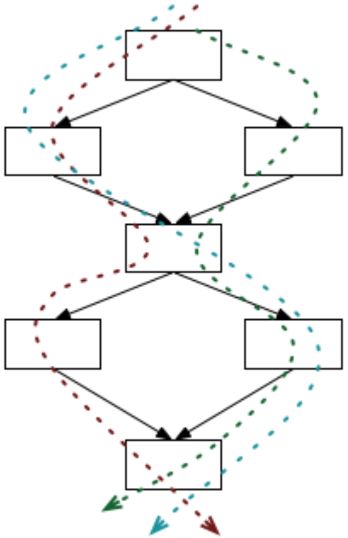
Structural Testing & Test Adequacy – Part 2

How good are your tests?

Path-based criteria, cyclomatic complexity, ~~data-flow testing~~

Paths? (beyond individual branches)

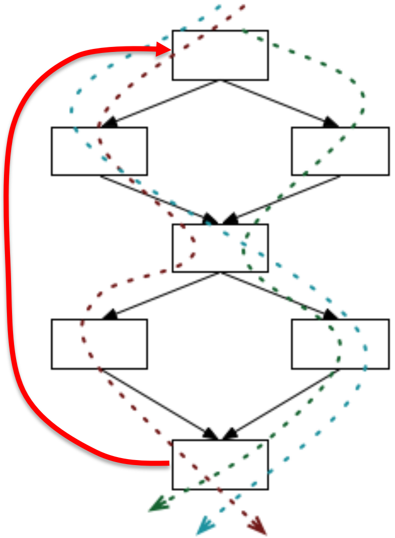
- Should we explore sequences of branches (paths) in the control flow?
- Many more paths than branches
  - Must combine all possible outcomes of all decisions
  - A pragmatic compromise will be needed



- 4 branches here

Paths? (beyond individual branches)

- Should we explore sequences of branches (paths) in the control flow?
- If program has loops => too many paths

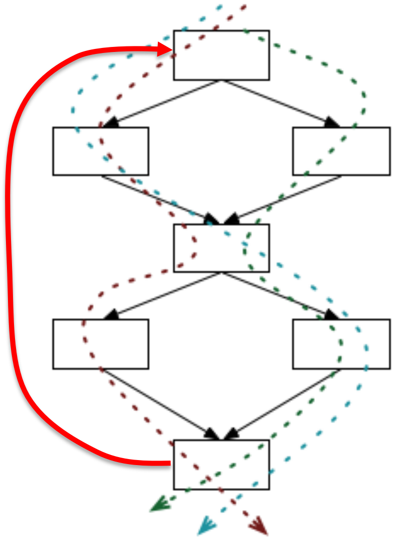


Optimized versions of path coverage exist

Example: **loop boundary adequacy**

Each loop is executed

- 0,
- 1, and
- multiple times



Cyclomatic Complexity and Cyclomatic Adequacy (McCabe's Basis Path Coverage)

A compromise between

full path coverage

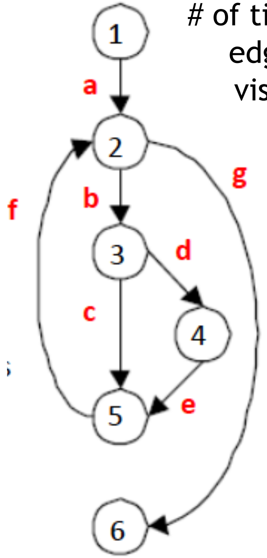
and

simple branch coverage

cyclomatic complexity is a common measure of code complexity related to the vector representation of paths

higher ←→ *program's logical (branching) structure is more complex*

To understand cyclomatic complexity fully , we need to represent paths through the CFG as vectors



edges
# of times edge is visited
path

a	b	c	d-e	f	g	Path (nodes)
1	0	0	0	0	1	1,2,6
1	1	1	0	1	1	1,2,3,5,2,6
1	1	0	1	1	1	1,2,3,4,5,2,6
1	2	1	1	2	1	1,2,3,5,2,3,4,5,2,6
1	2	1	1	2	1	1,2,3,4,5,2,3,5,2,6
1	2	2	0	2	1	1,2,3,5,2,3,5,2,6
1	2	0	2	2	1	1,2,3,4,5,2,3,4,5,2,6
		⋮				
1	20	0	20	20	1	1,2,3,4,5,2,3,4,5,2,3 ,4,5,2,..........,2,6
1	20	1	19	20	1	
		⋮				

Incidence Matrix

Mathematically: cyclomatic complexity is the max number of *linearly independent* paths that we can have in a CFG

- P_1, .., P_n: paths represented as vectors
- P_1,…, P_n are *linearly independent* if none of the P_i can be expressed as a linear combination of the others
  - If P_1 (or any other path) can be expressed as a linear combination of the others, e.g.,

$$P_1 = a_2P_2 + \ldots + a_nP_n$$

… then P_1,…, P_n are *linearly dependent*

"+" is vector addition, a_i are integer constants

Determining cyclomatic complexity, V(G), is easy!

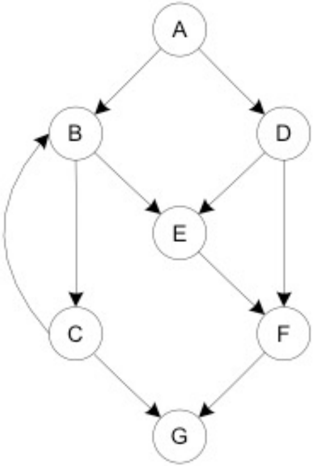
- Consider a directed graph G with
- e = # of edges, n = # of nodes,
  c = # of connected regions of a graph, then…

- V(G) = e − n + 2c for an arbitrary graph
- It can be proven that: V(G) = e − n + 2 if G is connected
  (c = 1 for CFG of a contiguous piece of code)

A graph theoretic result!

Cyclomatic complexity of a CFG

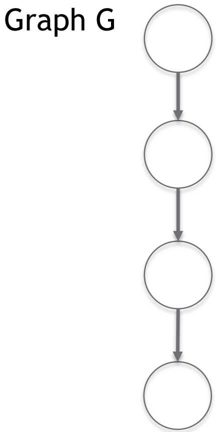
?



Graph G

Cyclomatic complexity = V(G) = e – n + 2

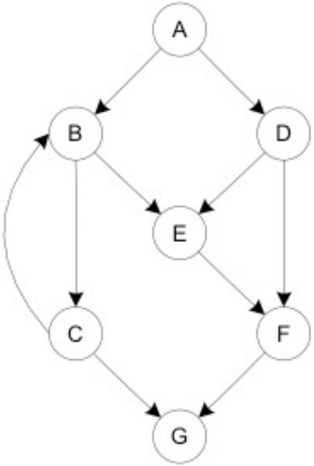
Cyclomatic complexity of a CFG



Minimum cyc. complexity: 1
Corresponds to a program fragment with no control structures

Cyclomatic complexity = V(G) = 4 – 3 + 2 = 1

Cyclomatic complexity of a CFG



Graph G

Cyclomatic complexity = V(G) = e – n + 2 = 10 – 7 + 2 = 5

Linear independence = Basis paths

- Any set of linearly independent paths of size V(G) is called a **basis** of G
- The basis covers all paths in G, it does not just cover G
- Each path in the basis is called a **basis path**

- How to determine whether a set of paths forms a basis?

  Order basis paths from shortest to longest, and check:
  – each basis path introduces at least one new edge that is not already included in any other basis path*
  – every edge of G is included in a basis path
  – there are V(G) paths in the set

new edge

full edge coverage

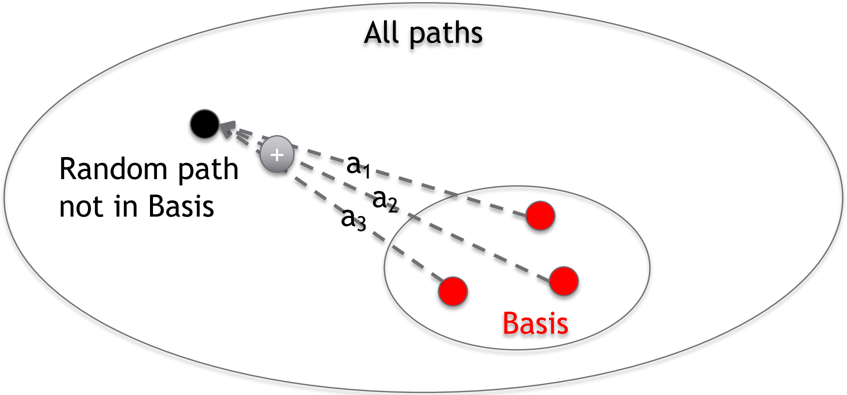
right number

*alternative condition applies to loops

- The basis is not unique !

The basis covers "all paths" in G, not just G, in a special mathematical sense

Any path through the CFG can be represented as a linear combination of the paths in the basis



Cyclomatic adequacy

- Do the executed paths include a basis?
- In other terms: *Is a basis covered?*
  YES => full cyclomatic coverage

Attractive because it answers the question:

- *To what extent does the test suite compensate for program's complexity?*
  - *Closely related to branch coverage, but stronger*

Coverage measure: $$\frac{V(G) - \overbrace{(\text{min \# extra paths needed to make a basis})}^{\textit{how may paths are we short?}}}{V(G)}$$

Example

A Java method with V(G) = 5

- Test suite covers 10 paths
- 1 more path needed to make a basis
- Cyclomatic coverage = ?

Mentimeter

Example

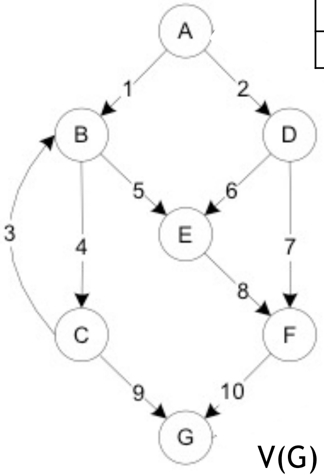
A Java method with V(G) = 5

- Test suite covers 10 paths
- 1 more path needed to make a basis
- Cyclomatic coverage = 4/5 = 80%

Cyclomatic adequacy = Linear independence

Path: Nodes *Edges →*	*e1*	*e2*	*e3*	*e4*	*e5*	*e6*	*e7*	*e8*	*e9*	*e10*
P1: A, B, C, G	1	0	0	1	0	0	0	0	1	0
P2: A, D, F, G	0	1	0	0	0	0	1	0	0	1
P3: A, B, E, F, G	1	0	0	0	1	0	0	1	0	1
P4: A, D, E, F, G	0	1	0	0	0	1	0	1	0	1
P5: A, B, C, B, C, G	1	0	1	2	0	0	0	0	1	0
P6: A, B, C, B, E, F, G	1	0	1	1	1	0	0	1	0	1
P7: A, B, C, B, C, B, C, G	1	0	2	3	0	0	0	0	1	0

basis



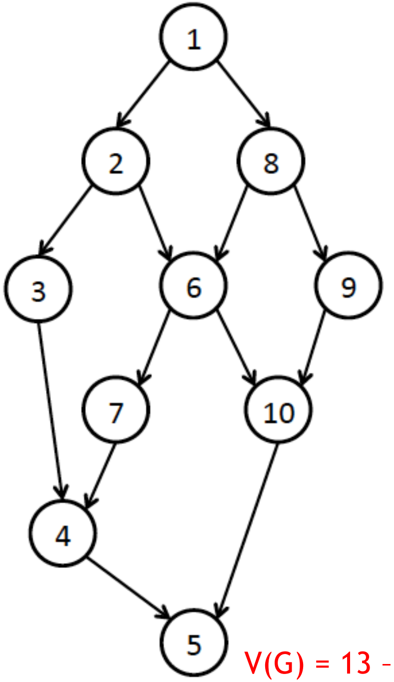
V(G) = 5

- Incidence matrix of CFG *relative to* given paths
- Each row is a vector that represents a path

Sample paths as linear combinations of basis paths:

- P6 = P5 + P3 – P1
- P7 = 2•P5 – P1

Selecting basis paths (no loops)



V(G) = 13 – 10 + 2 = 5

How do I exercise all feasible combinations of all outcomes of all decisions without full path coverage?

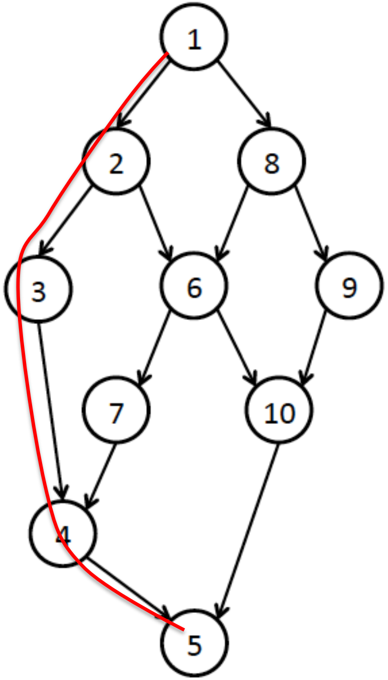
Part of a combination can be covered in one path and the remaining part in another path

- 1,2,3,4,5
- 1,2,6,7,4,5
- 1,2,6,10,5
- 1,8,6,7,4,5
- 1,8,6,10,5
- 1,8,9,10,5

Take advantage of overlapping test cases like in All-Pairs or MC/DC

Claire Le Goues (2014)

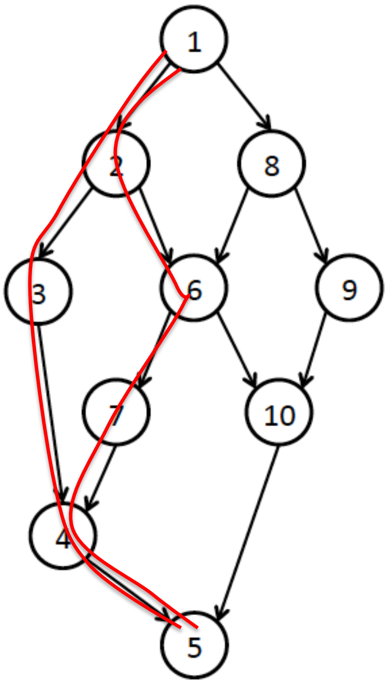
Selecting basis paths (no loops)



Paths	Decisions
– ~~1,2,3,4,5~~	Combine 1 and 2
– 1,2,6,7,4,5	
– 1,2,6,10,5	
– 1,8,6,7,4,5	
– 1,8,6,10,5	
– 1,8,9,10,5	

Claire Le Goues (2014)

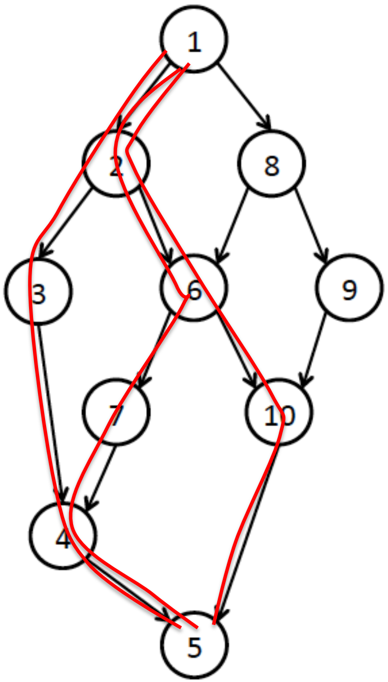
Selecting basis paths (no loops)



Paths	Decisions
– ~~1,2,3,4,5~~	
– ~~1,2,6,7,4,5~~	Combine 1 and !2 and 6
– 1,2,6,10,5	
– 1,8,6,7,4,5	
– 1,8,6,10,5	
– 1,8,9,10,5	

Claire Le Goues (2014)

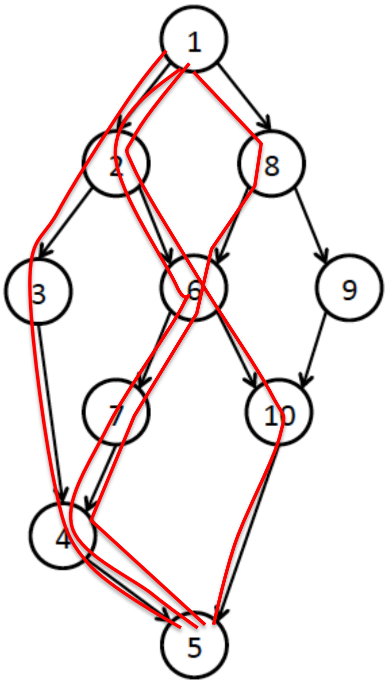
Selecting basis paths (no loops)



Paths	Decisions
– ~~1,2,3,4,5~~	
– ~~1,2,6,7,4,5~~	
– ~~1,2,6,10,5~~	Combine 1 and !2 and !6
– 1,8,6,7,4,5	
– 1,8,6,10,5	
– 1,8,9,10,5	

Claire Le Goues (2014)

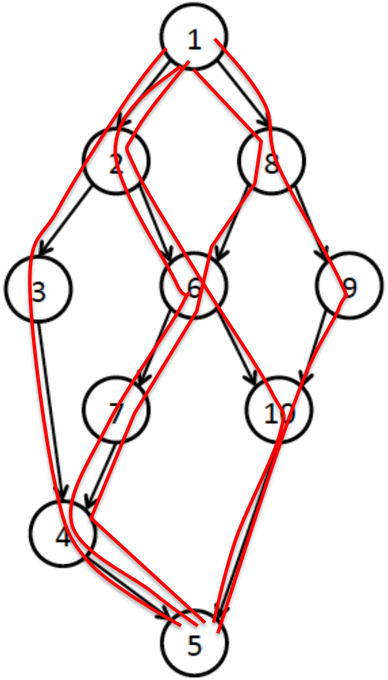
Selecting basis paths (no loops)



Paths	Decisions
– ~~1,2,3,4,5~~
– ~~1,2,6,7,4,5~~
– ~~1,2,6,10,5~~
– ~~1,8,6,7,4,5~~	Combine !1 and 8 and 6
– 1,8,6,10,5
– 1,8,9,10,5

Claire Le Goues (2014)

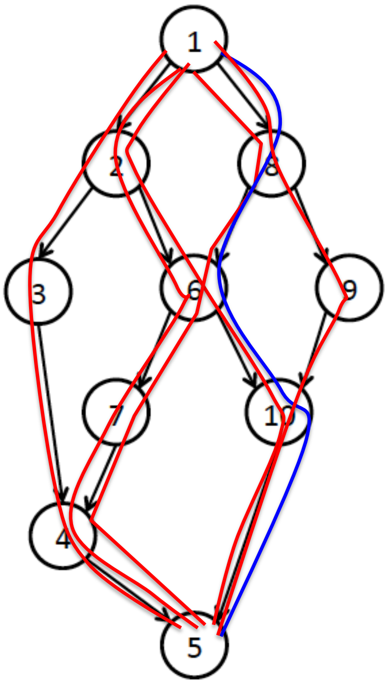
Selecting basis paths (no loops)



Paths	Decisions
– ~~1,2,3,4,5~~	
– ~~1,2,6,7,4,5~~	
– ~~1,2,6,10,5~~	
– ~~1,8,6,7,4,5~~	
– 1,8,6,10,5	
– ~~1,8,9,10,5~~	Combine !1 and !8

Claire Le Goues (2014)

Selecting basis paths (no loops)

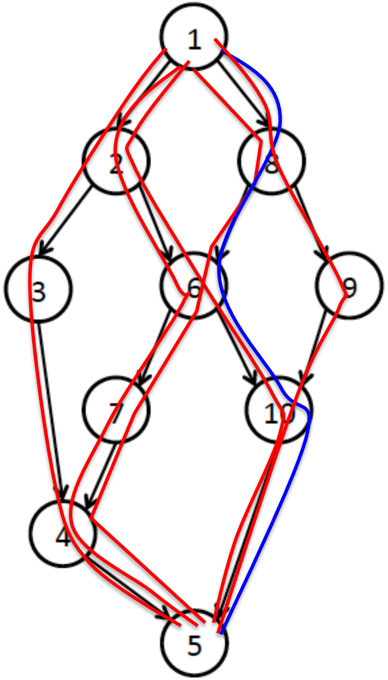


Paths | Decisions

- ~~1,2,3,4,5~~
- ~~1,2,6,7,4,5~~
- ~~1,2,6,10,5~~
- ~~1,8,6,7,4,5~~
- 1,8,6,10,5 ?
- 1,8,9,10,5

Claire Le Goues (2014)

Selecting basis paths (no loops)



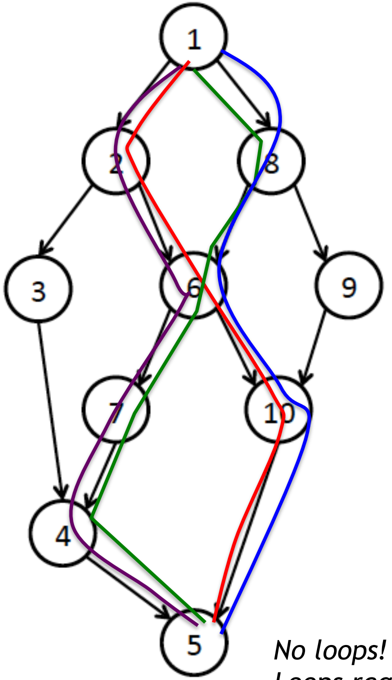
Can represent this with other paths

- ~~1,2,3,4,5~~
- ~~1,2,6,7,4,5~~
- ~~1,2,6,10,5~~
- ~~1,8,6,7,4,5~~
- 1,8,6,10,5 Don't need
- ~~1,8,9,10,5~~

Combine !1 and 8 and !6

Claire Le Goues (2014)

Selecting basis paths (no loops)



(1,8,6,10,5) =

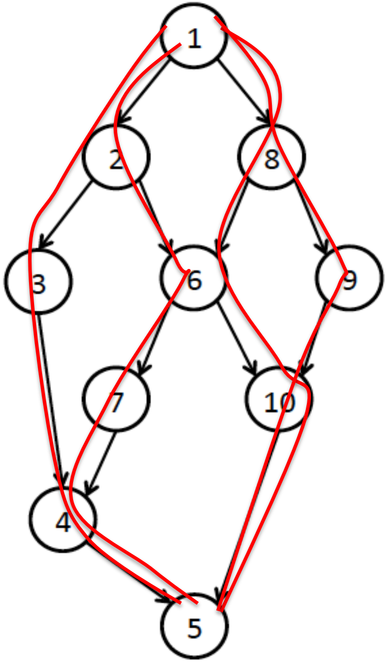
(1,8,6,7,4,5) + (1,2,6,10,5) - (1,2,6,7,4,5)

Paths	Decisions
– ~~1,2,3,4,5~~	
– ~~1,2,6,7,4,5~~	3. Subtract: Combine 1 and !2 and 6
– ~~1,2,6,10,5~~	2. Add: Combine 1 and !2 and 6
– ~~1,8,6,7,4,5~~	1. Add: Combine !1 and 8 and !6
– 1,8,6,10,5	Combine !1 and 8 and !6
– ~~1,8,9,10,5~~	

No loops!
Loops require special treatment.

Claire Le Goues (2014)

Not enough paths?



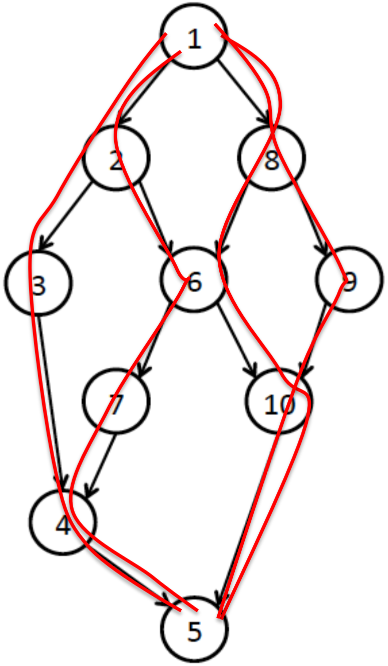
What about these 4 paths?
They cover G, but do they form a basis?

- 1,2,3,4,5
- 1,2,6,7,4,5

- 1,8,6,10,5
- 1,8,9,10,5

Claire Le Goues (2014)

Not enough paths?



What about these 4 paths?
They cover G, but do they form a basis?
No, because you can't reproduce all other paths with a linear combination of them.
E.g., for 1,2,6,10,5:
You need:

- 1,2,6,7,4,5 (because of edge 2-6)
- 1,8,6,10,5 (because of edge 6-10)

But you have to subtract edge 6-7, but there are no other paths that contain that edge!

- 1,2,3,4,5
- ~~1,2,6,7,4,5~~
- ~~1,8,6,10,5~~
- 1,8,9,10,5

Claire Le Goues (2014)

Heuristic for cyclomatic testing: generate a basis

1. Produce the CFG G of program
2. Generate all basis paths
   a. Pick a baseline path that represents a nominal behavior, not an error outcome
      - *avoid entering loops*
   b. Retrace the baseline path and “flip” the first un-flipped decision encountered
      - *loop exit decisions require special care: when you encounter a new loop, choose exit decision before entry decision so skipping loop is for sure covered*
   c. Repeat this until all decisions have been flipped/covered or when you reach V(G)
3. Analyze the conditions that force the execution along each basis path
4. Write test cases that will generate the conditions defined in 3

Note:

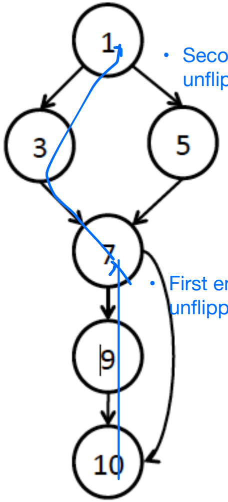
- the basis is not unique
- there might be infeasible basis paths (impossible to exercise)

Claire Le Goues (2014)

Generating test cases from a basis (no loops)

F

```
1.  if(a < 0)
2.  {
3.      S1
4.  } else {
5.      S2
6.  }
7.  if(b > 10)
8.  {
9.       S4
10. }
```



$V(G)=7-6+2=3$

- Second encountered unflipped point
- First encountered unflipped pointt

Base line path P1: 1, 3, 7, 9, 10

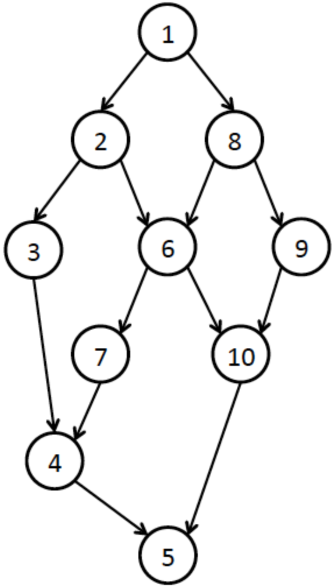
Flip 7 to get path P2: 1, 3, 7, 10

Flip 1 to get path P3: 1, 5, 7, 9, 10

	a	b
P1	-1	11
P2	-1	10
P3	2	11

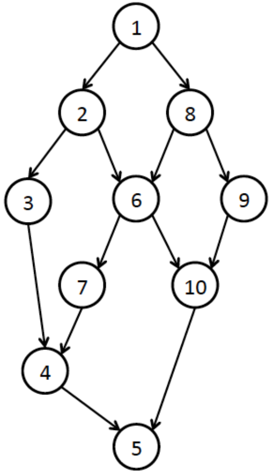
Claire Le Goues (2014)

Find a basis for this CFG using the algorithm described!

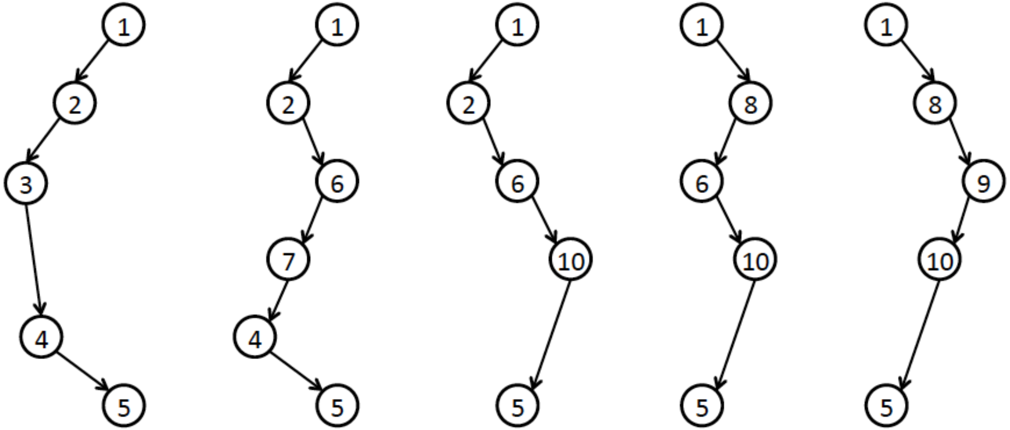


Claire Le Goues (2014)

Is the RHS a basis for the CFG on the LHS?



=?



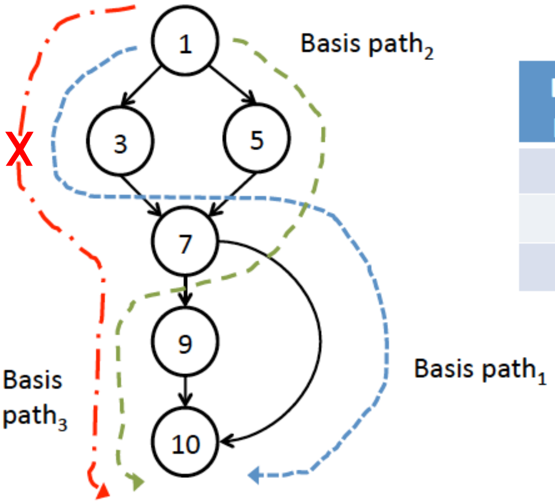
Claire Le Goues (2014)

Infeasibility in cyclomatic testing

```
1.  if(a < 0)
2.  {
3.     S1
4.  } else {
5.     S2
6.  }
7.  if(a > 10)
8.  {
9.     S4
10. }
```

cannot be executed together

★ Assume S_1, S_2 don't modify variable a



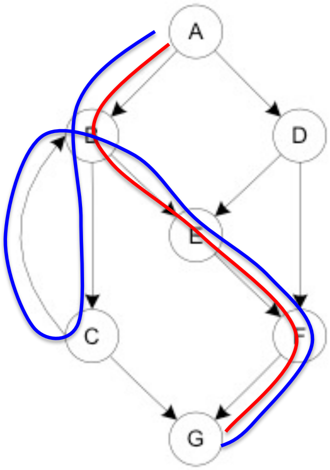
Basis path	a
1	-1
2	11
3	X

Some paths that can be derived from the topology of the CFG are semantically impossible due to unreachable code. This usually happens when a variable appears in more than one condition along the same path.
→ Might indicate a faulty logic in the program or bad coding practice.

Claire Le Goues (2014)

Basis with loops

- Loops must be both skipped and executed!
- Need both red and blue paths even though red is included in blue;
- **Choose red (exit) path before blue (entry) path, and apply the same heuristic**



Cyclomatic complexity = V(G) = e – n + 2 = 10 – 7 + 2 = 5

You can verify that without red path, it is impossible to construct a basis with V(G) paths!

Algorith for cyclomatic testing: generate a basis with loops treated specially

1. Produce the CFG G of program
2. Generate all basis paths
   a. Pick a baseline path that represents a nominal behavior, not an error outcome
      – *avoid entering loops*
   b. Retrace the baseline path and “flip” the first un-flipped decision encountered
      – *loop exit decisions require special care: when you encounter a new loop, choose the exit decision before the entry decision so skipping loop is for sure covered*
   c. Repeat this until all decisions have been flipped/covered or when you reach V(G)
3. Analyze the conditions that force the execution along each basis path
4. Write test cases that will generate the conditions defined in 3

Note:

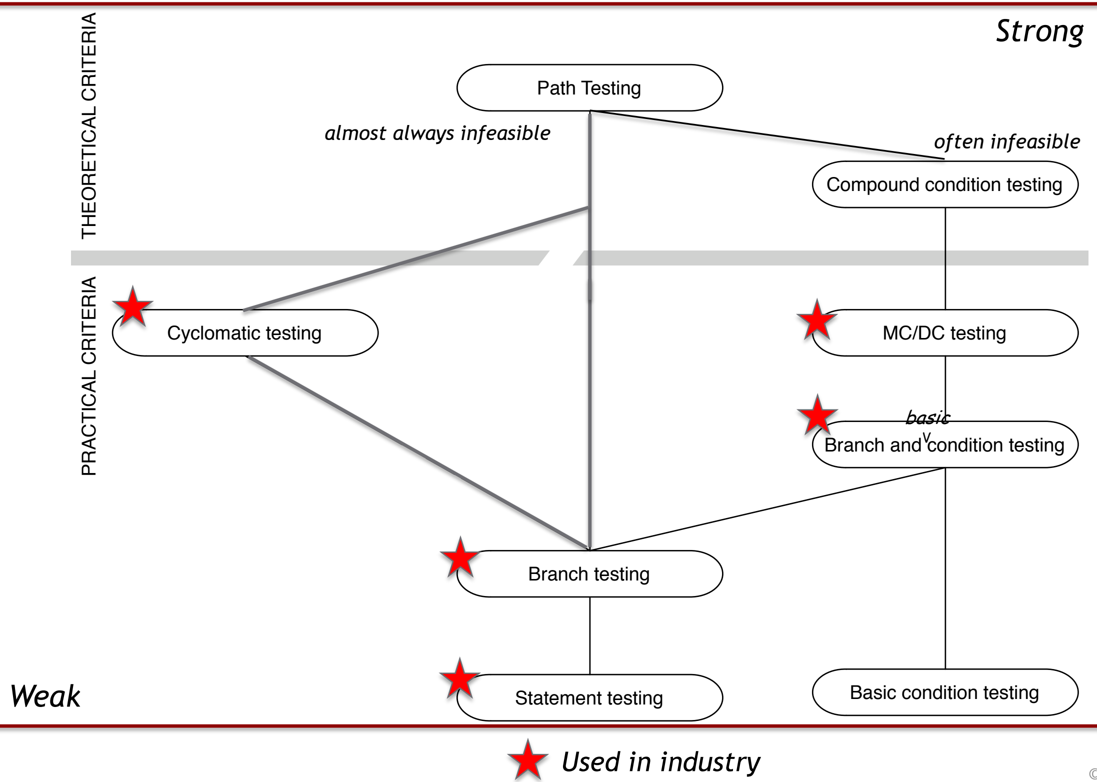
- the basis is not unique
- there might be infeasible basis paths (impossible to exercise)

Claire Le Goues (2014)

Cyclomatic adequacy

- Advantages
  - Simple, heuristic to compute basis paths (linearly independent paths) from CFG
  - Size of test suite linear in #edges
  - More complex program require proportionally more test cases
  - Reduced number of test cases
  - **Subsumes branch (decision) coverage**
- Disadvantages
  - No guarantee that basis paths are feasible (can we force execution by selecting right inputs?)
    - Manually too laborious, error prone: Need automated tools to generate the paths and path conditions for oracles
    - *Full coverage may be impossible (but this is also a disadvantage with branch coverage) -- we'll simply assume infeasible paths don't exist*

Subsumption relation for CFG coverage



Path-based testing process

- Select the coverage criterion
- Execute spec-based tests
- Generate a CFG
- Identify paths in the CFG satisfying the criterion, but not covered in spec-based tests
- Is the spec, or spec-based test design missing something?
- Build test cases – determine the oracles
  - derive path condition expressions from the selected paths and solve those expressions to derive test inputs (involves constraint solving – SAT problem)
- Execute test cases

This may be a lot of work… some of it can be automated (tools exist to generate test cases)

Structural testing: key messages

- We defined a number of adequacy criteria
  - NOT test design techniques *per se*, *but lead to* test generation/supplementation techniques!
- Different criteria address different classes of errors
- Full coverage for strong criteria may be unattainable (too many test cases)

  …and when attainable, creating actual test cases may be hard
  - How do I find program inputs (oracle problem) allowing to cover something buried deeply in the CFG?
  - Oracle problem reduces to constraint satisfaction (SAT)
  - Automated support (e.g., symbolic execution) may be necessary
  - Some paths may be infeasible - that's ok, we'll have to ignore them
- Therefore, rather than requiring full adequacy, the "degree of adequacy" of a test suite is estimated and reported by coverage measures
  - Adequacy drives test suite and test design improvement

Sources

Pezze & Young: *Software Analysis & Testing; Ch 12, 13*

Jorgensen: *Software Testing: A Craftsman's Approach*

Claire Le Goues and Eduardo Miranda, *CMU-CS & CMU-ISR*

Midterm Review Head Start

All-Pairs Coverage Example

4 characteristics

- A: 4 blocks {a1,a2, a3, a4 }
- B: 3 blocks {b1, b2, b3 }
- C: 2 blocks {c1, c2 }
- D: 2 blocks {d1, d2 }

Possible ways of choosing characteristic pairs:

C(4, 2) = 4!/2!(4 - 2!) = (4 x 3 x 2)/(2 x 2) = 6

(A, B), (A, C), (A, D), (B, C), (B, D), (C,D)

Pairwise combinations that need to be covered

Characteristic Pair	Possible Pairwise Combinations
(A, B)	4 x 3 = 12
(A, C)	4 x 2 = 8
(A, D)	4 x 2 = 8
(B, C)	3 x 2 = 6
(B, D)	3 x 2 = 6
(C, D)	2 x 2 = 4
Total: 6 Pairs	44 Pairwise Combinations

- If each case covers 6 entirely new pairs…
  - Min # of test cases = Ceiling(44/6) = 8, but…
  - to cover largest attributes (A, B), must have at least 4 x 3 = **12** test cases
    - Max(8,12) = 12
- If each case covers just 1 new pair (no overlaps):
  - Max # of test cases = **44**

Calculating all pairs coverage

pair examples:
no duplicates

Case #	A	B	C	D	# new	Cum. new
1	a1	b1	c1	d1	6	6
2	a1	b2	c1	d2	5	11
3	a1	b3	c2	d2	5	16
4	a2	b1	c1	d1	3	19
5	a2	b2	c1	d2	2	21
6	a2	b3	c2	d2	2	23
7	a3	b1	c1	d1	3	26
8	a3	b2	c1	d2	2	28
9	a3	b3	c2	d2	2	30
10	a4	b1	c1	d1	3	33
11	a4	b2	c1	d2	2	35
12	a4	b3	c2	d2	2	37

given test cases

Total pairwise combinations to be covered = 44

Covered pairwise combinations = 37

All-Pairs Coverage = 37/44 = 84%

Appendix – Other Path Coverage Criteria

(not included in exam)

Path testing

- Decision (branch) and condition adequacy criteria consider individual program decisions
- Path testing considers combinations of decisions along paths through the CFG
- Adequacy criterion: each path must be executed at least once
- Coverage measure

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

Loops?

Practical path coverage criteria

- The number of paths in a program with loops is unbounded
  - the simple criterion is usually impossible to satisfy
- For a feasible criterion: partition infinite set of paths into a finite number of blocks
- Useful criteria can be obtained by:
  - limiting number of traversals of loops
  - limiting length of the paths to be traversed (depth limit)
  - exploiting subpath relationships

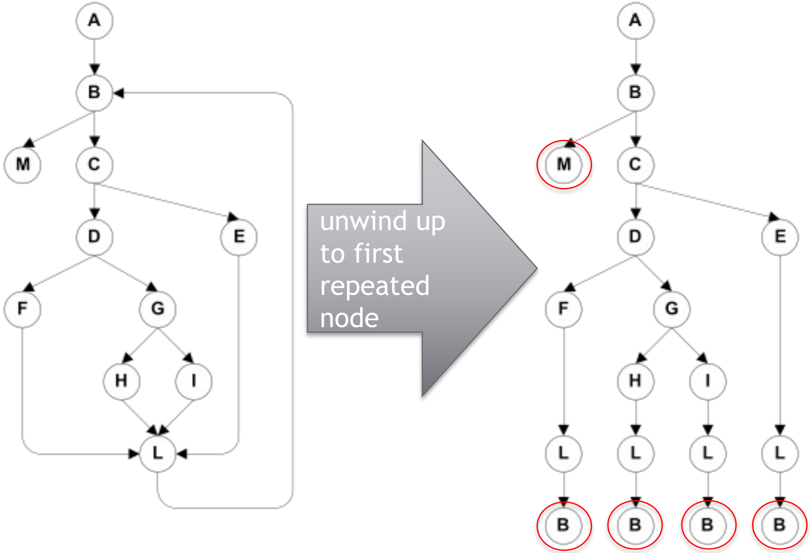
Boundary-interior path testing

- Group together paths that differ only in the sub-path they follow when repeating the body of a loop

  finite path X with sub-path Y repeated n times $\equiv$ *finite path X' where X differs from X' only in that sub-path Y is repeated n+1 times*

  - unwind CFG to 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 url-decode



CFG of url_decode

Boundary-interior adequate paths for url_decode

Limitations of boundary-interior adequacy

- The number of paths can still grow exponentially

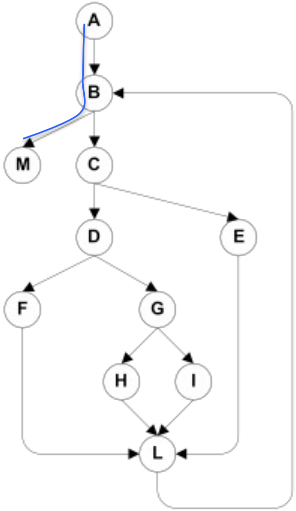
```
if (a) {
   S1;
}
if (b) {
   S2;
}
if (c) {
   S3;
}
...
if (x) {
   Sn;
}
```

- The subpaths through this control flow can include or exclude each of the statements Si, so that in 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, or even impossible if the conditions are dependent

Loop-boundary adequacy

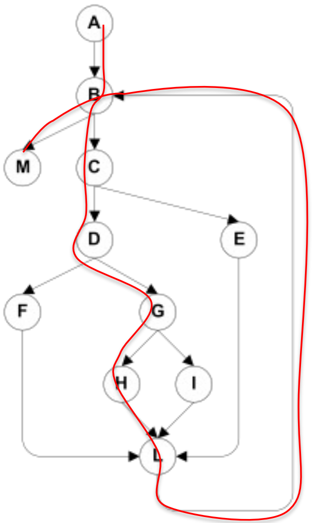
- Variant of the boundary-interior criterion that treats loop boundaries similarly but is less stringent with respect to other differences among paths
- Criterion: A test suite satisfies the loop boundary adequacy criterion iff for every loop:
  - in at least one test case, the loop body is iterated zero times
  - in at least one test case, the loop body is iterated once
  - in at least one test case, the loop body is iterated more than once
- Emulates “proof by induction”

Loop-boundary adequacy: example



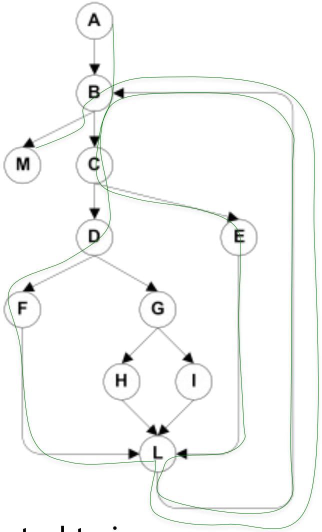
1^{st} case: No loop execution

Loop-boundary adequacy: example



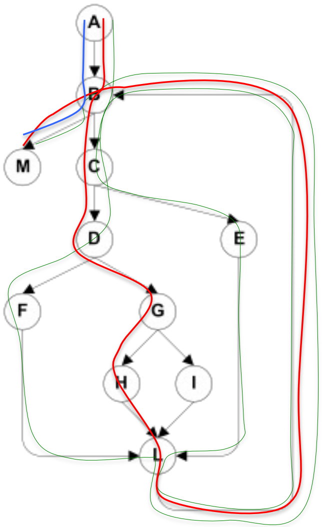
2nd case: Loop executed once

Loop-boundary adequacy: example



3rd case: Loop executed twice

Loop-boundary adequacy: example



- **Weak criterion**: covered the loop according to loop-boundary adequacy, but some branches within the loop are not executed
- Ok if combined with branch coverage

LCSAJ adequacy

- **Linear Code Sequence And Jump:**
  sequential subpath in the CFG starting and ending with a jump
  - TER_1 = statement coverage
  - TER_2 = (almost) branch coverage
  - TER_{n+2} = coverage of *n* consecutive LCSAJs

A Quick Look At

Data Flow Testing

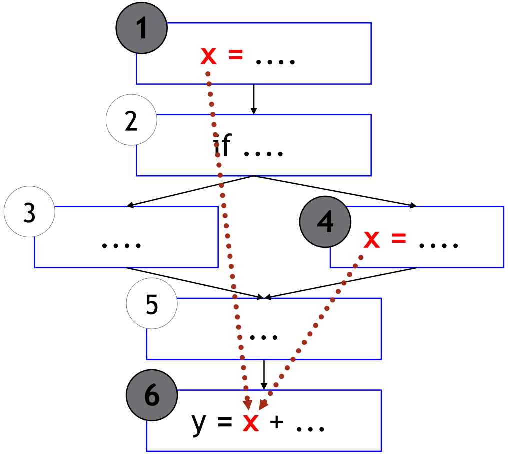
… another kind of structural testing …

Motivation for data flow testing

- Middle ground in structural testing
  - Statement and branch coverage don’t cover fine-grained interactions among program elements
  - Path-based criteria require impractical number of test cases
    - and only a few paths uncover additional faults, anyway
  - Need to distinguish “important” paths
- Intuition: statements interact through *data flow*
  - Value computed (defined) in one statement, used in another
  - Bad value computation revealed only when it is used

Data flow concept

- Value of x at 6 could be computed at 1 or at 4
- Bad computation at 1 or 4 could be revealed only if they are used at 6
- Nodes (1,6) and (4,6) are *def-use (DU) pairs*
  - **defined** at 1,4
  - **used** at 6



Terms

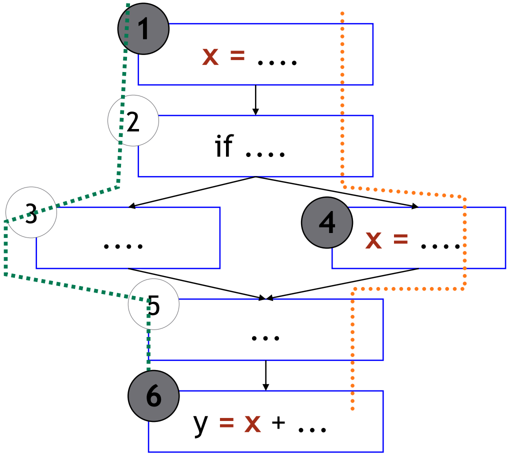
- **DU pair**: a pair of *definition* and *use* for some variable, such that at least one DU path exists from the definition to the use

  x = … is a *definition* of ***x***

  … = … ***x*** … is a *use* of ***x***

- **DU path**: a *definition-clear* path in the CFG from a definition to a use of the same variable
  - ***Definition-clear***: value is not updated on path
  - Loops may create infinite DU paths between a definition and a use

Definition-clear path



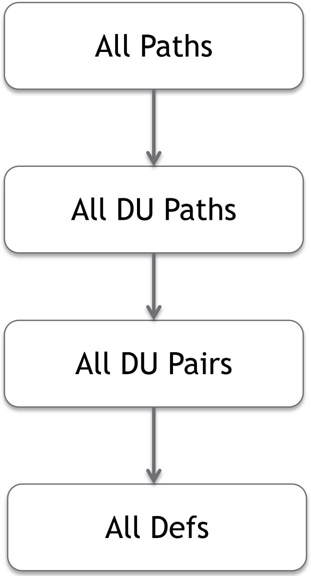
- 1,2,3,5,6 is a definition-clear path from 1 to 6
  - x is not updated between 1 and 6
- 1,2,4,5,6 is *not* a definition-clear path from 1 to 6
  - the value of x is ***killed*** (updated) at node 4
  - (1,6) is a DU pair because 1,2,3,5,6 is a definition-clear path

Adequacy criteria

- **All DU Pairs**: each DU pair is exercised by at least one test case
- **All DU Paths**: each *simple* (non looping) DU path (def-clear) is exercised by at least one test case
- **All Defs**: for each definition, there is at least one test case which exercises a DU pair containing it
  - *every computed value is used somewhere*

Corresponding coverage measures (fractions) can also be defined

Data flow testing subsumption



Difficult cases

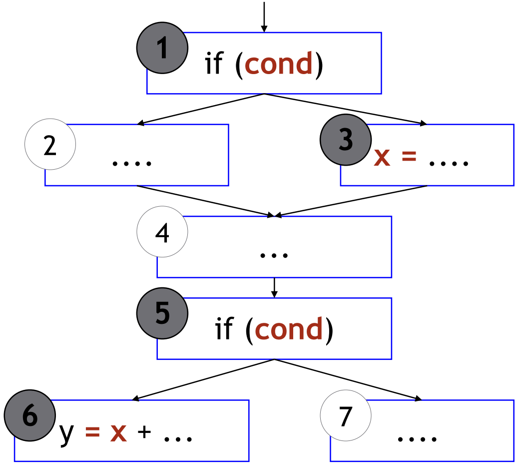
- `x[i] = … ; …; y = x[j]` def use?
  - DU pair (only) if `i==j`
- `x = … ; p = &x ; … ; *p = 99 ; … ; q = *p` def use
  - `*p` is an alias of `x`
- `m.putFoo(…); … ;` def
  `y = n.getFoo(…);` use?
  - Are `m` and `n` the same object?
  - Do `m` and `n` share a `foo` field?

Alias problem: which references are (always or sometimes) the same?

Data flow coverage with complex structures

- Arrays and pointers are critical for data flow testing
  - Under-estimation of aliases may fail to include some DU pairs
  - Over-estimation, on the other hand, may introduce unfeasible test obligations
- For testing, it may be preferable to accept under-estimation of alias set rather than over-estimation or expensive analysis
  - Controversial: in other applications (e.g., compilers), a *conservative* over-estimation of aliases is usually required
  - Alias analysis may rely on external guidance or other global analysis to calculate good estimates
  - Undisciplined use of dynamic storage, pointer arithmetic, etc. may make the whole analysis infeasible

Infeasibility in data flow analysis



- Suppose *cond* has not changed between 1 and 5
  - or the conditions could be different, but the first implies the second
- Then (3,6) is not a (feasible) DU pair
  - *but* it is difficult or impossible to determine which pairs are infeasible
- Infeasible test obligations are a problem
  - no test case can cover them

Coverage in data flow analysis

- In practice, reasonable coverage is (often, not always) achievable
  - Number of DU pairs is exponential in worst case, but often linear
  - All DU Paths is more often impractical

Data flow testing process

Similar to general structural testing…

- Draw an annotated CFG for the program
- Select the DF coverage criterion
- Identify paths in the CFG not covered according to the criterion
- Build test cases – determine the oracles
  - derive path condition expressions from the selected paths and solve those expressions to derive test inputs (involves constraint solving)
- Execute test cases

This may be a lot of work…

Data flow testing: key messages

- Data flow testing attempts to distinguish "important" paths: interactions between statements
  - intermediate between simple statement and branch coverage and more expensive path coverage
- Cover Def-Use (DU) pairs: from computation of value to its use
  - intuition: bad computed value is revealed only when it is used
  - Levels: All DU Pairs, All DU Paths, All Defs, …
- Limits: aliases, infeasible paths
  - Worst case is bad (undecidable properties, exponential blowup of paths), so pragmatic compromises are required