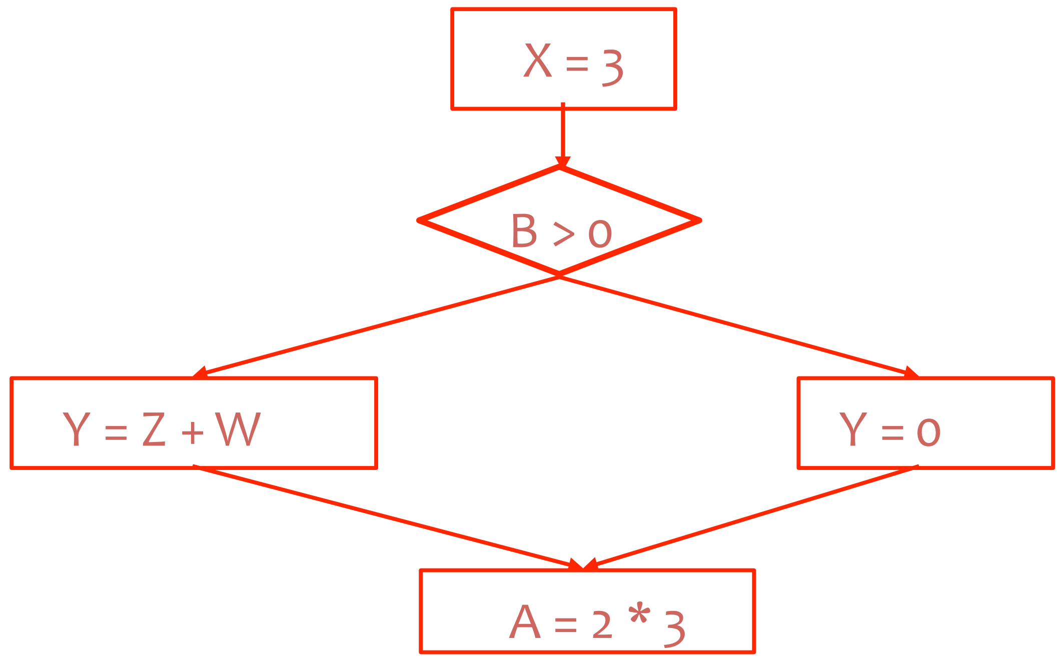
(B > 0)

    Y = 0;

else

    Y = Z + W; A

= 2 * 3;



- A graph
- Nodes are basic blocks
  - statements
- Edges are transfers of control between basic blocks

Test Adequacy Criterion

- Adequacy criterion of a test suite

  Whether a test suite satisfies some property deemed important to thoroughly test a program

- e.g.,
  - Cover all statements
  - Cover all branches

Control Flow Based Adequacy Criteria and Coverage

- Statement coverage
  - Cover every statement at least once
- Branch coverage, a.k.a. decision coverage
  - Cover every branch at least once
- (Basic) Condition coverage
  - Cover each outcome of every condition
- Branch-Condition coverage
  - Cover all conditions and all branches
- Modified condition decision coverage (MC/DC)
- Compound condition coverage
  - Cover all possible combinations of every condition

A Simple Example of Coverage

```
if ( a < b and c == 5) {
        y++;
}
x = 5;
```

* *and* is interpreted as logical-and

Test cases:

(a) a < b, c == 5

(b) a < b, c != 5

(c) a >= b, c == 5

(d) a >= b, c != 5

- Statement coverage:
  Test case (a)
- Branch coverage:
  Test cases (a) and (b)
- (Basic) Condition coverage:
  Test case (b) and (c)
  Problem: and (&&) short circuits!
  And second half of (c) not executed.
- Branch-Condition coverage:
  Test case (a) (b) and (c)
- Compound condition coverage:
  Test case (a) (b) (c) and (d)

Statement Testing

- Adequacy criterion:
  - each statement (or node in the CFG) must be executed at least once
- Coverage:

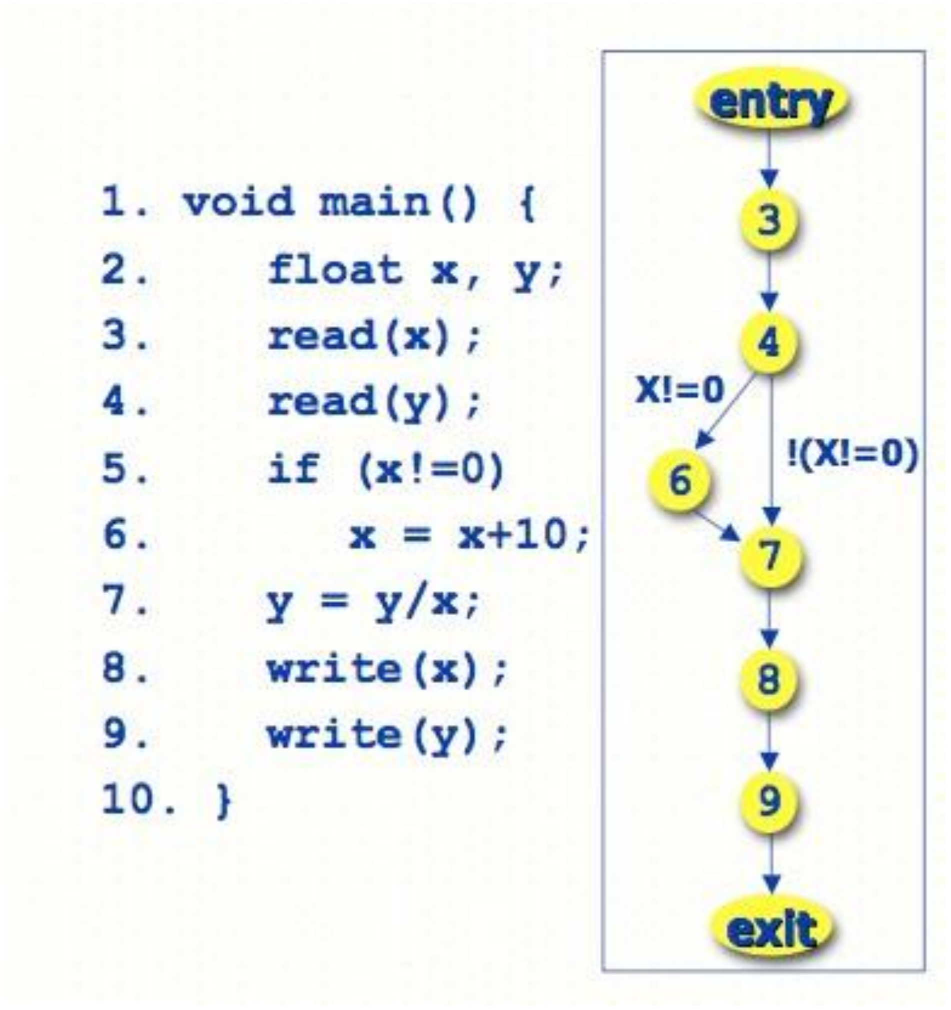
$$\frac{\text{\# executed statements}}{\text{\# statements}}$$

- Rationale:
  - A defect in a statement can only be revealed by executing the faulty statement

Statements or Blocks?

- Nodes in a CFG often represent basic blocks of multiple statements
  - *basic block* coverage or *node coverage*
  - difference in granularity, not in concept
- No essential difference
  - 100% node coverage ⇔ 100% statement coverage
    - but levels will differ below 100%
  - A test case that improves one *<u>will</u>* improve the other
    - though not by the same amount, in general

Statement Coverage: Example

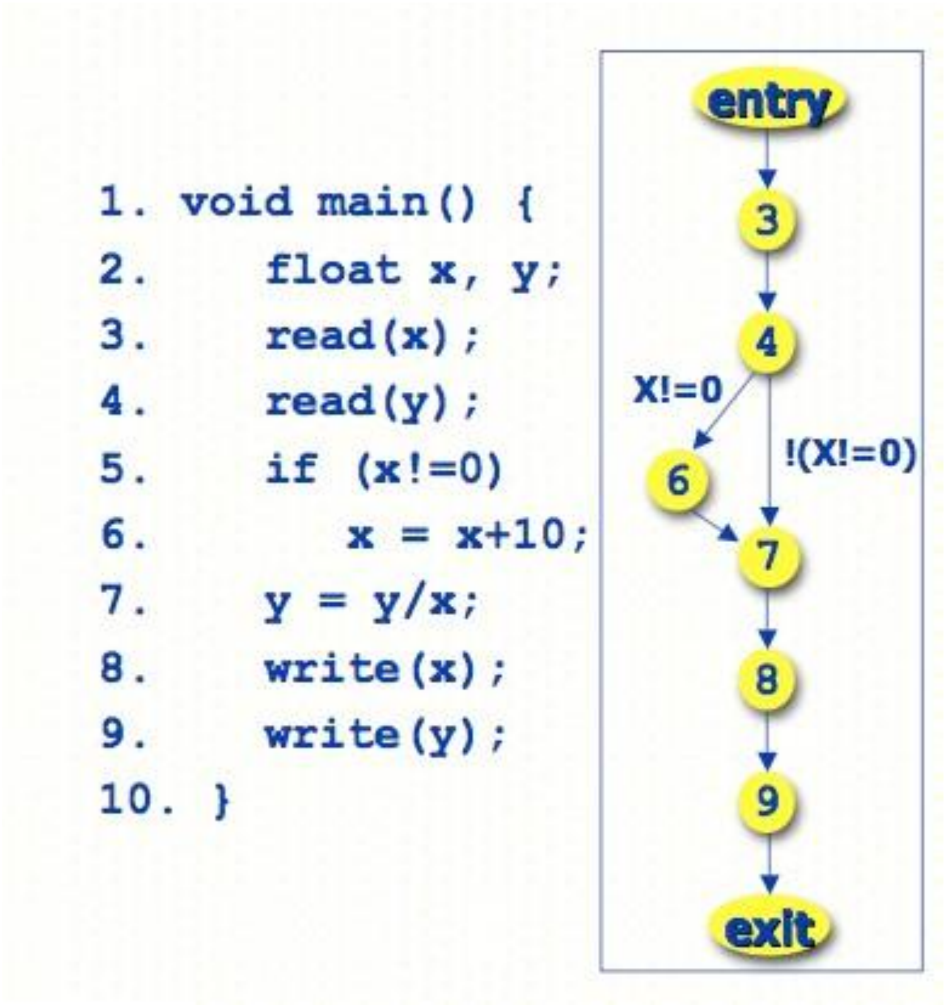


```
1. void main() {
2.      float x, y;
3.      read(x);
4.      read(y);
5.      if (x!=0)
6.          x = x+10;
7.      y = y/x;
8.      write(x);
9.      write(y);
10. }
```

- Test requirements
  - Nodes 3, … , 9
- Test cases
  - (x = 20, y = 30)

Any problems with this example?

Statement Coverage: Example



```
1. void main() {
2.      float x, y;
3.      read(x);
4.      read(y);
5.      if (x!=0)
6.          x = x+10;
7.      y = y/x;
8.      write(x);
9.      write(y);
10. }
```

- Test requirements
  - Nodes 3, … , 9
- Test cases
  - (x = 20, y = 30)

Such test does not reveal the fault at statement 7

To reveal it, we need to traverse edge 4-7

=> Branch Coverage

Statement Coverage in Practice

- Microsoft reports 80-90% statement coverage
- Boeing must get 100% statement coverage (feasible) for all software
- Usually can about 85% coverage; 100% is harder
  - Unreachable code; dead code
  - Complex sequences
  - Not enough resources

Branch Testing

- Adequacy criterion:
  - each branch (edge in the CFG) of every selection statement (if, switch) must be executed at least once
- Coverage:

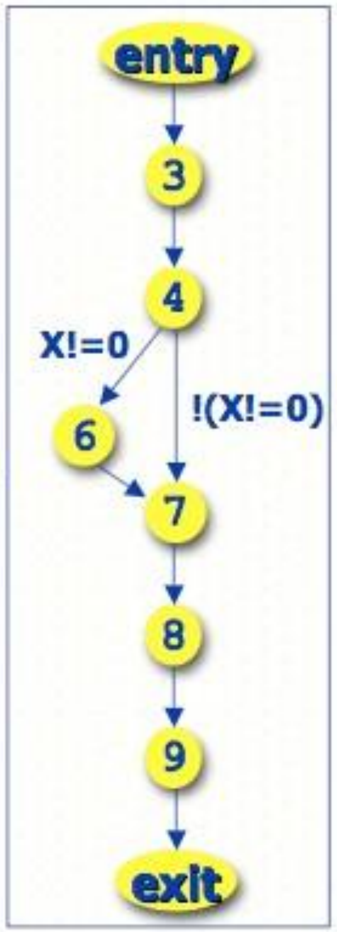
$$\frac{\#\ \text{executed branches}}{\#\ \text{branches}}$$

Statements vs. Branches

- Traversing all edges of a graph causes all nodes to be visited
  - Satisfying branch adequacy implying satisfying the statement adequacy
- The converse is not true
  - A statement-adequate (or node-adequate) test suite may not be branch-adequate (edge-adequate)

Branch Coverage: Example

```
1. void main() {
2.     float x, y;
3.     read(x);
4.     read(y);
5.     if (x!=0)
6.         x = x+10;
7.     y = y/x;
8.     write(x);
9.     write(y);
10. }
```



- Test requirements
  - Edges 4-6, Edges 4-7
- Test cases
  - (x = 20, y = 30)
  - (x = 0, y = 30)

Branch Coverage: Example

```
main(){
        int x, y, z, w;
        read(x);
        read(y);
        if (x != 0)
            z = x + 10;
        else
            z = 0;
        if (y>0)
            w = y / z;

        }
```

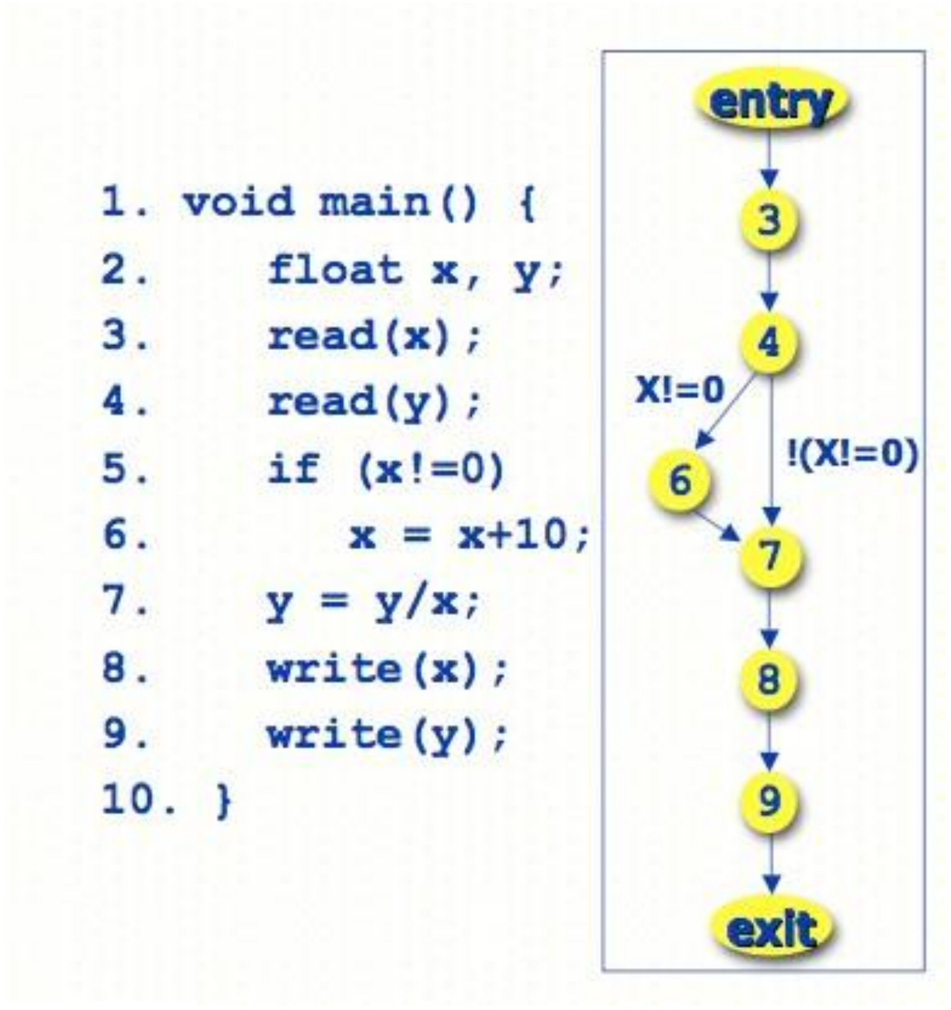
- Branch Coverage
- Test Cases
  - (x = 1, y = 22)
  - (x = 0, y = -10)
- Is the test suite adequate for branch coverage?

Branch Coverage: Example

```
1. main(){
2.        int x, y, z, w;
3.        read(x);
4.        read(y);
5.        if (x != 0)
6.              z = x + 10;
7.        else
8.              z = 0;
9.        if (y>0)
10.             w = y / z;
11.
12.       }
```

- Branch Coverage
- Test Cases
  - (x = 1, y = 22)
  - (x = 0, y = -10)
- Is the test suite adequate for branch coverage?
- Yes, but it does not reveal the fault at statement 10
- Test case (x = 0, y = 22)
  - Reveals fault

Branch Coverage: Example



```
1. void main() {
2.     float x, y;
3.     read(x);
4.     read(y);
5.     if (x!=0)
6.         x = x+10;
7.     y = y/x;
8.     write(x);
9.     write(y);
10. }
```

- Consider test cases
  - {(x=5,y=5), (x=5, y=-5)}
- The test suite is adequate for branch coverage, but does not reveal the fault at statement 6
- Predicate 4 can be true or false operating on only one condition

⇒ Basic condition coverage

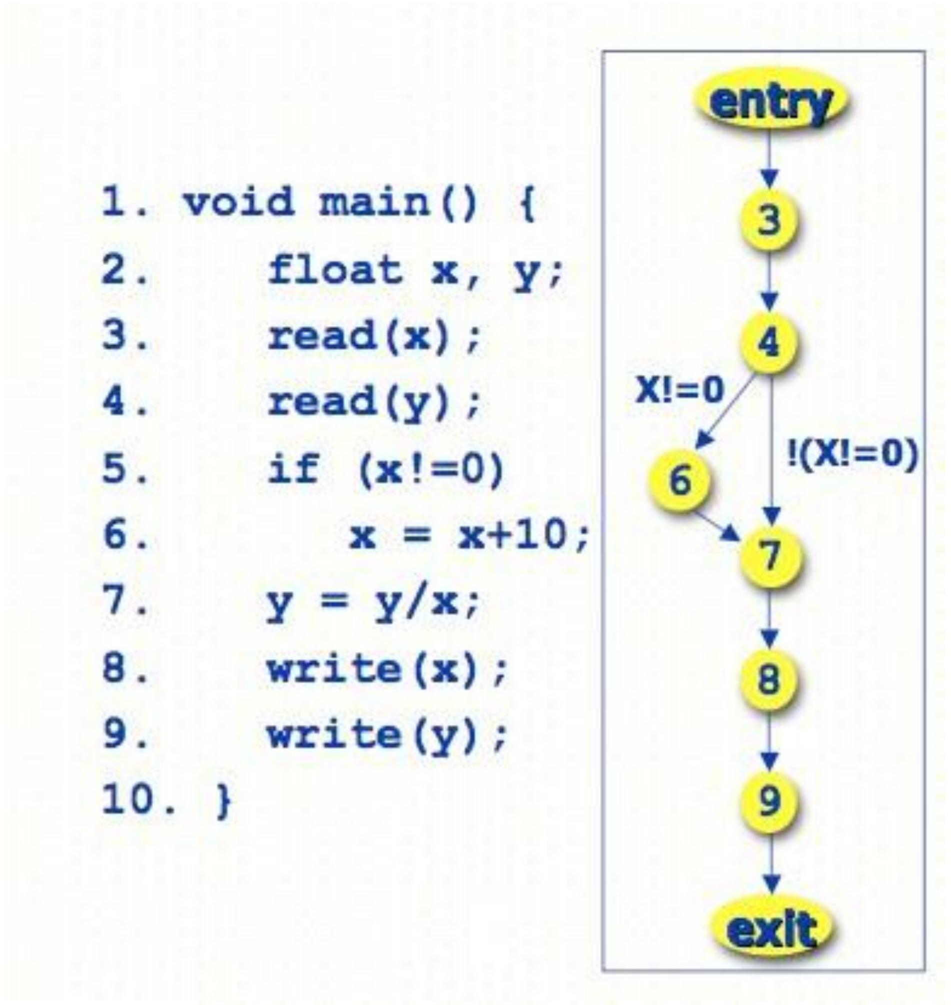
(Basic) Condition Testing

- Adequacy criterion:
  - Both outcomes (true and false) of each basic condition or predicate, must be tested at least once
  - Basic condition or predicate: a Boolean expression that does not contain other Boolean expression
- Coverage:

$$\frac{\text{\# truth values taken by all basic conditions}}{2 * \text{\# basic conditions}}$$

[See note]

Basic Condition Coverage: Example



```
1. void main() {
2.     float x, y;
3.     read(x);
4.     read(y);
5.     if (x!=0)
6.         x = x+10;
7.     y = y/x;
8.     write(x);
9.     write(y);
10. }
```

- Consider test cases
    - {(x=5,y=5), (x=5, y=-5)}
- The test suite is adequate for basic condition coverage, but it does not reveal the fault at statement 6
- The test suite is not adequate for branch coverage.

⇒Branch and condition coverage

Branch-Condition Testing

- Branch and condition adequacy
- Cover all conditions and all branches
  - Both outcomes (true and false) of each basic condition must be tested at least once.
  - All branches of every selection statement must be executed at least once .

- Notice that due to the left-to-right evaluation order and short-circuit evaluation of logical OR expressions, the value true for the first condition does not need to be combined with both values false and true for the second condition.

Compound Condition Testing

- Compound (multiple) condition adequacy:
  - Cover all possible combinations of compound conditions and cover all branches of a selection statement
  - For a compound condition with

    n basic conditions,

    2^n test cases may be needed.

C = C1 and C2

	C1	C2	C
T1	T	T	T
T2	T	F	F
T3	F	T	F
T4	F	F	F

Compound Conditions: Exponential Complexity

```
(((a || b) && c) || d) && e
```

Test Case	a	b	c	d	e
(1)	T	—	T	—	T
(2)	F	T	T	—	T
(3)	T	—	F	T	T
(4)	F	T	F	T	T
(5)	F	F	—	T	T
(6)	T	—	T	—	F
(7)	F	T	T	—	F
(8)	T	—	F	T	F
(9)	F	T	F	T	F
(10)	F	F	—	T	F
(11)	T	—	F	F	—
(12)	F	T	F	F	—
(13)	F	F	—	F	—

short-circuit evaluation often reduces this to a more manageable number, but not always

Modified Condition/Decision Coverage (MC/DC)

- Motivation:
  - Effectively test *important combinations* of conditions, without exponential blowup in test suite size
  - "Important" combinations means:
    - each basic condition shown to independently affect the outcome of each decision
- Requires: For each basic condition C, two test cases,
  - C evaluates to *true* for one and *false* for the other
  - Values of all other *evaluated* conditions remain the same
  - The compound condition as a whole evaluates to *true* for one and *false* for the other

Construct Test Cases for MC/DC

- MC/DC with two basic conditions
  - $C = C_1 \&\& C_2$

	C1	C2	C
T1	T	T	T
T2	T	F	F
T3	F	T	F

  - $C = C_1 \|\| C_2$

	C1	C2	C
T1	T	F	T
T2	F	T	T
T3	F	F	F

Construct Test Cases for MC/DC

- MC/DC with three basic conditions

$C = (C_1 \&\& C_2) \&\& C_3$

1. Copy rows T1, T2, T3 in the (C1 && C2) table to the table below and
2. Fill in true for column C3

	C1	C2	C3	C
T1	T	T	T	T
T2	T	F	T	F
T3	F	T	T	F
T4				

- Operator || can be handled similarly (symmetric)

Construct Test Cases for MC/DC

- MC/DC with three basic conditions

$C = (C_1 \&\& C_2) \&\& C_3$

3. Add a new row for T4
    - Column C3: false
    - Column C1 and C2: copy the values from one of T1, T2, or T3 with true outcome

	C1	C2	C3	C
T1	T	T	T	T
T2	T	F	T	F
T3	F	T	T	F
T4	T	T	F	F

MC/DC: Linear Complexity

- Only n+1 test cases needed for n basic conditions
- Adopted by many industry quality standards

```
(((a || b) && c) || d) && e
```

Test Case	a	b	c	d	e	Outcome
(1)	true	--	true	--	true	true
(2)	false	true	true	--	true	true
(3)	true	--	false	true	true	true
(6)	true	--	true	--	false	false
(11)	true	--	false	false	--	false
(13)	false	false	--	false	--	false

Values in red independently affect the output of the decision

Analysis of MC/DC

- MC/DC is
  - basic condition coverage (C)
  - decision (branch) coverage (DC)
  - plus one additional condition (M):
    every condition must *independently affect* the decision's outcome
- Subsumed by compound conditions
- Subsumes all other criteria discussed so far
  - stronger than statement and branch coverage
- A good balance of thoroughness and test size
  - therefore widely used

Leave (Almost) No Code Untested

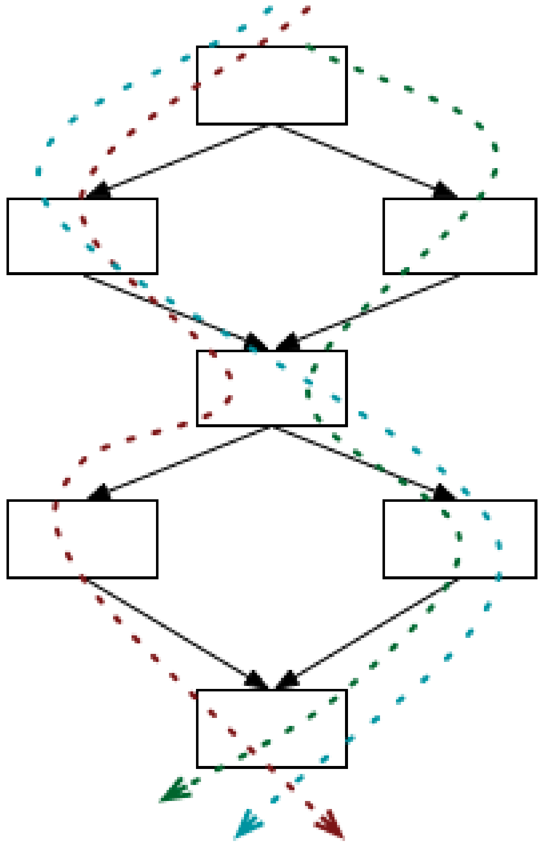
- In general
  - 90% coverage is achievable
  - 95% coverage require significant effort
  - 100% not always attainable
- Challenges in coverage
  - Platform specific code
  - Defensive programming
  - Exception handling
  - Non-public method, never invoked

Summary

- Rationale for structural testing
- Basic terms: adequacy, coverage
- Characteristics of common structural criteria
  - Statement, branch, condition, compound condition, MC/DC
- Practical uses and limitations of structural testing

Path Testing

Paths Testing



- Beyond individual branches.
- Should we explore sequences of branches (paths) in the control flow?
- Many more paths than branches
  - A pragmatic compromise will be needed

Path Coverage

- Path coverage means that all possible execution paths in the program must be executed.
- This is a very strong coverage criterion, but impractical.
- Programs with loops have an infinite number of execution paths, and thus would need an infinite number of test cases
- The fact that there is complete branch coverage does not mean that all errors will be found. Branch testing does not necessarily check all combinations of control transfers.

Path Adequacy

- Decision and condition adequacy
    - Test individual program decisions
- Paths testing
    - Test combinations of decisions along paths
- Adequacy criterion:
    - Each path must be executed at least once
- Coverage:

$$\frac{\#\ \text{executed paths}}{\#\ \text{total paths}}$$

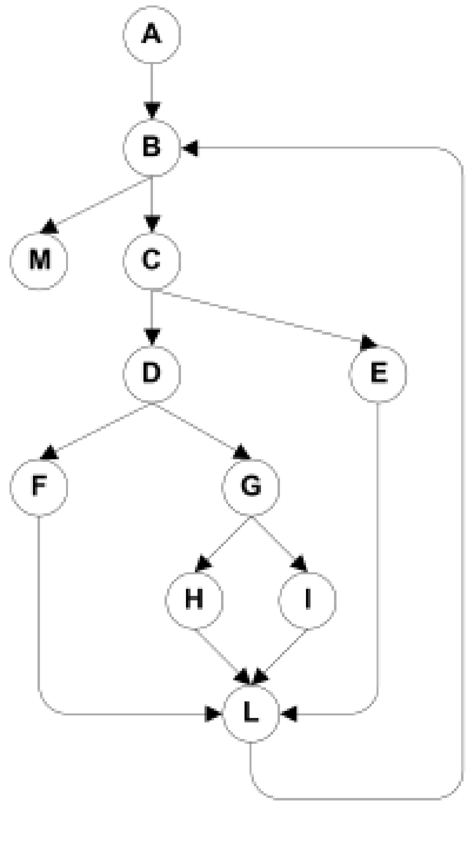
Practical Path Coverage Criteria

- The number of paths in a program with loops is unbounded
    - the simple criterion is impossible to satisfy
- For a feasible criterion:
    - Partition infinite set of paths into a finite number of classes
- Useful criteria can be obtained by limiting
    - the number of traversals of loops
    - the length of the paths to be traversed
    - the dependencies among selected paths

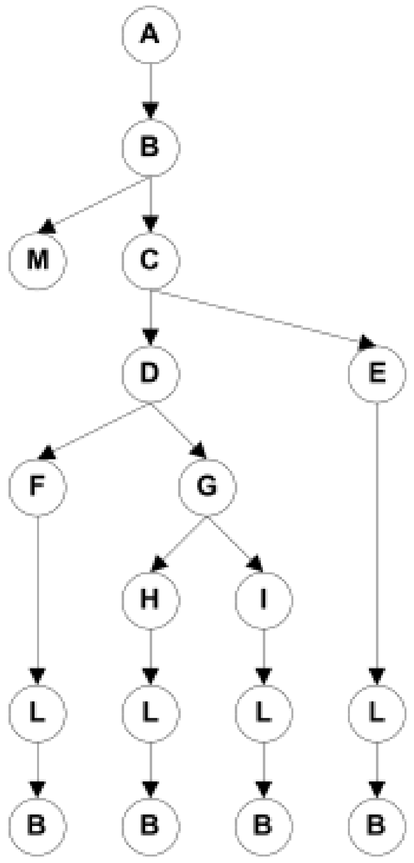
Boundary Interior Path Testing

- Group together paths that differ only in the sub-path they follow when repeating the body of a loop
- Follow each path in the control flow graph up to the first repeated node
- The set of paths from the root of the tree to each leaf is the required set of sub-paths for boundary/interior coverage

Boundary Interior Adequacy for cgi-decode



(i)



(ii)

Limitations of Boundary Interior Adequacy

- The number of paths can still grow exponentially

```
if (a) {
   S1;
}
if (b) {
   S2;
}
if (c) {
   S3;
}
...
if (x) {
   Sn;
}
```

- The sub-paths through this control flow can include or exclude each of the statements Si
- Total N branches result in 2^N paths that must be traversed
- Choosing input data to force execution of one particular path may be very difficult
  - even impossible if the conditions are not independent

Loop Boundary Adequacy

- A test suite satisfies the loop boundary adequacy criterion **iff** for every loop:
    - At least one test case: the loop body is iterated zero times
    - At least one test case: the loop body is iterated once
    - At least one test case: the loop body is iterated more than once

What is LCSAJ Testing ?

- LCSAJ stands for Linear Code Sequence and Jump, a white box testing technique to identify the code coverage, which begins at the start of the program or branch and ends at the end of the program or the branch.
- LCSAJ consists of testing and is equivalent to statement coverage.
- LCSAJ Characteristics:
    - 100% LCSAJ means 100% Statement Coverage
    - 100% LCSAJ means 100% Branch Coverage
    - 100% procedure or Function call Coverage
    - 100% Multiple condition Coverage

Linear Code Sequence and Jump (LCSAJ)

- Execution of sequential programs that contain at least one condition, proceeds in pairs where the first element of the pair is a sequence of statements, executed one after the other, and terminated by a jump to the next such pair.
- A Linear Code Sequence and Jump is a program unit comprised of a textual code sequence that terminates in a jump to the beginning of another code sequence and jump.
- An LCSAJ is represented as a triple (X, Y, Z) where X and Y are, respectively, locations of the first and the last statements and Z is the location to which the statement at Y jumps.

Linear Code Sequence and Jump (LCSAJ)

- Consider this program.

```
1    begin
2      int x, y, p;
3      input (x, y);
4      if(x<0)
5        p=g(y);
6      else
7        p=g(y*y);
8    end
```

LCSAJ	Start Line	End Line	Jump to
1	1	6	exit
2	1	4	7
3	7	8	exit

The last statement in an LCSAJ (X, Y, Z) is a jump and Z may be program exit. When control arrives at statement X, follows through to statement Y, and then jumps to statement Z, we say that the LCSAJ (X, Y, Z) is traversed or covered or exercised.

LCSAJ coverage: Example 1

```
1   begin
2     int x, y, p;
3     input (x, y);
4     if(x<0)
5       p=g(y);
6     else
7       p=g(y*y);
8   end
```

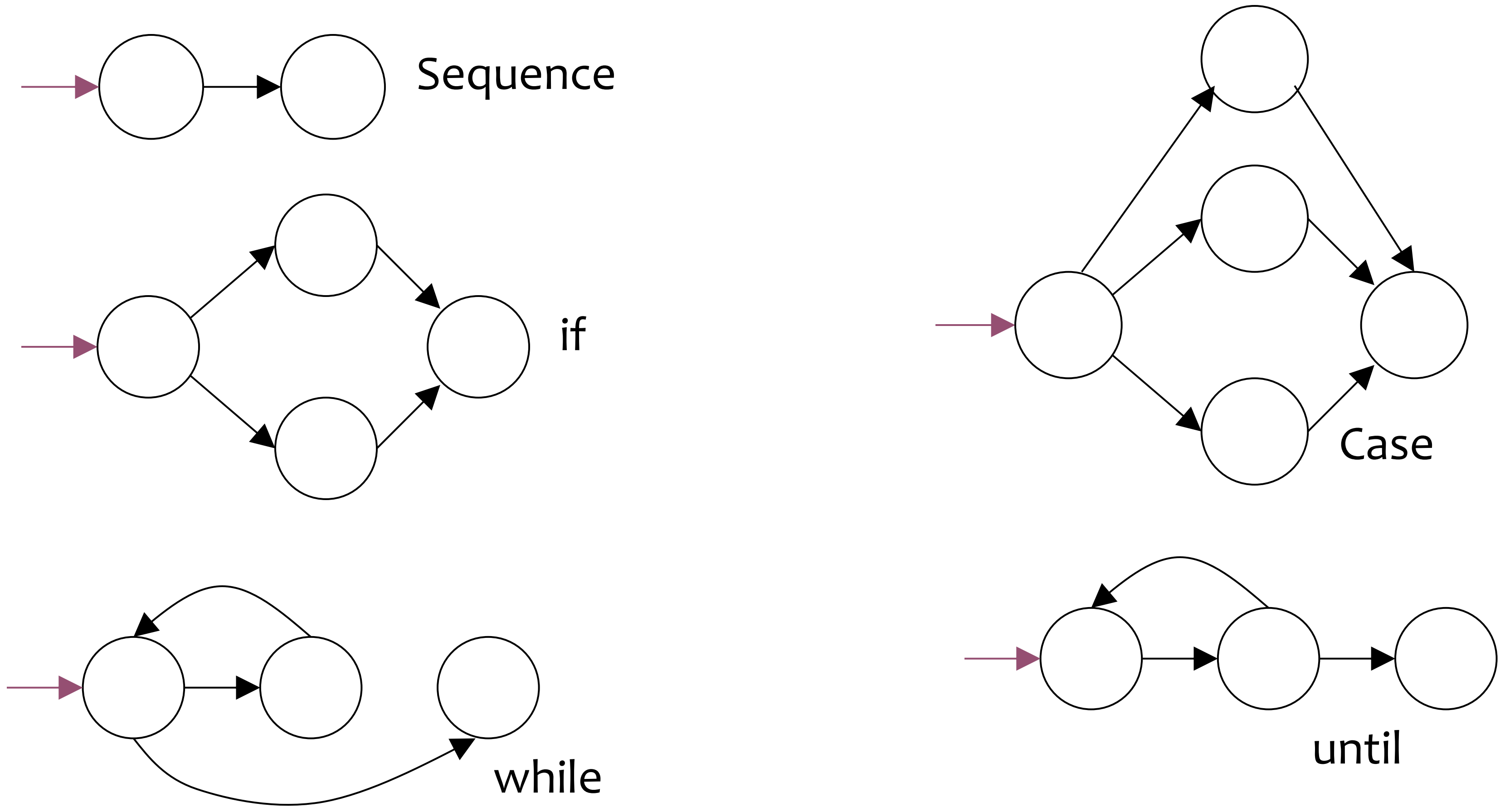
$$T = \left\{ \begin{array}{lll} t_1 : & < \mathrm{x} = -5 & \mathrm{y} = 2 > \\ t_2 : & < \mathrm{x} = 9 & \mathrm{y} = 2 > \end{array} \right\}$$

t2 covers (1,4,7) and (7, 8, exit).
t1 covers (1, 6, exit).
T covers all three LCSAJs.

Basis Path Testing

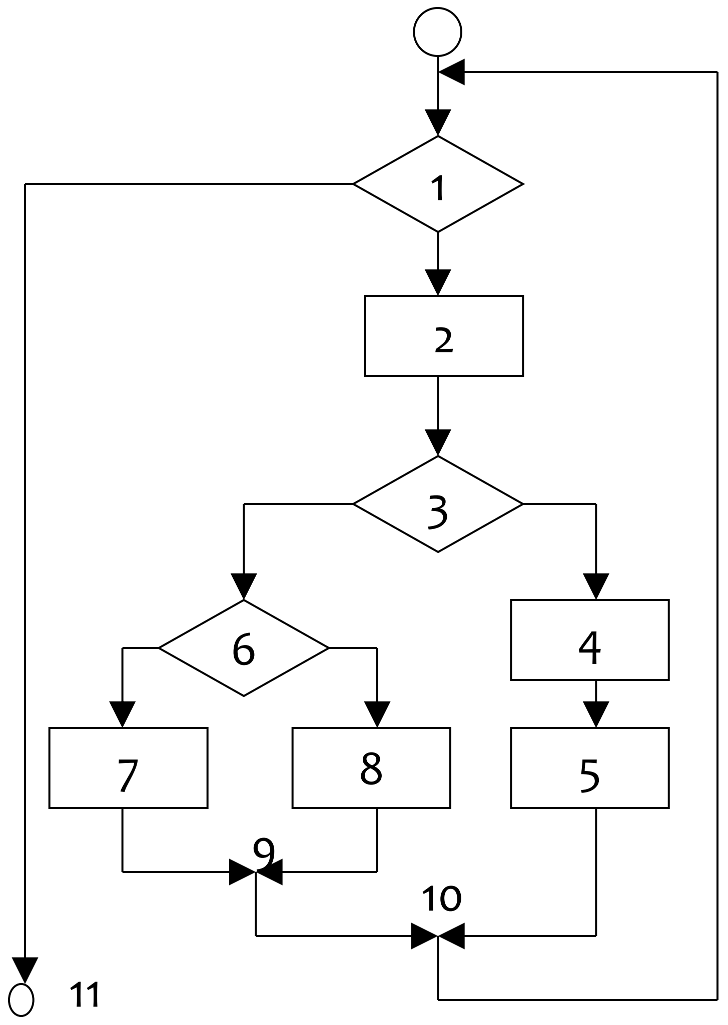
- First proposed by Tom McCabe in 1976. Later expanded it into Cyclomatic Complexity.
- Enables the test case designer to derive a logical complexity measure of the procedural design.
- Uses this measure as the basis for defining an upper bound on the number of execution paths needed to guarantee that every statement in the program is executed at least once.
- Uses a notation known as a flow graph.
  - Each structured notation has a corresponding flow graph symbol.

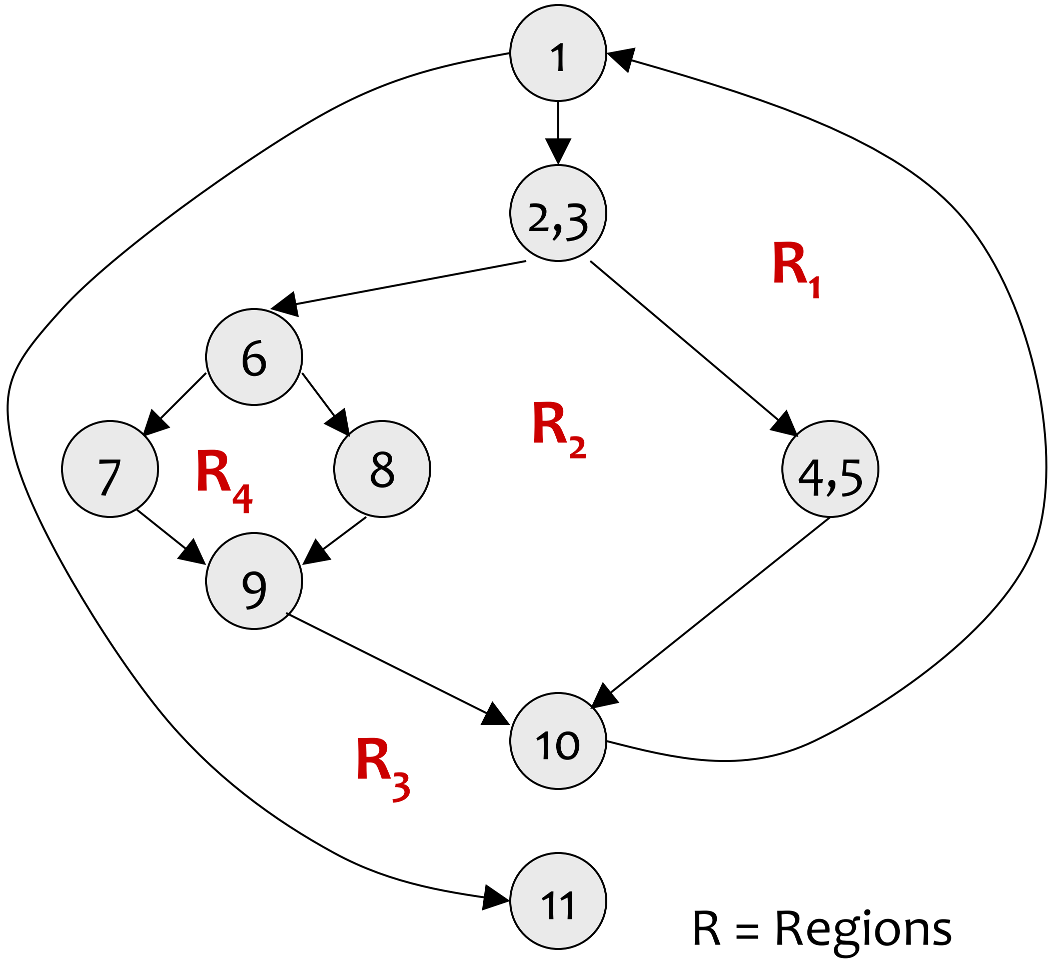
Flow Graph Notation



Where each circle represents **one or more** nonbranching set of source code statements.

Flow chart and corresponding flow graph



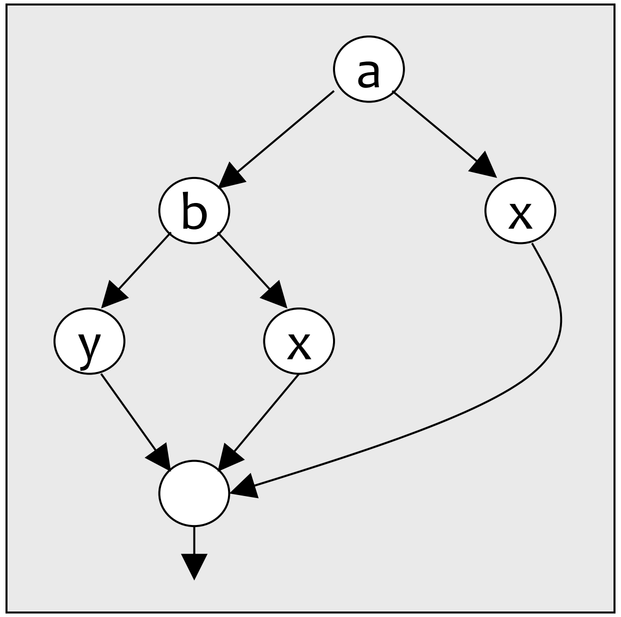


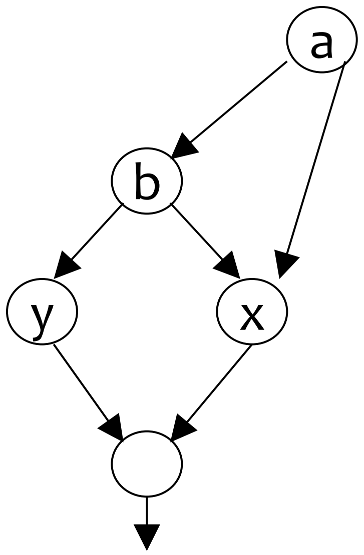
R = Regions

Compound logic

- A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement.
- Example:

```
if a OR b
then do X
else do Y
endif
```

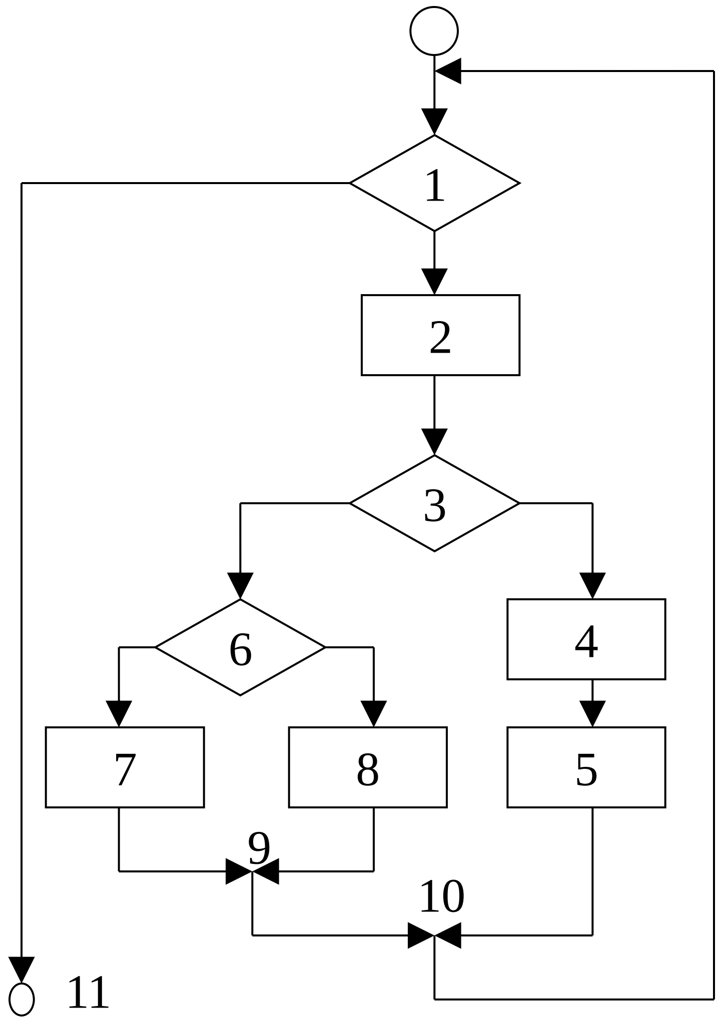




Independent Program Paths

- Any path through the program that introduces at least one new set of processing statements or a new condition.
- In terms of a flow graph, an independent path must move along at least one edge that has not previously been traversed.

Basis Paths Example



- Basis paths:

  Path 1: 1-11

  Path 2: 1-2-3-4-5-10-1-11

  Path 3: 1-2-3-6-8-9-10-1-11

  Path 4: 1-2-3-6-7-9-10-1-11

- The path below is **NOT** a basis path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 because it does not traverse any new edges.

Choose a Set of Basis Paths

- Traverse the CFG to identify basis paths
  - Each path contains at least one edge that is not in other paths.
  - No iteration of sub-paths
- There is more than one set of basis paths for a given CFG.

Basis Paths

- These paths constitute a basis set for the flow graph.
- Design tests to execute these paths.
- Guarantees:
  - Every statement has been executed at least once.
  - Every condition has been executed on both its true and false sides.
- **There is more than ONE correct set of basis paths for a given problem.**
- How many paths should we look for?
  - Calculate **Cyclomatic complexity** V(G)
    - V(G) = E-N+2
    - V(G) = P + 1 (Where P = number of predicate nodes)
    - V(G) = R (Where R = number of regions)

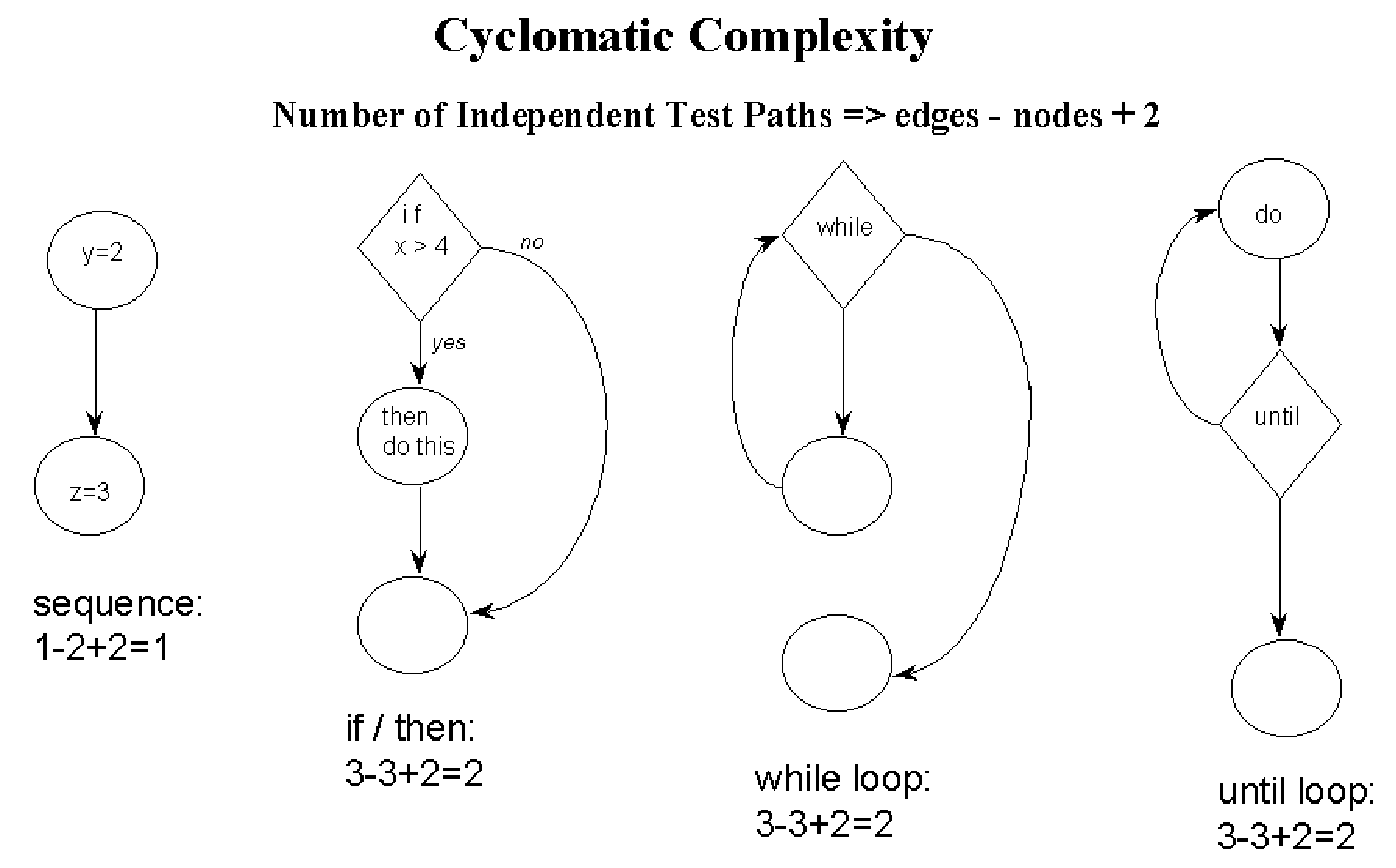
Cyclomatic Complexity (CC)

- Evaluates the complexity of an algorithm in a method. It measures the number of linearly independent paths through a program's source code
- Three equivalent definitions
  - V(G) = E-N+2
    - Or given a flow graph: = edges – nodes + 2
  - V(G) = P + 1 (Where P = number of predicate nodes)
    - Defined to be one larger than the number of decision points (if/case-statements, while-statements, etc.) in a module (function, procedure, chart node, etc.).
      - Note that in an **if** or **while** statement, a complex boolean counts each part, (e.g. **if(a<b and c>0)** counts as two decision points.
  - V(G) = R (Where R = number of regions)

Cyclomatic Complexity (CC)

- Calculate the cyclomatic complexity.
- A method with a low cyclomatic complexity is generally better.
- This may imply decreased testing and increased understandability or that decisions are deferred through message passing, not that the method is not complex

Cyclomatic Complexity (CC)



CC = edges – nodes + 2

Cyclomatic Testing

- A compromise between path testing and branch-condition testing
- Cyclomatic testing, a.k.a., basis path testing

  Test all of the independent paths that could be used to construct any arbitrary path through the computer program
- Steps:

  1. Calculate Cyclomatic complexity
  2. Choose a set of basis paths
  3. Design test cases to exercise each basis path

Cyclomatic Complexity (McCabe)

- A set of *basis paths* in a CFG
  - Each path contains at least one edge that is not in other paths
  - No iteration of sub-paths
- The number of basis paths in a CFG is known as the *Cyclomatic Complexity* of the CFG
- Calculate Cyclomatic complexity of CFG G
  - e = #edges in G
  - n = #nodes in G
- The cyclomatic complexity of G

$$V(G) = e - n + 2$$

Cyclomatic Complexity (McCabe)

- If a CFG has a single entry and single exit point,
  the calculation of the cyclomatic complexity can be simplified

  V(G) = #predicates + 1

- Rules for counting predicates in CFG
  - Condition in a if-statement: count each predicate
  - Condition in a loop statement count as 1, even if it's a compound condition
  - Switch statement: *n*-way choice count as (n -1)

Cyclomatic Complexity Examples

```
statement₁
statement₂
…
statementₙ
```

$\boxed{V(G) = 1}$

```
if (x > 5) {
 statement₁
}
```

$\boxed{V(G) = 2}$

```
if (x > 5) {
 statement₁
} else {
 statement₂
}
```

$\boxed{V(G) = 2}$

Cyclomatic Complexity Examples

```
switch (exp) {
case v₁: statement₁; break;
case v₂: statement₂; break;
…
case vₙ: statementₙ; break;
default: statementₙ₊₁;
}
```

$\boxed{V(G) = n + 1}$

```
if (x > 5 || x <= 10) {
  statement₁
}
```

$\boxed{V(G) = 3}$

```
if (x >= 0 && x <= 100) {
  statement₁
} else if (x <= 30) {
  statement₁
} else {
  statement₂
}
```

$\boxed{V(G) = 4}$

Cyclomatic Complexity Examples

```
while (i < 100) {
  statement1;
}
```

$V(G) = 2$

```
for (i = 0; i < 10; i++) {
  statement1
}
```

$V(G) = 2$

```
while (i < 100 && a != null) {
  statement1;
}
```

$V(G) = 2$

```
for (i = 0; i <= 10 || j == 0;
        i++) {
  statement1;
  if (i > 5 || i <= 10) {
    statement2
  }
}
```

$V(G) = 4$

Cyclomatic Adequacy and Coverage

- Cyclomatic adequacy criterion
  - Each basis path must be executed at least once
  - Guarantees:
    - Every statement has been executed at least once.
    - Every condition has been executed on both its true and false sides.
  - Recommended by **NIST** as a baseline technique
- Cyclomatic coverage
  - the number of basis paths that have been executed, relative to cyclomatic complexity

$$\frac{\text{\# executed basis paths}}{\text{cyclomatic complexity}}$$

An Example:

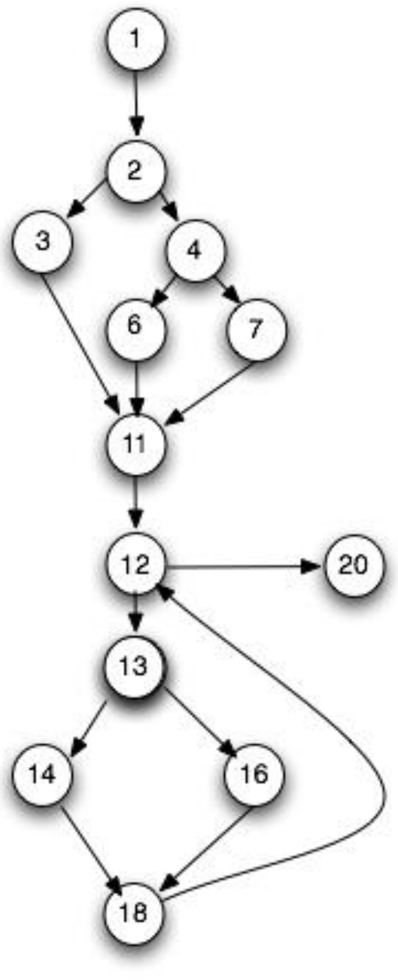
```
Float Function CalculateFees ( String mgroup, Integer mAge, Float mBaseFees, Integer
  mFamilyCount, Integer mMinFee)
/* How many months left in the year */
1  MonthsLeft = 12 - GetMonth(SystemDate())
/* Calculate base rate for the group */
2 if mgroup == 1 then
3      mRate = mBaseFees
4 else
5    if mgroup == 2 then
6      mRate = mBaseFees * 0.80
7   else
8     mRate = mBaseFees * 0.65
9   endif
10     endif
11     mBaseFees = mRate * MonthsLeft
12     while mFamilyCount > 1 and mBaseFees > mMinFee
13       if mAge >= 21
14     mBaseFees = mBaseFees - 10
15       else
16         mBaseFees = mBaseFees - 5
17   endif
18       mFamilyCount = mFamilyCount - 1
19     endwhile
20     return mBaseFees
```

Steps for deriving test cases

- Determine a basis set of linearly independent paths.

  1-2-3-11-12-20
  1-2-4-6-11-12-20
  1-2-4-7-11-12-20
  1-2-3-11-12-13-14-18-12-20
  1-2-4-6-11-12-13-14-18-12-20
  1-2-4-7-11-12-13-14-18-12-20
  1-2-3-11-12-13-16-18-12-20
  1-2-4-6-11-12-13-16-18-12-20
  1-2-4-7-11-12-13-16-18-12-20

- Prepare test cases that will force execution of each path in the basis set.



Infeasible Paths

Some paths are infeasible…

```
     begin
1.        readln (a);
2.        if a > 15
          then
3.                  b:=b+1;
          else
4.                  c:=c+1;
5.        if a < 10
          then
6.                  d:=d+1;
7.   end
```

V(G) = 3:
There are three basis paths:

Path 1: 1,2,3,5,7
Path 2: 1,2,4,5,7
Path 3: 1,2,3,5,6,7

Which of these paths is non-executable and why?

The Subsumes Relationship

- The subsumption relationship means that satisfying one test coverage criterion may implicitly force another test coverage criterion to be satisfied as well
  - E.g.: Branch coverage forces statement coverage to be attained as well (This is strict subsumption)
- "Subsumes" does not necessarily mean "better"

The Subsumes Relationship

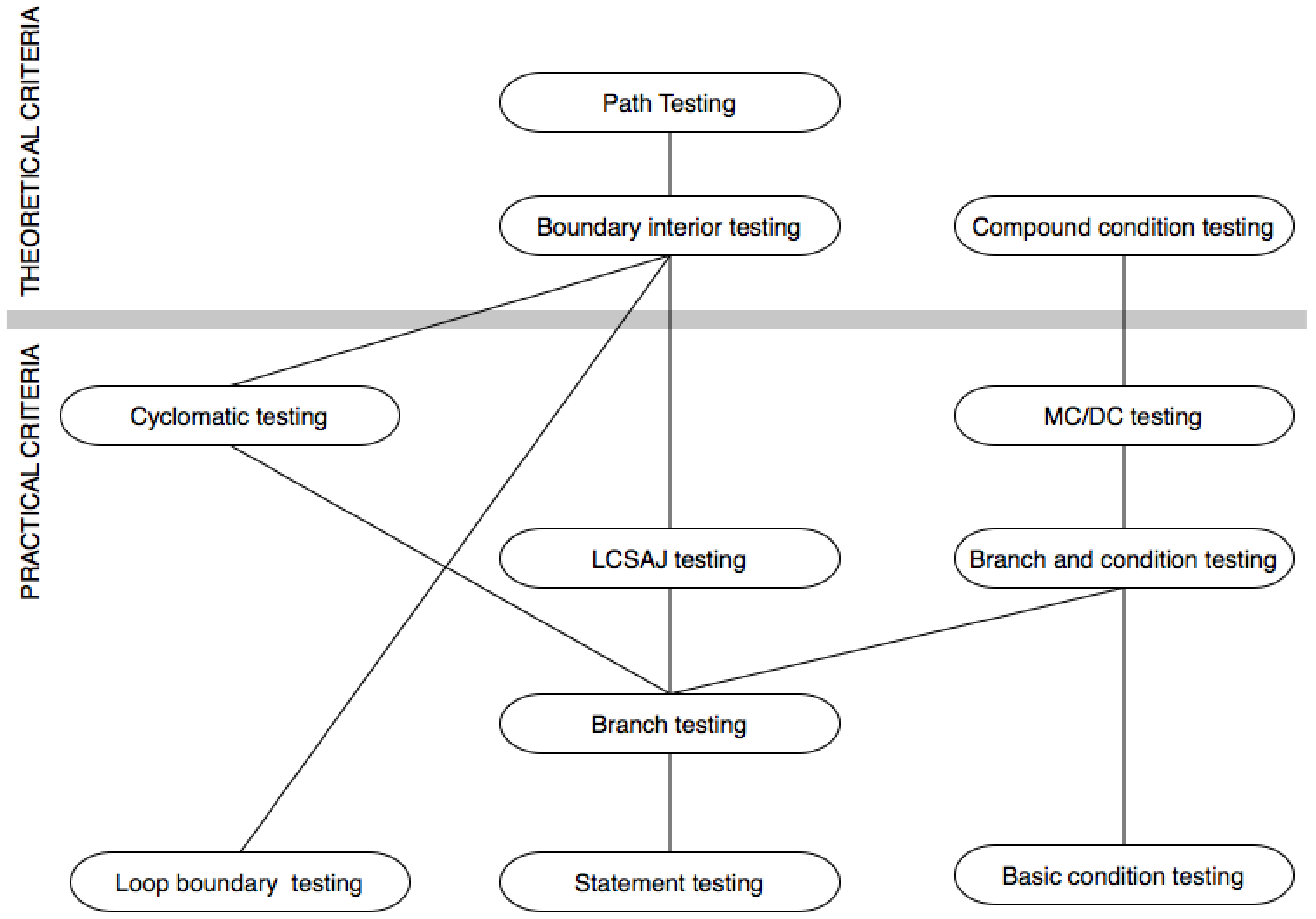
Test adequacy criterion A subsumes test adequacy criterion B iff, for every program P, every test suite satisfying A with respect to P also satisfies B with respect to P.

- Example:

  Exercising all program branches (branch coverage) *subsumes* exercising all program statements

- A common analytical comparison of closely related criteria
  - Useful for working from easier to harder levels of coverage, but not a direct indication of quality

The Subsumes Relation among Structural Testing Criteria



Satisfying Structural Criteria

- Sometimes criteria may not be satisfiable
- The criterion requires execution of
  - statements that cannot be executed as a result of
    - defensive programming
    - code reuse (reusing code that is more general than strictly required for the application)
  - conditions that cannot be satisfied as a result of
    - interdependent conditions
  - paths that cannot be executed as a result of
    - interdependent decisions

Satisfying Structural Criteria

- Large amounts of *fossil* code may indicate serious maintainability problems
  - But some unreachable code is common even in well-designed, well-maintained systems
- Solutions:
  - make allowances by setting a coverage goal less than 100%
  - require justification of elements left uncovered
    - RTCA-DO-178B and EUROCAE ED-12B for modified MC/DC

Java Code Coverage Tools

- Emma
- Clover
- Cobertura
- JaCoCo
- JCov
- Serenity

- See: https://en.wikipedia.org/wiki/Java_Code_Coverage_Tools

Summary

- Basis path testing should be applied to critical modules
- Adequacy criteria and coverage
    - statement
    - branch
    - condition, branch-condition, compound condition
    - MC/DC
    - paths
    - boundary/interior
    - loop boundary
    - LCSAJ
    - Cyclomatic, basis path
- Full coverage is usually unattainable