Path Testing and Test Coverage

Chapter 9

Structural Testing

- Also known as glass/white/open box testing
- Structural testing is based on using specific knowledge of the program source text to define test cases
  - **Contrast with functional testing where the program text is not seen but only hypothesized**

Structural Testing

- Structural testing methods are amenable to
  - **Rigorous definitions**
    - **Control flow**
    - **Data flow**
    - **Coverage criteria**
  - **Mathematical analysis**
    - **Graphs**
    - **Path analysis**
  - **Precise measurement**
    - **Metrics**
    - **Coverage analysis**

Program Graph Definition

- **What is a program graph?**

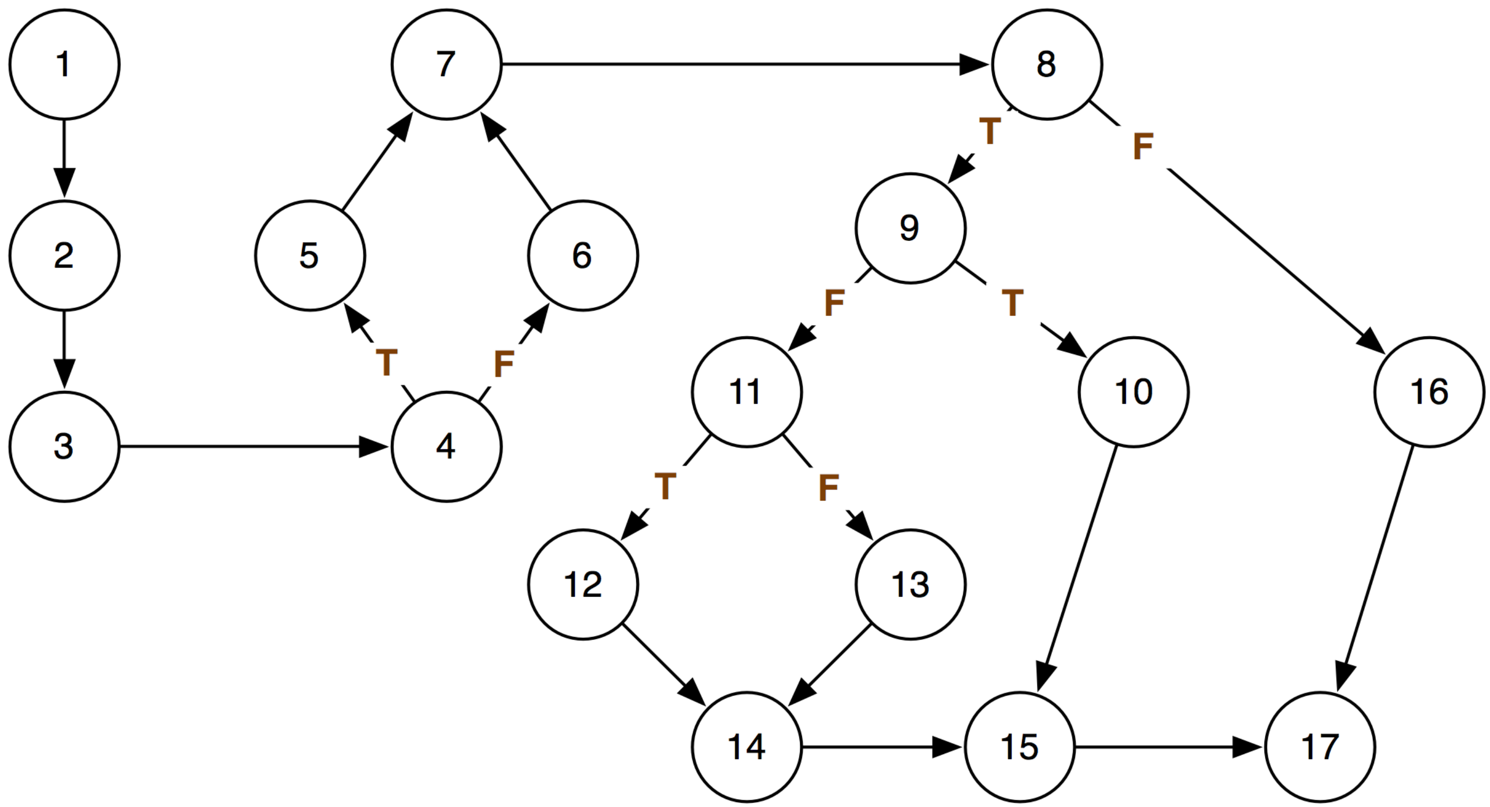
Program Graph Definition – 2

- Given a program written in an imperative programming language
  - **Its program graph is a directed graph**
    - **Nodes are statements and statement fragments**
    - **Edges represent flow of control**
  - **Two nodes are connected**
    - **If execution can proceed from one to the other**

Triangle program text

```
output ("Enter 3 integers")
input (a, b, c)
output("Side a b c: ", a, b, c)
if (a < b + c) and (b < a+c) and (c < a+b)
then isTriangle ← true
else isTriangle ← false
fi
if isTriangle
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")
            fi
      fi
 else output ("not a triangle")
 fi
```

Triangle Program Program Graph

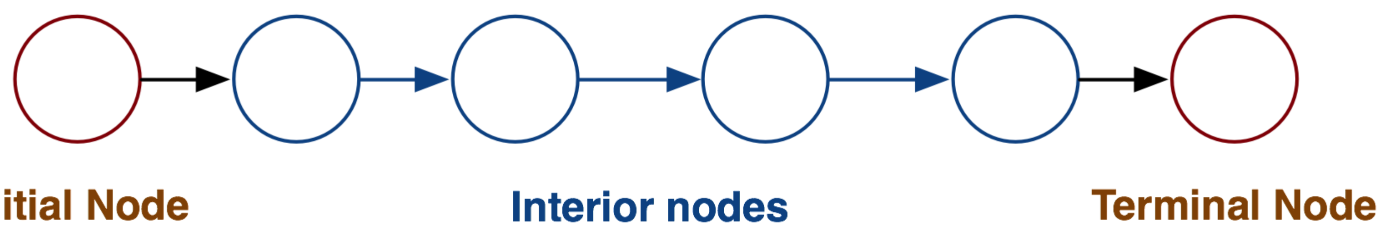


DD-Path

- **What is a DD-path?**

DD-Path – informal definition

- A **decision-to-decision** path (DD-Path) is a path chain in a program graph such that
  - **Initial and terminal nodes are distinct**
  - **Every interior node has indeg =1 and outdeg = 1**
    - **The initial node is 2-connected to every other node in the path**
    - **No instances of 1- or 3-connected nodes occur**

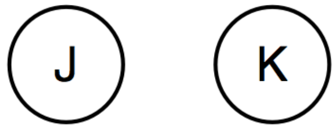


Connectedness definition

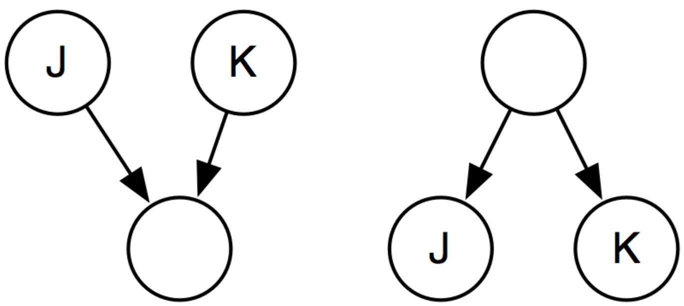
- **What is the definition of node connectedness?**
  - **Hint: There are 4-types of connectedness**

Connectedness definition – 2

- Two nodes J and K in a directed graph are
  - **0-connected iff no path exists between them**

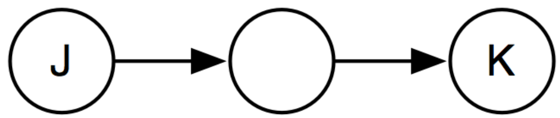


  - **1-connected iff a semi-path but no path exists between them**

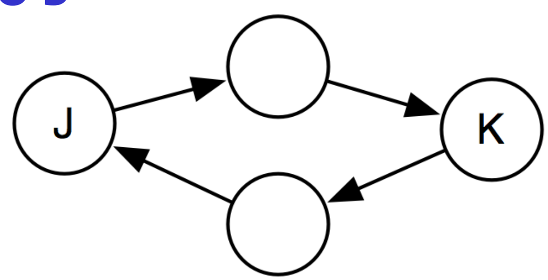


Connectedness definition – 2

- Two nodes J and K in a directed graph are
  - **2-connected iff a path exists between between them**



  - **3-connected iff a path goes from J to K , and a path goes from K to J**



DD-Path – formal definition

- A **decision-to-decision** path (DD-Path) is a chain in a program graph such that:
  - **Case 1: consists of a single node with indeg=0**
  - **Case 2: consists of a single node with outdeg=0**
  - **Case 3: consists of a single node with indeg ≥ 2 or outdeg ≥ 2**
  - **Case 4: consists of a single node with indeg =1, and outdeg = 1**
  - **Case 5: it is a maximal 2-connected chain of length ≥ 1**
- DD-Paths are also known as **segments**

Triangle program DD-paths

Nodes	Path	Case
1	First	1
2,3	A	5
4	B	3
5	C	4
6	D	4
7	E	3
8	F	3
9	G	3

Nodes	Path	Case
10	H	4
11	I	3
12	J	4
13	K	4
14	L	3
15	M	3
16	N	4
17	Last	2

DD-path Graph

- **What is a DD-path graph?**

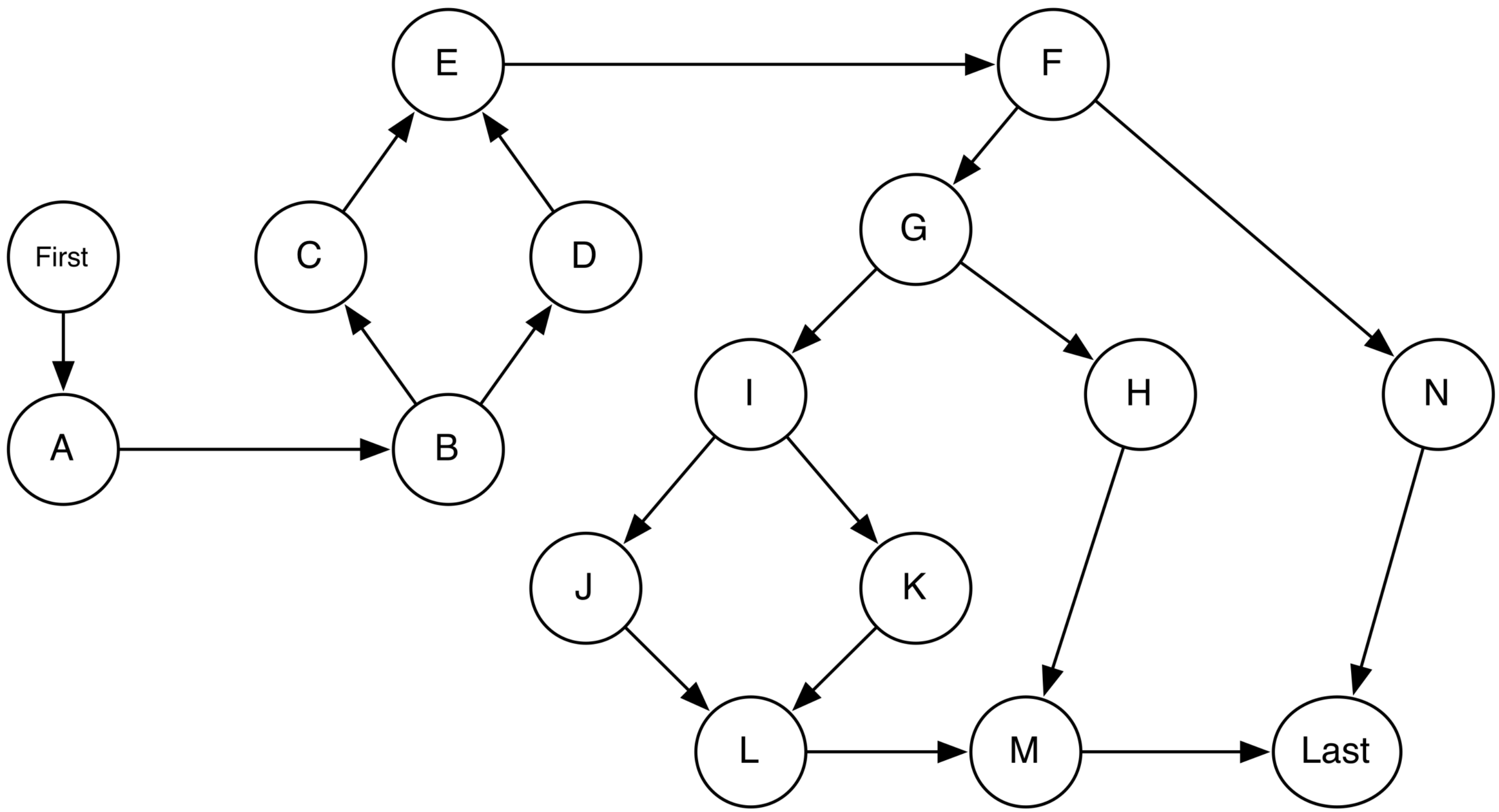
DD-Path Graph – informal definition

- Given a program written in an imperative language, its **DD-Path graph** is a directed graph, in which
  - **Nodes are DD-Paths of the program graph**
  - **Edges represent control flow between successor DD-Paths.**
- Also known as **Control Flow Graph**

Control Flow Graph Derivation

- Straightforward process
- Some judgment is required
- The last statement in a segment must be
  - **a predicate**
  - **a loop control**
  - **a break**
  - **a method exit**

Triangle program DD-path graph



displayLastMsg – Example Java program

```
public int displayLastMsg(int nToPrint) {
  np = 0;
  if ((msgCounter > 0) && (nToPrint > 0)) {
    for (int j = lastMsg; (( j != 0) && (np < nToPrint)); --j) {
      System.out.println(messageBuffer[j]);
      ++np;
    }
    if (np < nToPrint) {
      for (int j = SIZE; ((j != 0) && (np < nToPrint)); --j) {
        System.out.println(messageBuffer[j]);
        ++np;
      }
    }
  }
  return np;
}
```

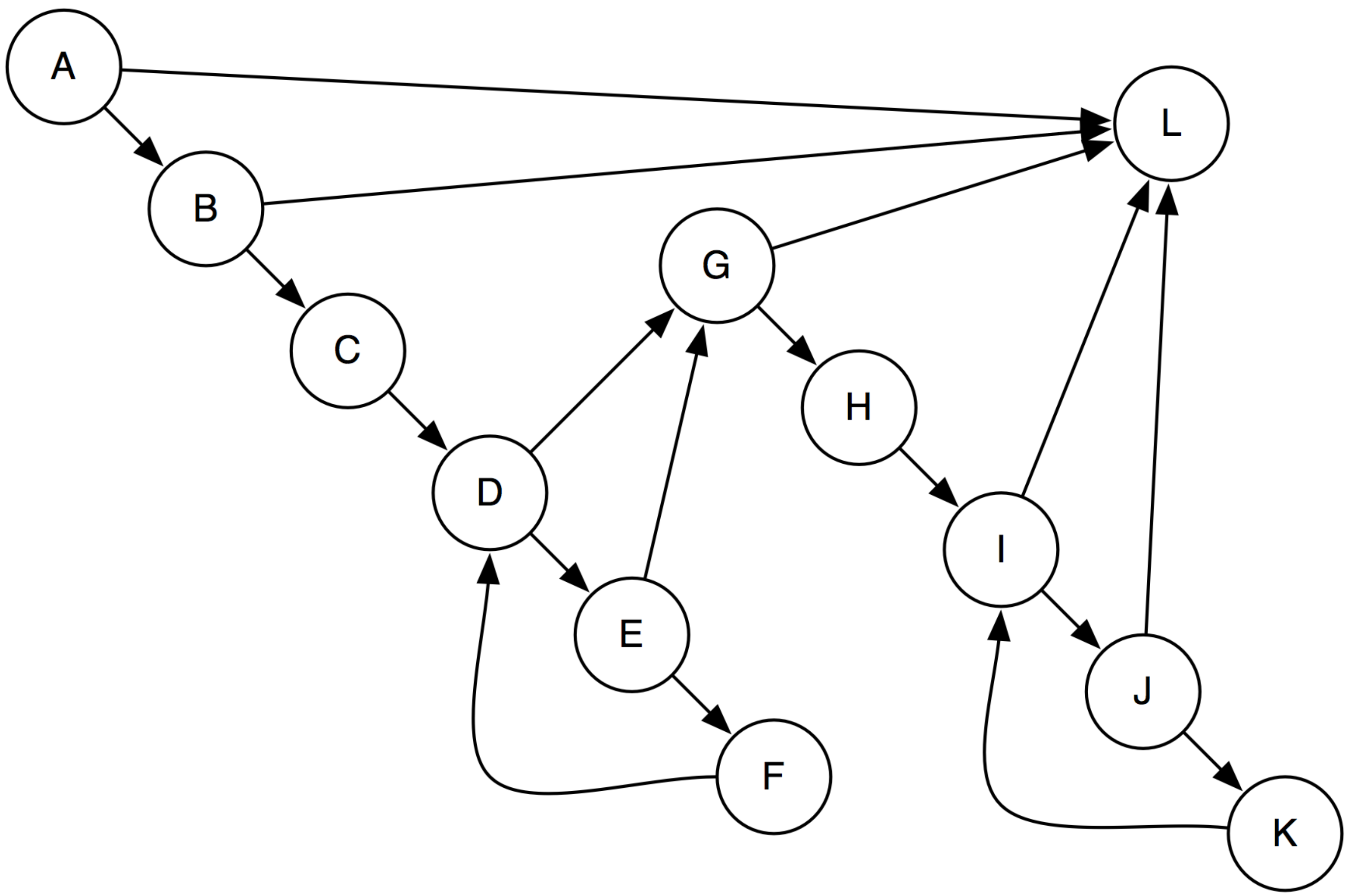
displayLastMsg– Segments part 1

Line		Segment
1	`public int displayLastMsg(int nToPrint) {`	
2	`np = 0;`	A
3a	`if (    (msgCounter > 0)`	A
3b	`&& (nToPrint > 0))`	B
4a	`{ for (int j = lastMsg;`	C
4b	`(    ( j != 0)`	D
4c	`&& (np < nToPrint));`	E
4d	`--j)`	F
5	`{ System.out.println(messageBuffer[j]);`	F
6	`++np;`	F
7	`}`	F

displayLastMsg– Segments part 2

Line		Segment
8	`if (np < nToPrint)`	G
9a	`{ for (int j = SIZE;`	H
9b	`((j != 0) &&`	I
9c	`(np < nToPrint));`	J
9d	`--j)`	K
10	`{ System.out.println(messageBuffer[j]);`	K
11	`++np;`	K
12	`}`	L
13	`}`	L
14	`}`	L
15	return np;	L
16	`}`	L

displayLastMsg – Control Flow Graph



Control flow graphs definition – 1

- Depict which program segments may be followed by others
- A segment is a node in the CFG
- A conditional transfer of control is a **branch** represented by an edge
- An **entry node** (no inbound edges) represents the entry point to a method
- An **exit node** (no outbound edges) represents an exit point of a method

Control flow graphs definition – 2

- An **entry-exit path** is a path from the entry node to the exit node
- **Path expressions** represent paths as sequences of nodes
  - **How do we represent path expressions?**

Control flow graphs definition – 3

- An **entry-exit path** is a path from the entry node to the exit node
- **Path expressions** represent paths as sequences of nodes
  - **Regular expressions involving loops**
    - **Loops are represented as segments within parentheses followed by an asterisk**
- There are 22 different entry-exit path expressions in displayLastMsg

Entry-exit path expressions – part 1

Entry-Exit paths

1	A L
2	A B L
3	A B C D G L
4	A B C D E G L
5	A B C (D E F)* D G L
6	A B C (D E F)* D E G L
7	A B C D G H I L
8	A B C D G H I J L
9	A B C D G H (I J K)* I L
10	A B C (D E F)* D E G H (I J K)* I J L
11	A B C D E G H I L

Entry-exit path expressions – part 2

Entry-Exit paths

12	A B C D E G H I J L
13	A B C D E G H (I J K)* I L
14	A B C D E G H (I J K)* I J L
15	A B C (D E F)* D G H I L
16	A B C (D E F)* D G H I J L
17	A B C (D E F)* D G H (I J K)* I L
18	A B C (D E F)* D G H (I J K)* I J L
19	A B C (D E F)* D E G H I L
20	A B C (D E F)* D E G H I J L
21	A B C (D E F)* D E G H (I J K)* I L
22	A B C (D E F)* D E G H (I J K)* I J L

Paths displayLastMsg – decision table – part 1

Path condition by Segment Name

	Entry/Exit Path	A	B	D	E	G	I	J
1	A L	F	–	–	–	–	–	–
2	A B L	T	F	–	–	–	–	–
3	A B C D G L	T	T	F	–	F	–	–
4	A B C D E G L	T	T	T	F	–	–	–
5	A B C (D E F)* D G L	T	T	T/F	T/–	F	–	–
6	A B C (D E F)* D E G L	T	T	T/T	T/F	F	–	–
7	A B C D G H I L	T	T	F	–	T	F	–
8	A B C D G H I J L	T	T	F	–	T	T	F
9	A B C D G H (I J K)* I L	T	T	F	–	T/F	T/–	T
10	A B C D G H (I J K)* I J L	T	T	F	–	T/T	T/F	T
11	A B C D E G H I L	T	T	T	F	T	F	–

x/x Conditions at loop-entry / loop-exit – is don't care

Paths displayLastMsg – decision table – part 2

Path condition by Segment Name

	Entry/Exit Path	A	B	D	E	G	I	J
12	A B C D E G H I J L	T	T	T	F	T	T	F
13	A B C D E G H (I J K)* I L	T	T	T	F	T	T/F	T/–
14	A B C D E G H (I J K)* I J L	T	T	T	F	T	T/T	T/F
15	A B C (D E F)* D G H I L	T	T	T/F	T/–	T	F	–
16	A B C (D E F)* D G H I J L	T	T	T/T	T/F	T	T	F
17	A B C (D E F)* D G H (I J K)* I L	T	T	T/F	T/–	T	T/F	T/–
18	A B C (D E F)* D G H (I J K)* I J L	T	T	T/F	T/–	T	T/T	T/F
19	A B C (D E F)* D E G H I L	T	T	T/T	T/F	T	F	–
20	A B C (D E F)* D E G H I J L	T	T	T/T	T/F	T	T	F
21	A B C (D E F)* D E G H (I J K)* I L	T	T	T/T	T/F	T	T	T
22	A B C (D E F)* D E G H (I J K)* I J L	T	T	T/T	T/F	T	T	T

x/x Conditions at loop-entry / loop-exit – is don't care

Program text coverage Metrics

- **List the program text coverage metrics.**

Program text coverage Metrics – 2

- C_0 Every Statement
- C_1 Every DD-path
- C_{1p} Every predicate to each outcome
- C_2 C_1 coverage + loop coverage
- C_d C_1 coverage + every dependent pair of DD-paths
- C_{MCC} Multiple condition coverage
- C_{lk} Every program path that contains k loop repetitions
- C_{stat} Statistically significant fraction of the paths
- C_∞ Every executable path

Program text coverage models

- **What are the common program text coverage models?**

Program text coverage models – 2

- Statement Coverage
- Segment Coverage
- Branch Coverage
- Multiple-Condition Coverage

Statement coverage – C_0

- **When is statement coverage achieved?**

Statement coverage – C_0 – 2

- Achieved when all statements in a method have been executed at least once
- A test case that will follow the path expression below will achieve statement coverage in our example

A B C (D E F)* D G H (I J K)* I L

- One test case is enough to achieve statement coverage!

Segment coverage

- **When is segment coverage achieved?**

Segment coverage – 2

- Achieved when all segments have been executed at least once
  - **Segment coverage counts segments rather than statements**
  - **May produce drastically different numbers**
    - **Assume two segments P and Q**
    - **P has one statement, Q has nine**
    - **Exercising only one of the segments will give either 10% or 90% statement coverage**
    - **Segment coverage will be 50% in both cases**

Statement coverage problems

- **What problems are there with statement coverage?**

Statement coverage problems – 2

- Important cases may be missed
  - **Predicate may be tested for only one value**
    - **Misses many bugs**
  - **Loop bodies may only be iterated only once**

```
String s = null;
if (x != y) s = "Hi";
String s2 = s.substring(1);
```

- **What coverage solves this problem?**
  - **Define it**

Branch coverage – C_{1p}

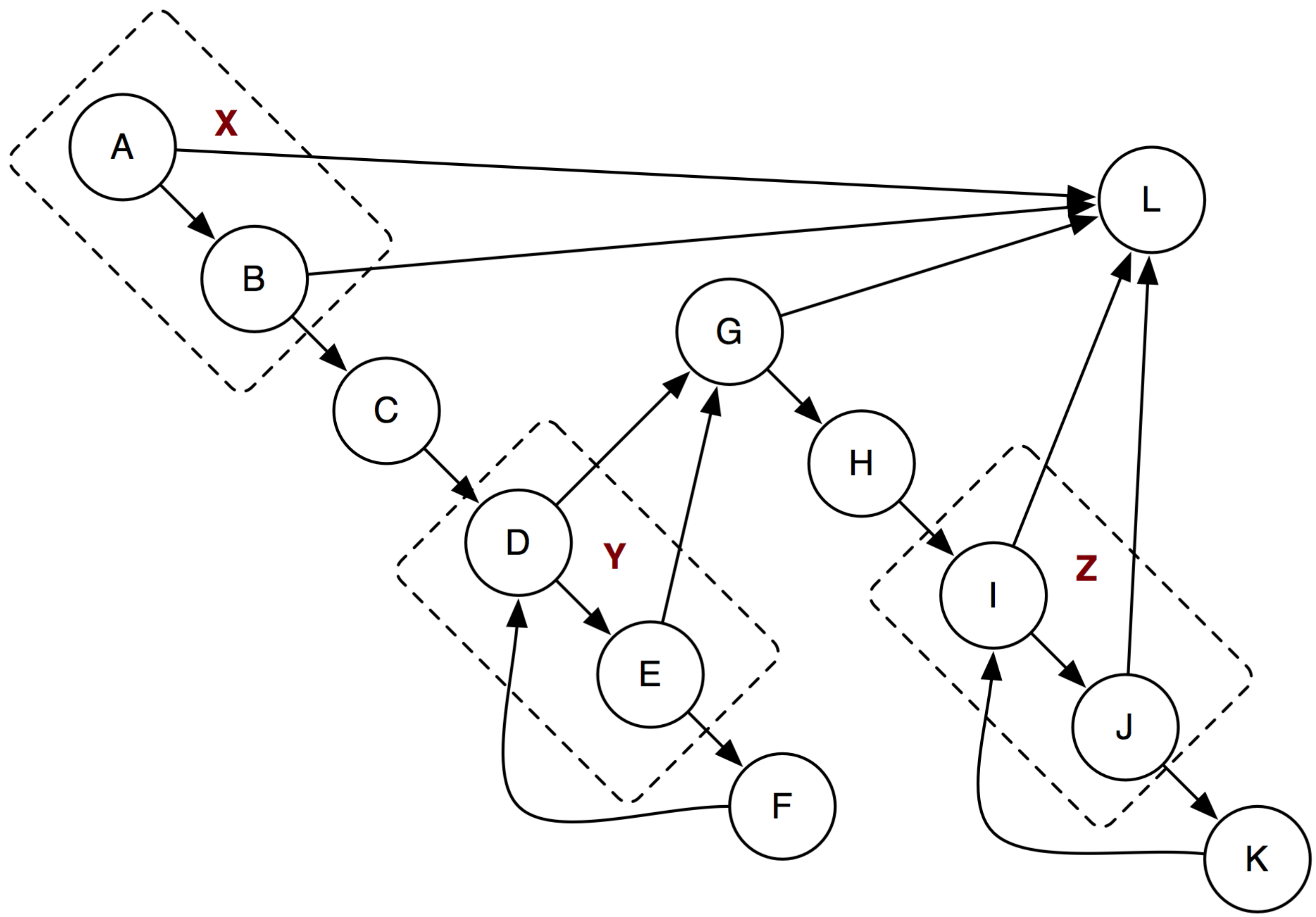
- Achieved when every edge from a node is executed at least once
- At least one true and one false evaluation for each predicate
- **How many test cases are required?**

Branch coverage – C_{1p} – 2

- Can be achieved with D+1 paths in a control flow graph with D 2-way branching nodes and no loops
  - **Even less if there are loops**
- In the Java example displayLastMsg branch coverage is achieved with three paths – see next few slides

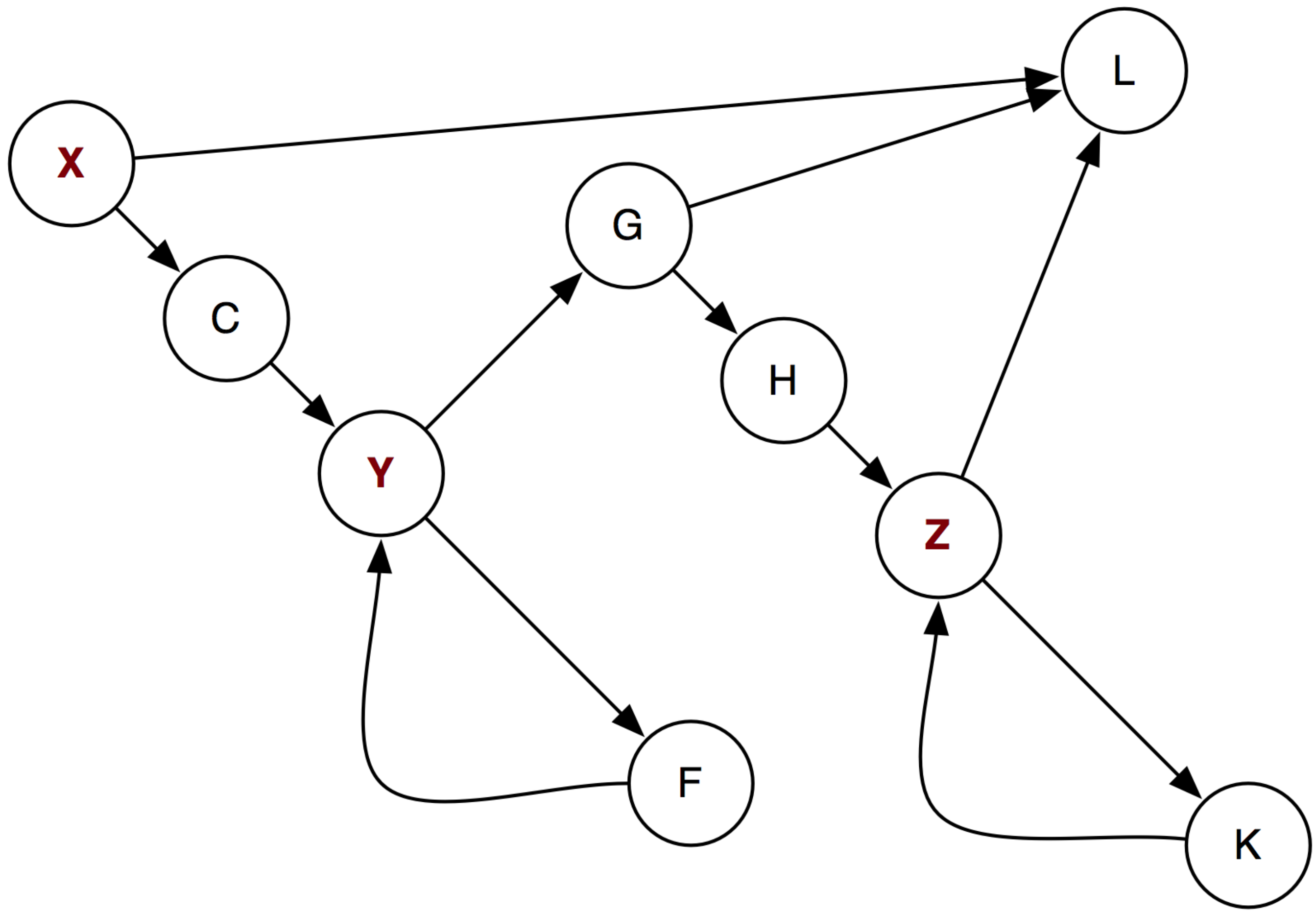
X L
X C (Y F)* Y G L
X C (Y F)* Y G H (Z K)* Z L

Java example program displayLastMsg – DD-path graph



X, Y & Z are shorthand for the nodes within the dotted boxes; used for branch testing

Java example program displastLastMsg – aggregate predicate DD-path graph



Aggregate Paths – decision table – part 1

Path condition by Segment Name

	Branch Coverage	A	B	D	E	G	I	J
1	X L	F	–	–	–	–	–	–
2	X L	T	F	–	–	–	–	–
3	X C Y G L	T	T	F	–	F	–	–
4	X C Y G L	T	T	T	F	–	–	–
5	X C (Y F)* Y G L	T	T	T/F	T/–	F	–	–
6	X C (Y F)* Y G L	T	T	T/T	T/F	F	–	–
7	X C Y G H Z L	T	T	F	–	T	F	–
8	X C Y G H Z L	T	T	F	–	T	T	F
9	X C Y G H (Z K)* I L	T	T	F	–	T/F	T/–	T
10	X C Y G H (Z K)* I L	T	T	F	–	T/T	T/F	T
11	X C Y G H Z L	T	T	T	F	T	F	–

x/x Conditions at loop-entry / loop-exit – is don't care

Aggregate Paths – decision table example – part 2

Path condition by Segment Name

	Branch Coverage	A	B	D	E	G	I	J
12	X C Y G H Z L	T	T	T	F	T	T	F
13	X C Y G H (Z K)* Z L	T	T	T	F	T	T/F	T/–
14	X C Y G H (Z K)* Z L	T	T	T	F	T	T/T	T/F
15	X C (Y F)* Y G H Z L	T	T	T/F	T/–	T	F	–
16	X C (Y F)*Y G H Z L	T	T	T/T	T/F	T	T	F
17	X C (Y F)* Y G H (Z K)* Z L	T	T	T/F	T/–	T	T/F	T/–
18	X C (Y F)* Y G H (Z K)* Z L	T	T	T/F	T/–	T	T/T	T/F
19	X C (Y F)* Y G H Z L	T	T	T/T	T/F	T	F	–
20	X C (Y F)* Y G H Z L	T	T	T/T	T/F	T	T	F
21	X C (Y F)* Y G H (Z K)* Z L	T	T	T/T	T/F	T	T	T
22	X C (Y F)* Y G H (Z K)* Z L	T	T	T/T	T/F	T	T	T

x/x Conditions at loop-entry / loop-exit   – is don't care

Branch coverage problems

- **What are the problems with branch coverage?**

Branch coverage problems – 2

- Ignores implicit paths from compound paths
  - **11 paths in aggregate model**

    vs
  - **22 in full model**

Branch coverage problems – 3

- Short-circuit evaluation means that many predicates might not be evaluated
  - **A compound predicate is treated as a single statement**
  - **If n clauses**
    - **2^n combinations**
    - **Only 2 are tested**

Branch coverage problems – 4

- Only a subset of all entry-exit paths is tested
  - **Two tests for branch coverage vs 4 tests for path coverage**
    - **a = b = x = y = 0**
    - **a = x = 0 $\wedge$ b = y = 1**

```
if (a == b) x++;
if (x == y) x--;
```

Overcoming branch coverage problems

- **How do we overcome branch coverage problems?**

Overcoming branch coverage problems – 2

- **Use Multiple condition coverage**
  - **All true-false combinations of simple conditions in compound predicates are considered at least once**
    - **Guarantees statement, branch and predicate coverage**
    - **Does not guarantee path coverage**

Overcoming branch coverage problems – 3

- **Use Multiple condition coverage**
  - **A truth table may be necessary**
  - **Not necessarily achievable**
    - **Lazy evaluation – true-true and true-false are impossible**
    - **Mutually exclusive conditions – false-false branch is impossible**

```
if ((x > 0) || (x < 5)) …
```

Overcoming branch coverage problems – 4

- Can have infeasible paths due to dependencies and redundant predicates
  - **Paths perpetual .. motion and free .. lunch are impossible**
  - **In this case indicates a potential bug**
    - **At least poor program text**

```
if x = 0 then oof.perpetual
         else off.free
fi

if x != 0 then oof.motion
          else off.lunch
fi
```

Dealing with Loops

- Loops are highly fault-prone, so they need to be tested carefully
- **Based on the previous slides on testing decisions what would be a simple view of testing a loop?**

Dealing with Loops – 2

- Simple view
  - **Involves a decision to traverse the loop or not**
  - **Test as a two way branch**

- **What would functional testing suggest as a better way of testing?**
- **What tests does it suggest?**

Dealing with Loops – 3

- A bit better
  - **Boundary value analysis on the index variable**
  - **Suggests**
    - **Zero iterations**
    - **One iteration**
    - **Many iterations**

- **How do we deal with nested loops?**

Dealing with Loops – 3

- Nested loops
  - **Tested separately starting with the innermost**

- **Once loops have been tested what can we do with the control flow graph?**

Dealing with Loops – 4

- Once loops have been tested
  - **They can be condensed to a single node**

Condensation graphs

- **What is a condensation graph?**

Condensation graphs – 2

- **What is a condensation graph?**
  - **Condensation graphs are based on removing strong components or DD-paths**
  - **For programs remove structured program constructs**
    - **One entry, one exit constructs for sequences, choices and loops**
    - **Each structured component once tested can be replaced by a single node when condensing a graph**

Violations of proper structure

- Program text that violates proper structure cannot be condensed
  - **Branches either into or out of the middle of a loop**
  - **Branches either into or out of then and else phrases of if…then…else statements**
  - **Increases the complexity of the program**
  - **Increases the difficulty of testing the program**

Cyclomatic number

- The cyclomatic number for a graph is given by
  - **CN(G) = e – v + 2*c**
    - **e number of edges**
      v number of vertices
      c number of connected regions
  - **For strongly connected graphs, need to add edges from every sink to every source**

Cyclomatic number for structured programs

- For properly structured programs there is only one component with one entry and one exit. There is no edge from exit to entry.
- Definition 1: **$CN(G) = e - v + 2$**
  - **Only 1 component, not strongly connected**
- Definition 2: **$CN(G) = p + 1$**
  - **p is the number of predicate nodes with out degree = 2**
- Definition 3: **$CN(G) = r + 1$**
  - **r is the number of enclosed regions**