White Box Testing – Another Look

Motivation

- People are not perfect
  - Errors are made in design and code
- Goal of testing: given some code, uncover as many errors as possible
- Important and expensive activity:
  - May spend ~50% of total project effort on testing
  - For safety critical system cost of testing is several times higher than all other activities combined

A Way of Thinking

- Design and coding are creative activities
- Testing is destructive
  - Primary goal is to “break” the code
- Often same person does both coding and testing
  - Need “split personality”: when you start testing, become paranoid and malicious
  - This is surprisingly difficult: people don’t like to find out that they made mistakes.

Testing Objective

- **Testing:** a process of executing software with the intent of finding errors
- **Good testing:** high probability of finding as-yet-undiscovered errors
- **Successful testing:** discovers unknown errors

Basic Definitions

- **Test case:** specifies
  - Inputs + pre-test state of the software
  - Expected results (outputs an state)
- **Black-box testing:**
- **White-box testing:**

Testing Approaches

- We’ve been looking at a small sample of approaches for testing
- Black-box testing
  - Equivalence partitioning
- White-box testing
  - Control-flow-based testing
  - Loop testing
  - Data-flow-based testing

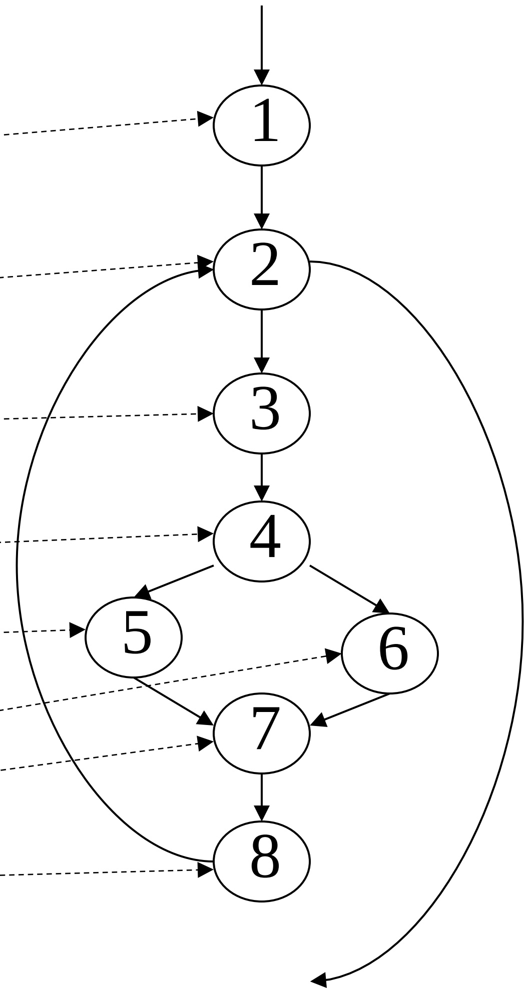
Control-flow Testing Revisited

Control-Flow-Based Testing

- A traditional form of white-box testing
- Step 1: From the source, create a graph describing the flow of control
  - Called the control flow graph (CFG)
  - The graph is created (extracted from the source code) manually or automatically
- Step 2: Design test cases to cover certain elements of this graph
  - Nodes, edges, paths

Example of a Control Flow Graph (CFG)

```
s:=0;
d:=0;
while (x<y) {
    x:=x+3;
    y:=y+2;
    if (x+y < 100)
        s:=s+x+y;
    else
        d:=d+x-y;
}
```



Elements of a CFG

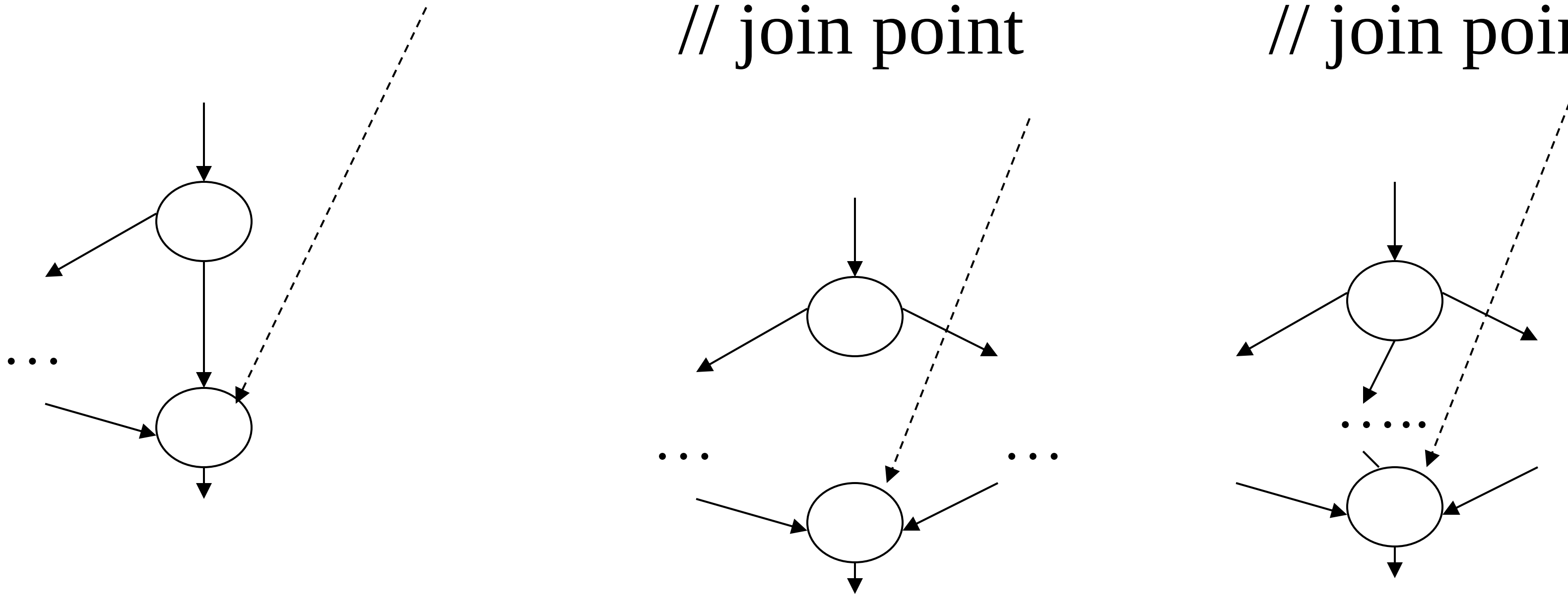
- Three kinds of nodes:
  - Statement nodes:
  - Predicate nodes:
  - Auxiliary nodes:
- Edges:
- It is relatively easy to map standard constructs from programming languages to elements of CFGs

IF-THEN, IF-THEN-ELSE, SWITCH

if (c)
  then
  // join point

if (c)
  then
  else
  // join point

switch (c)
  case 1:
  case 2:
  // join point

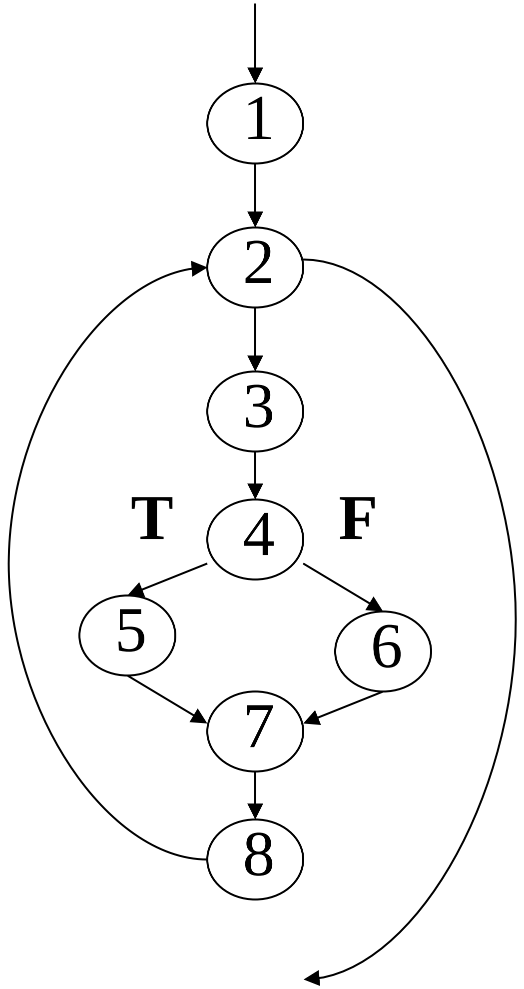


Statement Coverage

- Basic idea: given the control flow graph define a "coverage target" and write test cases to achieve it
- Traditional target: statement coverage
  - Need to write test cases that cover all nodes in the control flow graph
- Intuition: code that has never been executed during testing may contain errors

Example

- Suppose we write and execute two test cases
- Test case #1: follows path 1-2-exit (e.g., we never take the loop)
- Test case #2: 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit (loop twice, and both times take the true branch)
- Do we have 100% statement coverage?



Branch Coverage

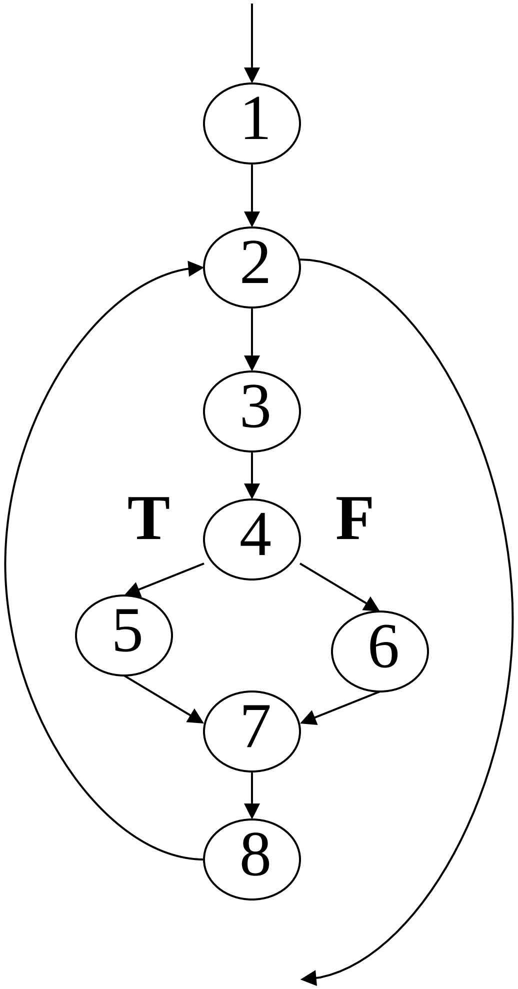
- Target: write test cases that cover all branches of predicate nodes
  - True and false branches of each IF
  - Two branches corresponding to the condition of a loop
  - All alternatives in a SWITCH statement
- In modern languages, branch coverage <u>implies</u> statement coverage

Branch Coverage

- Statement coverage does not imply branch coverage
  - Think of an example?
- Motivation for branch coverage:
  - experience shows that many errors occur in “decision making” (i.e., branching)
  - Plus, it subsumes statement coverage.

Example

- Same example as before
- Test case #1: follows path 1-2-exit
- Test case #2: 1-2-3-4-5-7-8-2-3-4-5-7-8-2-exit
- What is the branch coverage?

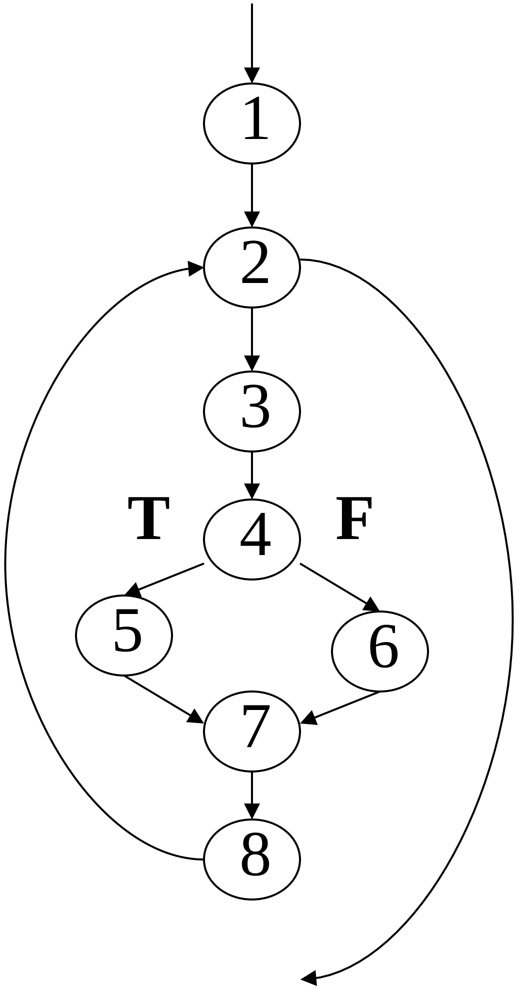


Achieving Branch Coverage

- For decades, branch coverage has been considered a necessary testing minimum
- To achieve it: pick a set of start-to-end paths (in the CFG) that cover all branches, and then write test cases to execute these paths
- It can be proven that branch coverage can be achieved with at most E-N+2 paths

Example

- First path: 1-2-exit (no execution of the loop)
- Second path: we want to include edge 2-3, so we can pick 1-2-3-4-5-7-8-2-exit
- What would we pick for the third path?



Determining a Set of Paths

- How do we pick a set of paths that achieves 100% branch coverage?
- Basic strategy:
  – Consider the current set of chosen paths
  – Try to add a new path that covers at least one edge that is not covered by the current paths
- The set of paths chosen with this strategy is called the “basic set”

Some Observations

- It may be impossible to execute some of the chosen paths from start-to-end.
  – Why?
  – Thus, branches should be executed as part of other chosen paths
- There are many possible sets of paths that achieve branch coverage

Loop Testing

- Branch coverage is not sufficient to test the execution of loops
  - It means two scenarios will be tested: the loop is executed zero times, and the loop is executed at least once
- Motivation for more testing of loops: very often there are errors in the boundary conditions
- Loop testing is a white-box technique that focuses on the validity of loops

Testing of Individual Loops

- Suppose that **m** is the **min** possible number of iterations, and **n** is the **max** possible number of iterations
- Write a test case that executes the loop **m** times and another one that executes it **m+1** times
- Write a test case that executes the loop for a “typical number” of iterations
- Write a test case that executes the loop **n-1** times and another one for **n** times

Testing of Individual Loops (cont.)

- If it is possible to have variable values that imply less than **m** iterations or more than **n** iterations, write test cases using those
- E.g., if we have a loop that is only supposed to process at most the 10 initial bytes from an array, run a test case in which the array has 11 bytes

Nested Loops

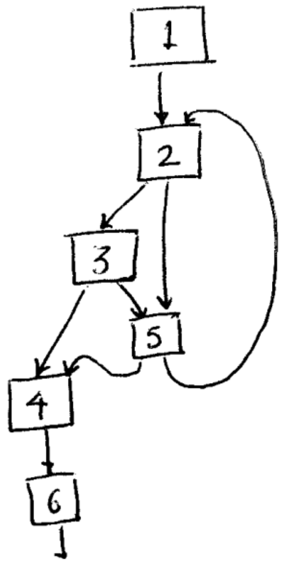
- Example: with 3 levels of nesting and 5 test cases for each level, total of 125 possible combinations: too many
- Start with the innermost loop do the tests (**m**,**m+1**, typical,), keep the other loops at their **min** number of iterations
- Continue toward the outside: at each level, do tests (**m,m+1**, typical,)
  - The inner loops are at typical values
  - The outer loops are at min values

Control-Flow Analysis

What is a loop ?

A subgraph of CFG with the following properties:

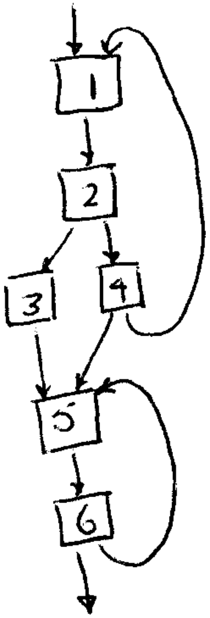
- Strongly Connected: there is a path from any node in the loop to any other node in the loop; and
- Single Entry: there is a single entry into the loop from outside the loop. The entry node of the loop is called the loop header.



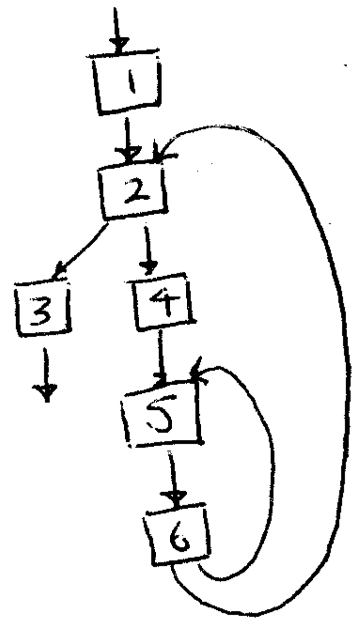
Loop nodes: 2, 3, 5
Header node: 2
Loop back edge: 5→2
Tail→Head

Property

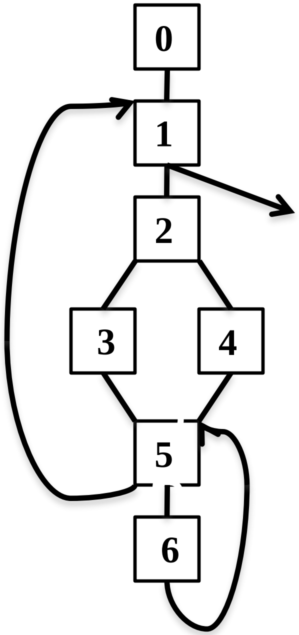
Given two loops: they are either disjoint or one is completely nested within the other.



Loops {1,2,4} and {5,6} are Disjoint.



Loop {5,6} is nested within loop {2,4,5,6}.



Loop {5,6} is nested within loop {1,2,3,4,5,6}.

Identifying Loops

Definitions:

Dominates: node n dominates node m iff all paths from start node to node m pass through node n, i.e. to visit node m we must first visit node n.

A loop has

- A single entry →
- A back edge, an edge A→B .

Identifying Loops

Algorithm for finding loops:

1. Compute Dominator Information.
2. Identify Back Edges.
3. Construct Loops corresponding to Back Edges.

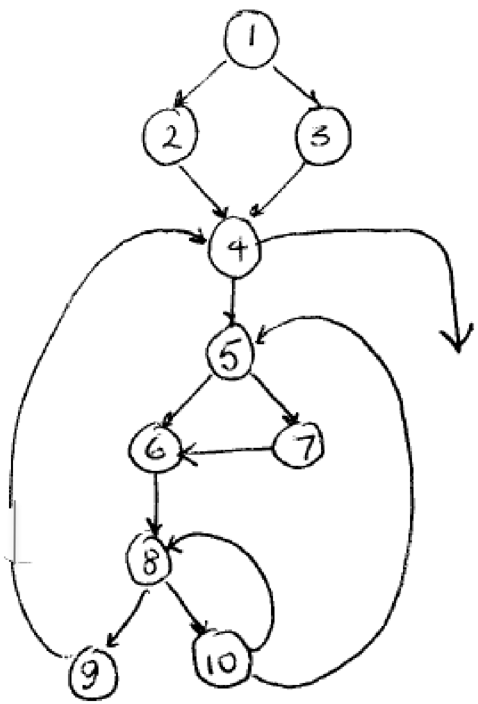
Dominators: *Characteristics*

1. Every node dominates itself.
2. Start node dominates every node in the flow graph.
3. If N *DOM* M and M *DOM* R then N *DOM* R.
4. If N *DOM* M and O *DOM* M then

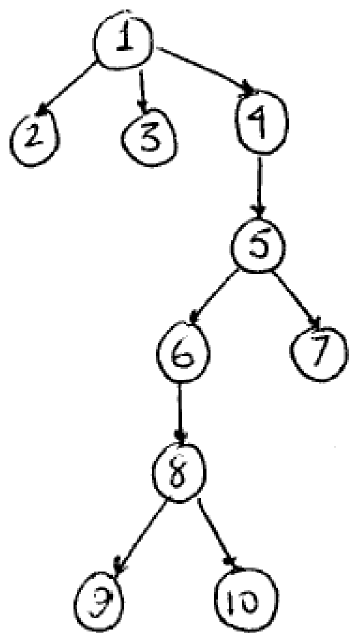
   either N *DOM* O or O *DOM* N
5. Set of dominators of a given node can be linearly ordered according to dominator relationships.

Dominators: *Characteristics*

6. Dominator information can be represented by a Dominator Tree. .



CFG

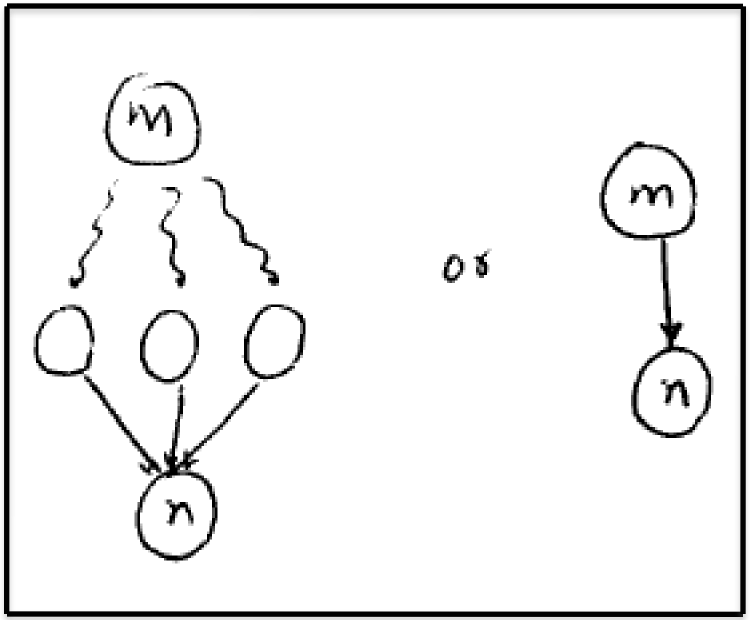


Dominator Tree

1 is the immediate dominator of 2, 3 & 4

Computing Dominator Sets

Observation: node m donimates node n iff m dominates all predecessors of n.



Let D(n) = set of dominators of n

$$D(n) = \{n\} \cup \bigcap_{p \in pred(n)} D(p)$$

Where Pred(n) is set of immediate predecessors of n in the CFG

Computing Dominator Sets

Initial Approximation:

$D(n_o) = \{n_o\}$

n_o is the start node.

$D(n) = N$, for all $n != n_o$

N is set of all nodes.

Iteratively Refine D(n)'s:

$$D(n) = \{n\} \cup \bigcap_{p \in pred(n)} D(p)$$

Algorithm:

$D(n_0) = \{n_0\}$

for all $n \in N$ st $n \neq n_0$ do $D(n) = N$

while changes to any $D(n)$ occur do

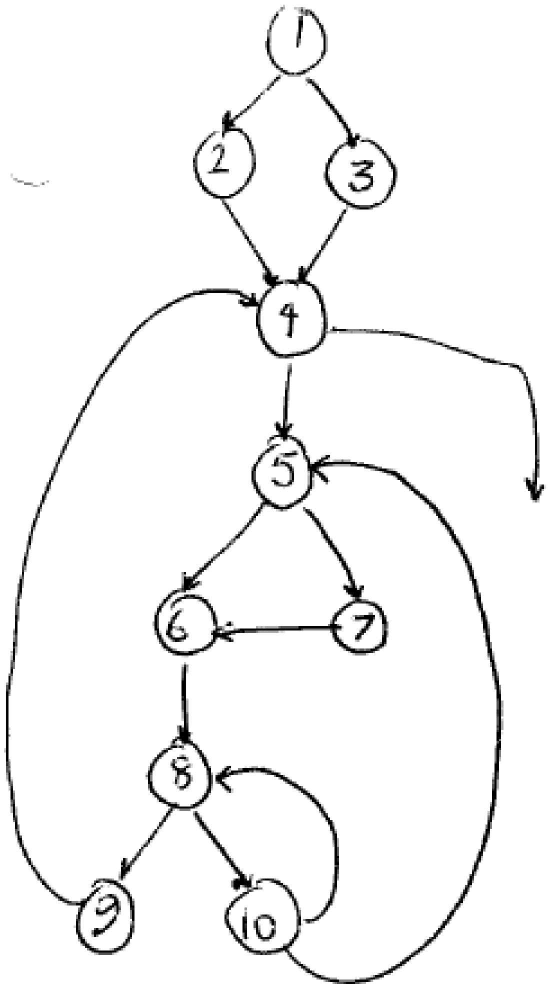
  for n in $N - \{n_0\}$ do

    $D(n) = \{n\} \cup \bigcap_{p \in pred(n)} D(p)$

  endfor

endwhile.

Example: Computing Dom. Sets



D(1) = {1}
D(2) = {2} U D(1) = {1,2}
D(3) = {3} U D(1) = {1,3}
D(4) = {4} U (D(2) ∩ D(3) ∩ D(9)) = {1,4}
D(5) = {5} U (D(4) ∩ D(10)) = {1,4,5}
D(6) = {6} U (D(5) ∩ D(7)) = {1,4,5,6}
D(7) = {7} U D(5) = {1,4,5,7}
D(8) = {8} U (D(6) ∩ D(10)) = {1,4,5,6,8}
D(9) = {9} U D(8) = {1,4,5,6,8,9}
D(10)= {10} U D(8) = {1,4,5,6,8,10}

Back Edges: 9→4, 10→8, 10→5

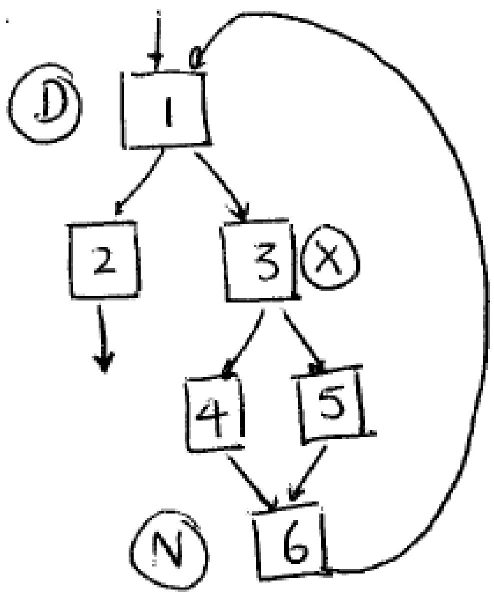
Loop

Given a back edge N → D

Loop corresponding to edge N → D

= {D} +

{X st X can reach N without going through D}



1 dominates 6

⇒ 6→1 is a back edge

Loop of 6→1

= {1} + {3,4,5,6}

= {1,3,4,5,6}

Algorithm for Loop Construction

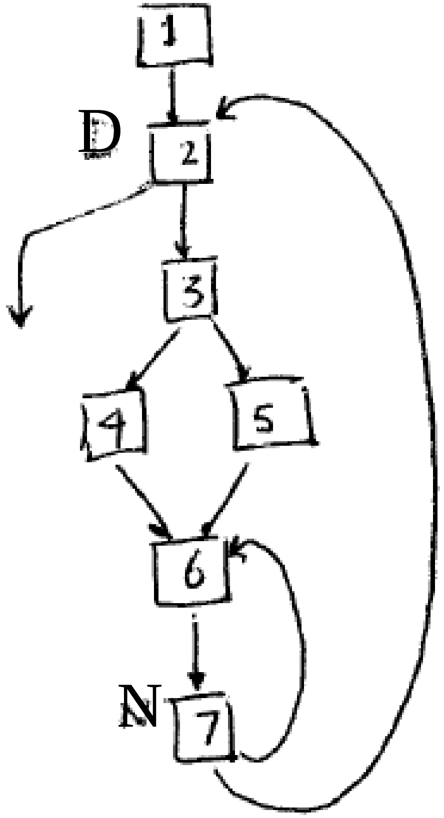
Given a Back Edge N→D

```
Stack = empty
Loop = {D}
Insert(N)
While stack not empty do
    pop m – top element of stack
    for each p in pred(m) do
         Insert(p)
    endfor
Endwhile
```

```
Insert(m)
  if m not in Loop then
    Loop = Loop U {m}
    push m onto Stack
  endif
End Insert
```

Example

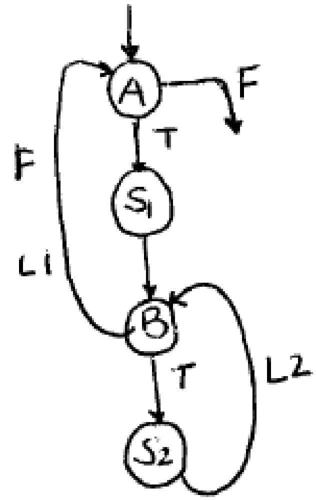
Back Edge 7→2



Loop = {2} + {7} + {6} + {4} + {5} + {3}

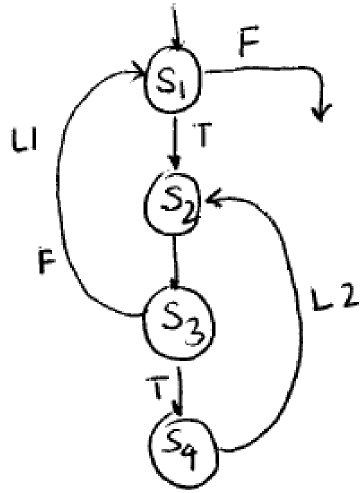
Stack = ~~7~~ ~~6~~ ~~4~~ ~~5~~ ~~3~~

Examples



L2 → B, S2
L1 → A,S1,B,S2
L2 nested in L1

```
While A do
    S1
    While B do
        S2
    Endwhile
Endwhile
```



L1 → S1,S2,S3,S4
L2 → S2,S3,S4
L2 nested in L1

?

Reducible Flow Graph

The edges of a reducible flow graph can be partitioned into two disjoint sets:

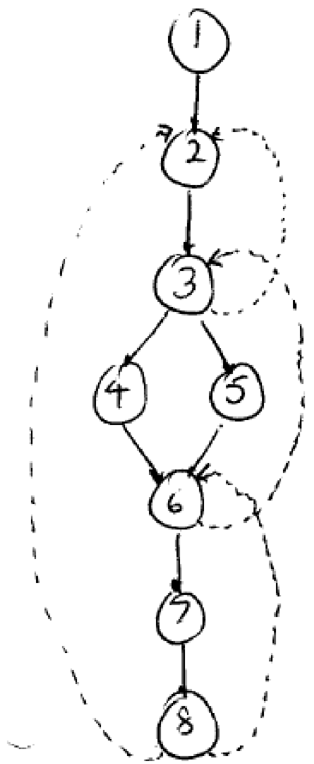
- *Forward* – from an acyclic graph in which every node can be reached from the initial node.
- *Back* – edges whose heads (sink) dominate tails (source).

Any flow graph that cannot be partitioned as above is a non-reducible or irreducible.

Reducible Flow Graph

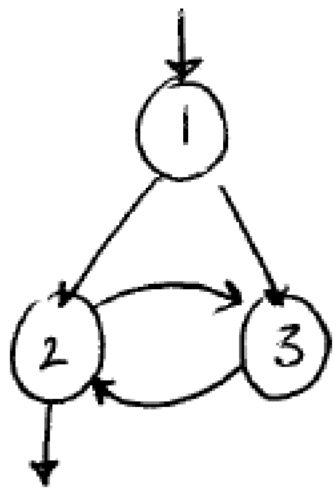
How to check reducibility ?

– Remove all back edges and see if the resulting graph is acyclic.



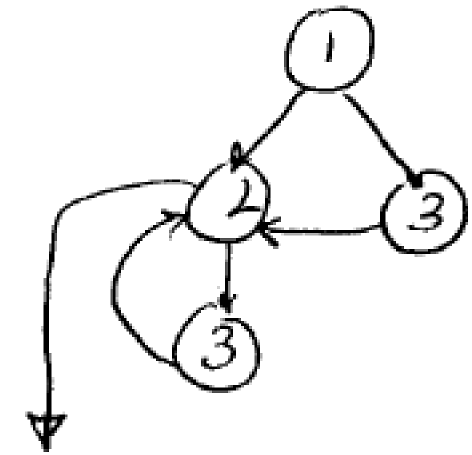
Reducible

Irreducible



2→3 not a back edge
3→2 not a back edge
graph is not acyclic

Node Splitting

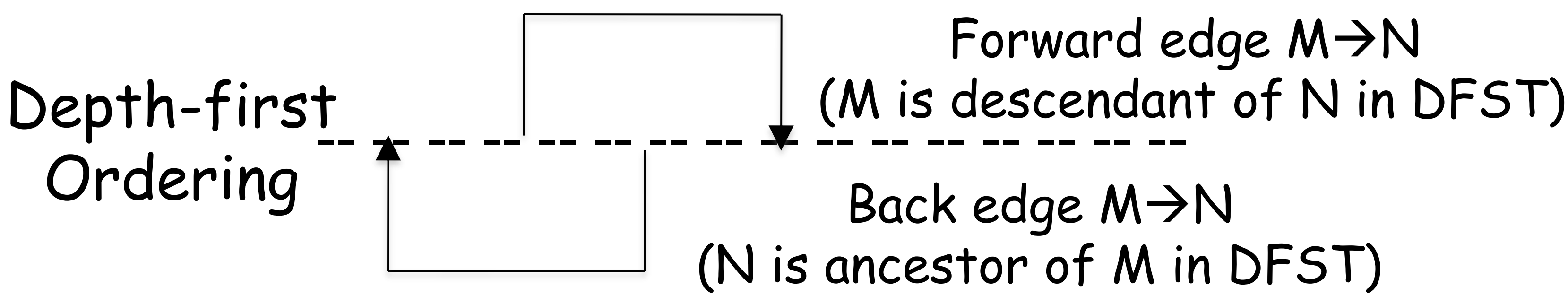


Converts irreducible
to reducible

Loop Detection in Reducible Graphs

Depth-first Ordering: numbering of nodes in the reverse order in which they were last visited during depth first search.

M→N is a back edge iff DFN(M) >= DFN(N)



Sample Problems

Control Flow Analysis