Graph coverage

In his landmark

book *Software Testing Techniques*, Beizer wrote that testing is simple –

All a tester needs to do is “find a graph and cover it.”

Structural Graph Coverage for Source Code

To apply one of the graph coverage criteria, the first step is to define a **graph abstraction of the source code.**

The graph is not the same as the source code (**Abstraction**); details are removed.

Source code has typically several useful, but nonetheless quite different, graph abstractions.

Most common graph abstractions:

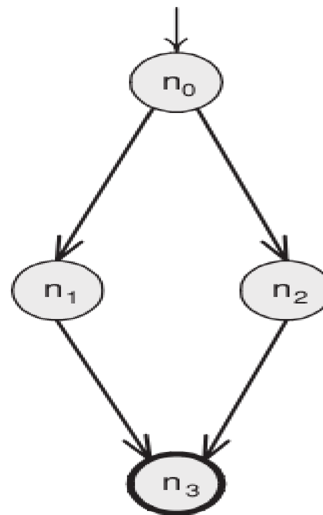
- **Control Flow Graph**
- **Data Flow Graph**

A graph based coverage criterion evaluates a test set in terms of how the paths corresponding to the test cases “cover” the graph abstraction.

Graph: Basic definitions

A graph G =(N, N_0 , N_f , E) is

- a set N of *nodes*
- a set N_0 of *initial nodes*, where N_0 is a non empty subset of N
- a set N_f of *final nodes*, where N_f is a non empty subset of N
- a set E of *edges*, where E is a subset of $N \times N$



Structural Graph Coverage for Source Code

Graph abstractions of the source code.

Control Flow graph (CFG) which associates

- an **edge** with each possible branch in the program
- a **node** with sequences of statements.

Data Flow Graph (DFG) a control flow graph annotated with locations where variable are used (**Def** and **Use**)

- **Def**: a location where a value for a variable is stored
- **Use**: a location where a variable value is accessed.

Constructing a control flow graph (CFG)

Basic Block:

a **maximum sequence of program statements** such that if any one statement of the block is executed, all statements in the block are executed. A basic block has only one entry point and one exit point.

Conditional instruction **if-then-else**

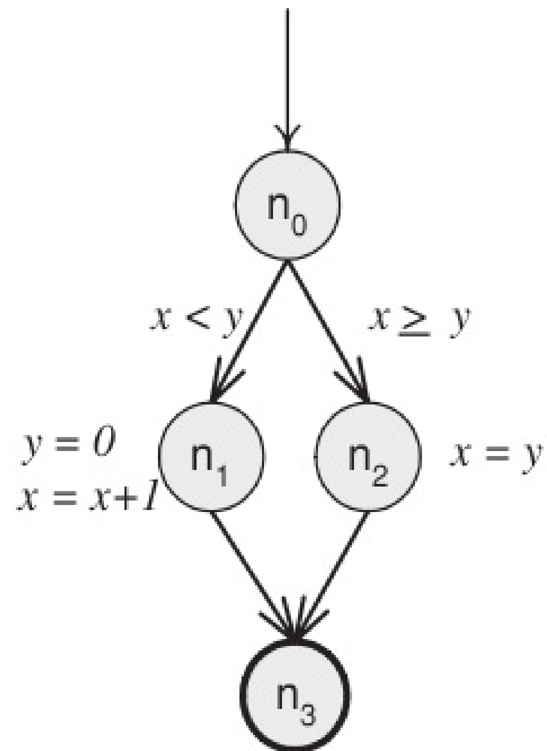
- **A decision node** for the conditional test

Two edges for true/false evaluation of the contion test

- Two graphs for then/else instructions
- **A junction node** for joining control from then/else instructions

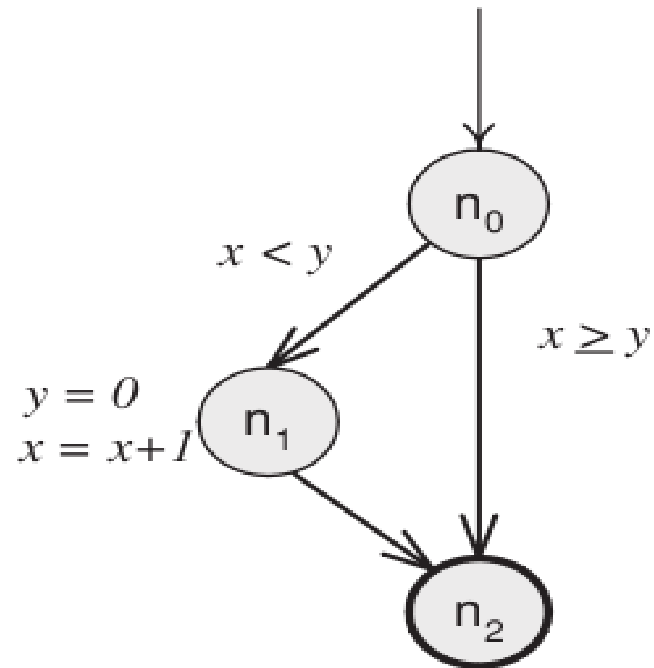
CFG fragment for a if-then-else

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
```



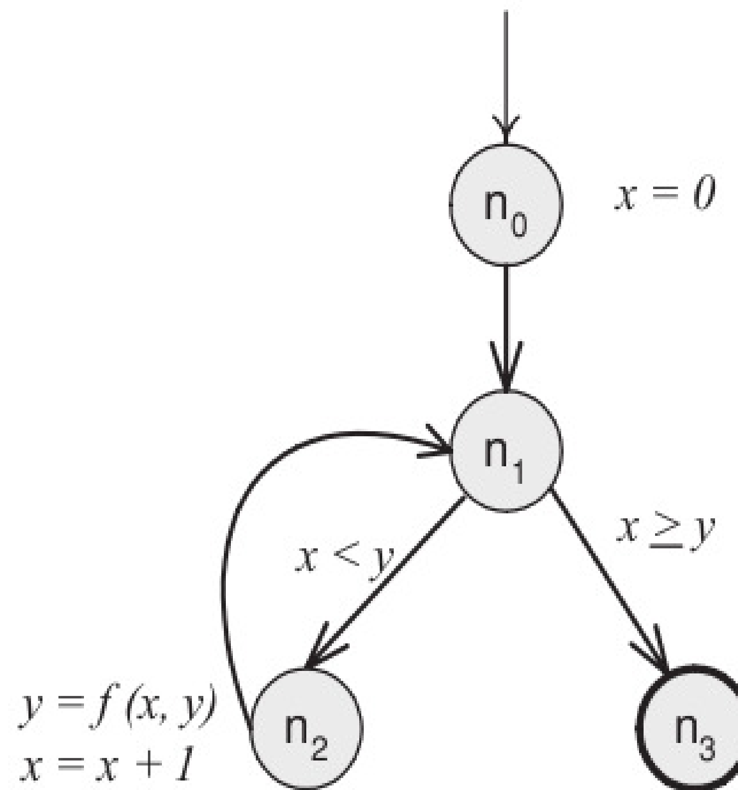
CFG fragment for a if-then

```
if ( x < y )
{
   y = 0;
   x = x + 1;
}
```



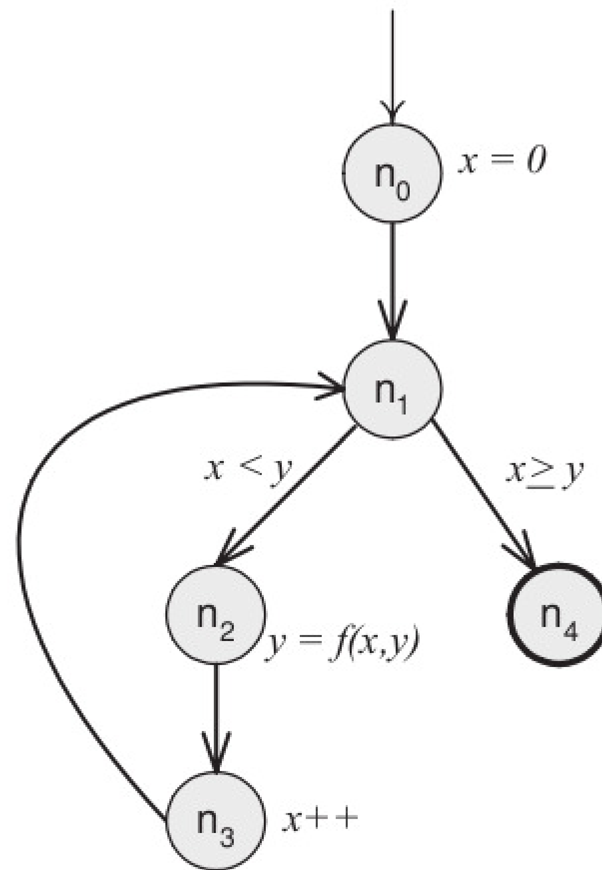
CFG fragment for a while loop

```
x = 0;
while (x < y)
{
   y = f (x, y);
   x = x+1;
}
```



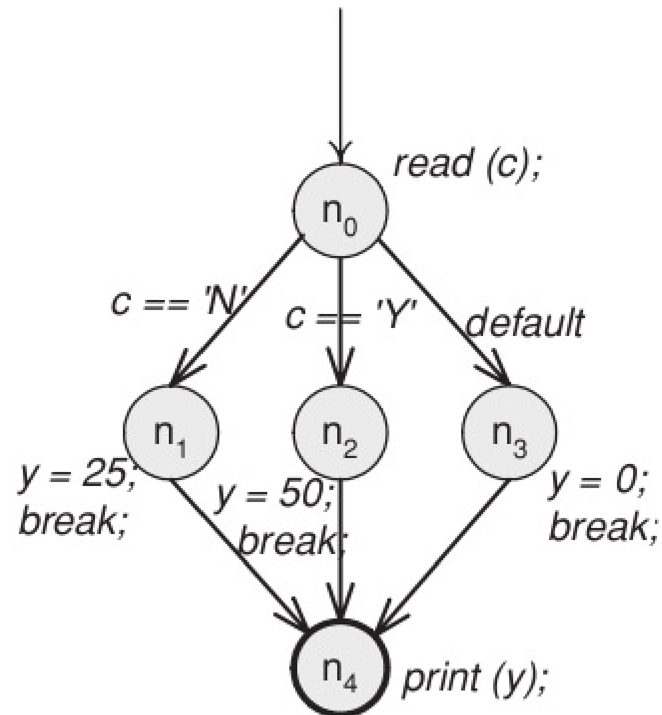
CFG fragment for the for loop

```
for (x = 0; x < y; x++)
{
  y = f(x,y);
}
```



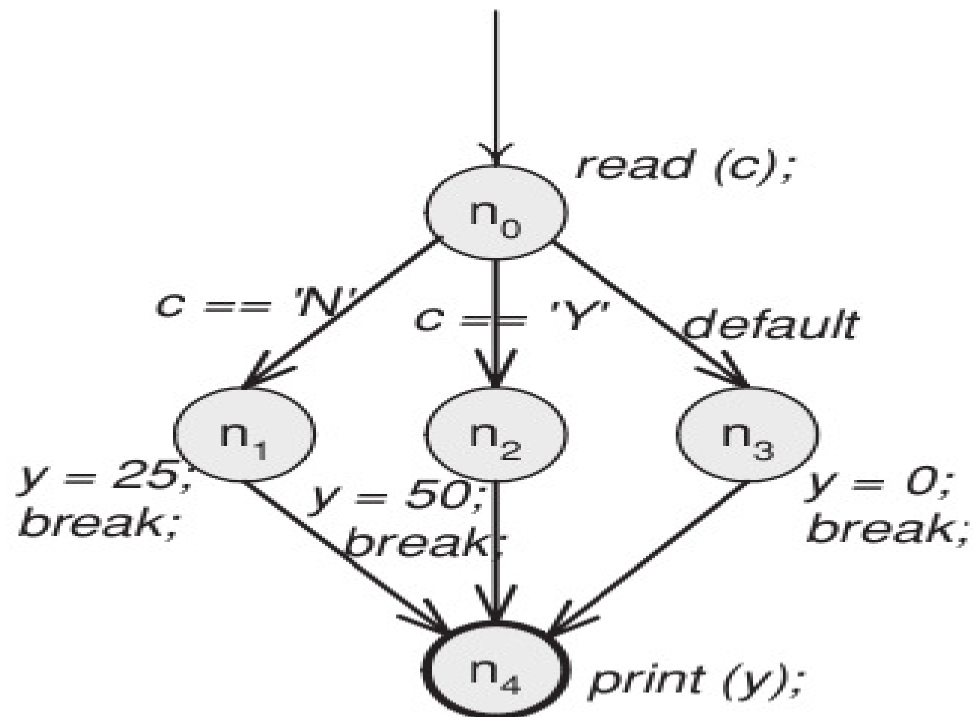
CFG fragment for the case instr.

```
read (c);
switch (c)
{
case 'N':
      y = 25;
      break;
case 'Y':
      y = 50;
      break;
default:
      y = 0;
      break;
}
print (y);
```



CFG fragment for the case instr.

```
read (c);
switch (c)
{
case 'N':
      y = 25;
      break;
case 'Y':
      y = 50;
      break;
default:
      y = 0;
      break;
}
print (y);
```



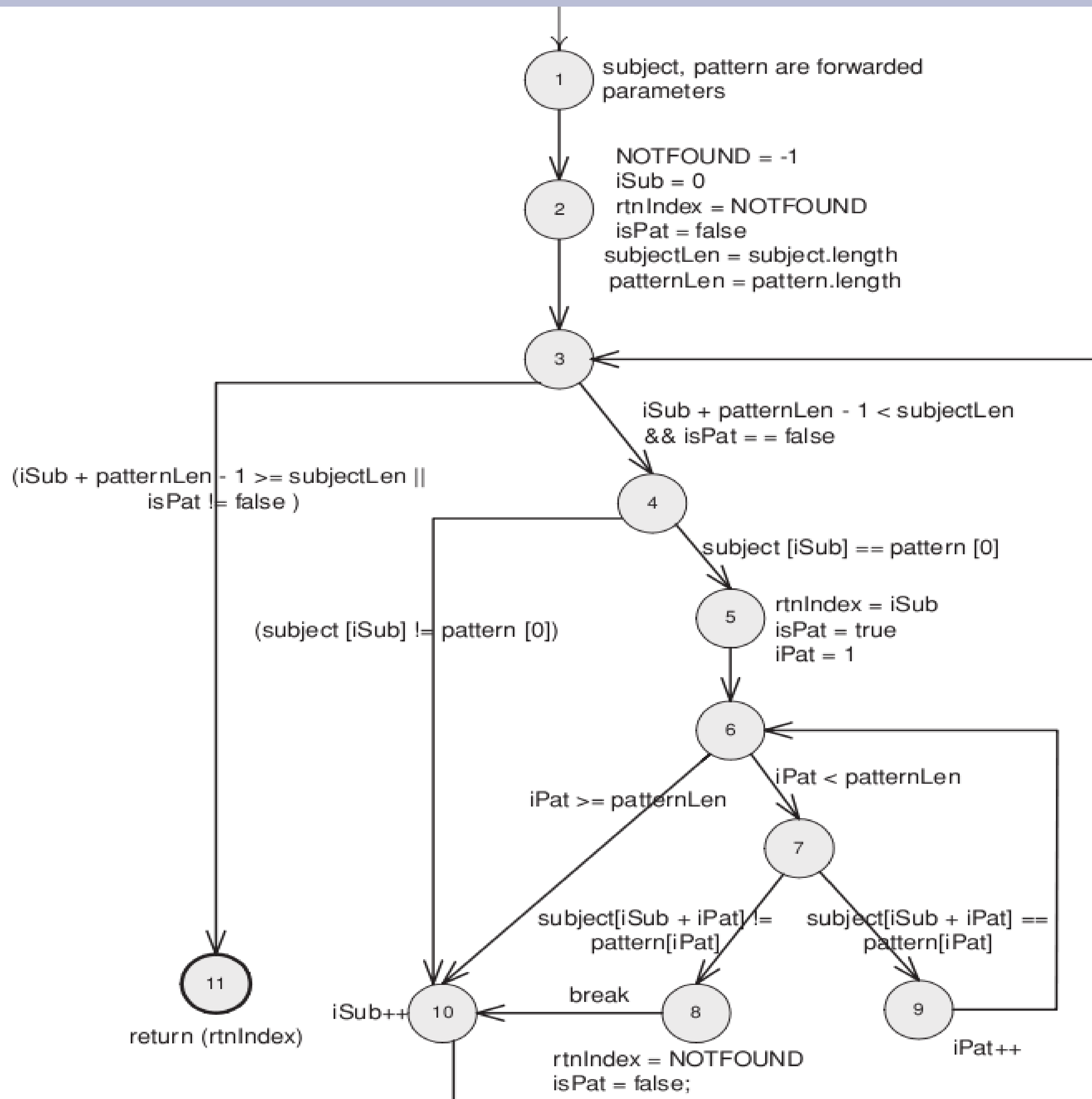
Example: source code

```
public int pat (char[] subject, char[] pattern)
{
// Post: if pattern is not a substring of subject, return -1
//       else return (zero-based) index where the pattern (first)
//       starts in subject

  final int NOTFOUND = -1;
  int  iSub = 0, rtnIndex = NOTFOUND;
  boolean isPat  = false;
  int subjectLen = subject.length;
  int patternLen = pattern.length;

  while (isPat == false && iSub + patternLen - 1 < subjectLen)
  {
   if (subject [iSub] == pattern [0])
   {
    rtnIndex = iSub; // Starting at zero
    isPat = true;
    for (int iPat = 1; iPat < patternLen; iPat ++)
    {
     if (subject[iSub + iPat] != pattern[iPat])
     {
      rtnIndex = NOTFOUND;
      isPat = false;
      break;  // out of for loop
     }
    }
   }
   iSub ++;
  }
  return (rtnIndex);
}
```

Example: CFG



Basic definitions on a graph $G = (N, N_0, N_f, E)$

path: a sequence p= $[n_1, n_2, \ldots, n_M]$ of nodes, where each pair of adjacent nodes, (n_i, n_{i+1}), $1 \leq i < M$, is in the set E of edges.

length of a path: the number of edges it contains.

subpath of a path p: is a subsequence of p (possibly p itself).

Reachability:

a node n (or an edge e) is *syntactically reachable* from node n_i if there exists a path from node n_i to n (or edge e).

a node n (or an edge e) is *semantically reachable* from node n_i if it is syntactically reachable and it is possible to execute (with some input) at least one of the path from node n_i to n (or edge e).

Reachability on a graph $G = (N, N_0, N_f, E)$

Reach$_G(n_i)$: the subgraph of G that is syntactically reachable from node n_i

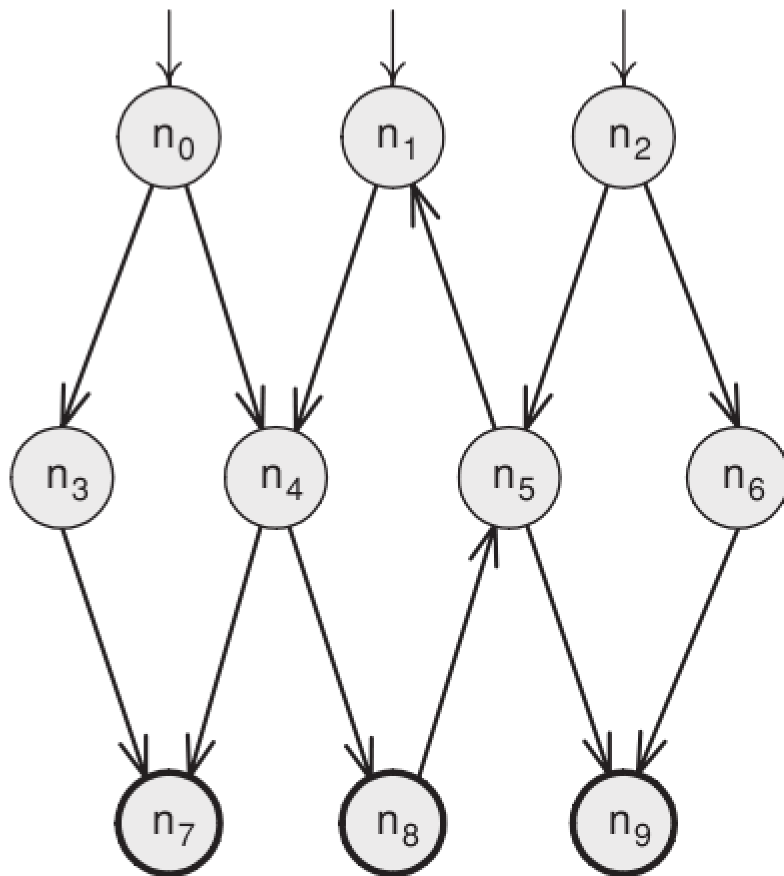
Reach$_G(N_0)$: the subgraph of G that is syntactically reachable from some initial node in N_0

Some graphs have nodes that cannot be syntactically reached from any of the initial nodes N_0.

These graphs frustrate attempts to satisfy a coverage criterion, so the attention is usually restricted to *reachG*(N_0).

Basic graph algorithms, usually given in standard data structures texts, can be used to compute syntactic reachability.

Example



Path Examples	
1	n_0, n_3, n_7
2	n_1, n_4, n_8, n_5, n_1
3	n_2, n_6, n_9

Invalid Path Examples	
1	n_0, n_7
2	n_3, n_4
3	n_2, n_6, n_8

(a) Path examples

Reachability Examples	
1	*reach* $(n_0) = N - \{n_2, n_6\}$
2	*reach* $(n_0, n_1, n_2) = N$
3	*reach* $(n_4) = \{n_1, n_4, n_5, n_7, n_8, n_9\}$
4	*reach* $([n_6, n_9]) = \{ n_9 \}$

(b) Reachability examples

Test path on a graph $G = (N, N_0, N_f, E)$

Test path: A path *p*, possibly of length zero, that starts at some node in N_0 and ends at some node in N_f.

Test paths must start in N_0 because test cases always begin from an initial node.

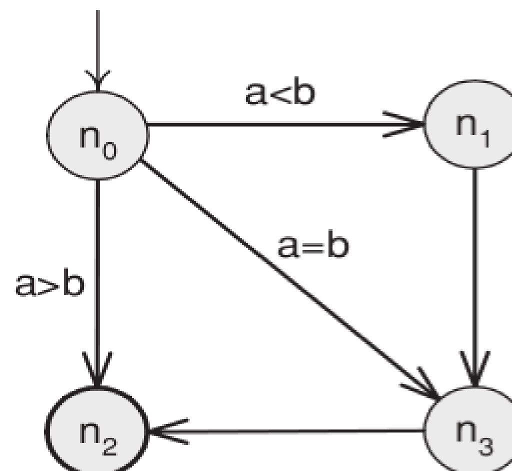
A single test path may correspond to many test cases.

A test path may correspond to zero test cases if the test path is infeasible.

If the program is deterministic, a single test case **t** corresponds to a single test path denoted by **$path_G(t)$.**

For a test set **T, pathG(T)={$path_G(t)$ | t in T}**

Example: mapping of test cases against test paths



(a) Graph for testing the case with input integers a, b and output (a+b)

	Map to	
Test case t_1 : (a=0, b=1)	→	[Test path p_1 : n_0, n_1, n_3, n_2]
Test case t_2 : (a=1, b=1)	→	[Test path p_2 : n_0, n_3, n_2]
Test case t_3 : (a=2, b=1)	→	[Test path p_3 : n_0, n_2]

Graph coverage criteria

For any graph-based coverage criterion, the idea is to identify the test requirements in terms of **various structures in the graph**

Criteria diveded in two type depending on the graph type:

- ***structural graph coverage criteria*** *(on Control Flow Graph)*
- ***data flow coverage*** **criteria** *(on Data Flow Graph)*

Coverage criteria define test requirements, *TR*, in terms of properties of test paths in a graph *G*. A test requirement is *met* by ***visiting*** **a particular node** or edge or by **to*uring*** **a particular path**

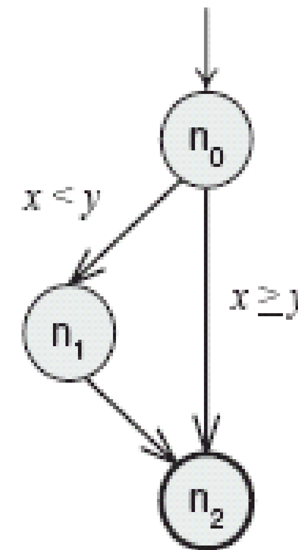
Graph Coverage*:* Given a set *TR* of test requirements for a graph criterion *C*, a test set *T* satisfies *C* on graph *G* if and only if for every test requirement *tr* in *TR*, there is at least one test path *p* in *path*(*T*) such that *p* meets *tr* .

Structural criteria: node coverage

(Variously called) "statement coverage," "block coverage," "state coverage," and "**node coverage**."

Node Coverage (NC)*:* Test set *T* satisfies node coverage on graph *G* if and only if for every syntactically reachable node *n* in *N*, there is some path *p* in *path*(*T*) such that *p* visits *n*.

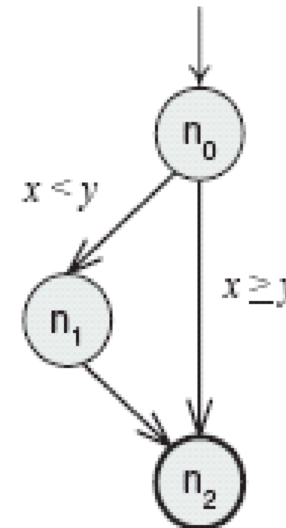
Node coverage is implemented in many commercial testing tools, most often in the form of statement coverage.



$path(t_1) = [\, n_0, n_1, n_2 \,]$
$path(t_2) = [\, n_0, n_2 \,]$

Structural criteria: edge coverage

(Variously called) “**edge coverage**,” “branch coverage,”

Edge Coverage (NC)*:* Test set *T* satisfies edge coverage on graph *G* if and only if for every path p in G to a syntactically reachable node, of length up to 1, inclusive, *p* in *path*(*T*). Edge coverage **subsumes** node coverage.	

Structural criteria: round trip coverage

Start the software in some state and then follow transitions so that the last state is the same as the start state.

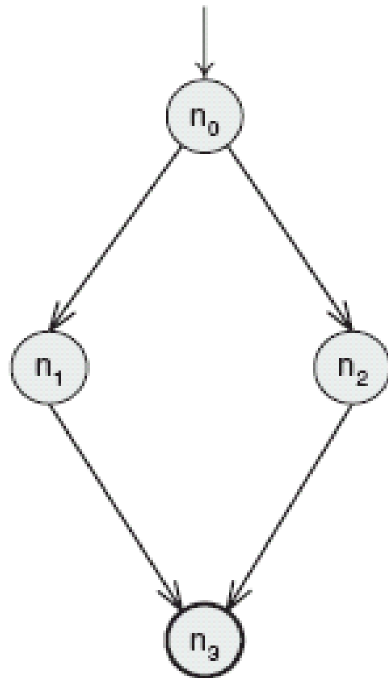
Testing used to verify that the system is not changed by certain inputs.

Simple path: A path from node n_i to node n_j is ***simple*** if no node appears more than once in the path, with the exception that the first and last nodes may be identical.

- Simple paths do not contain any proper subpath which is a loop
- A simple path may be a loop
- Any path can be composed by means of simple paths.

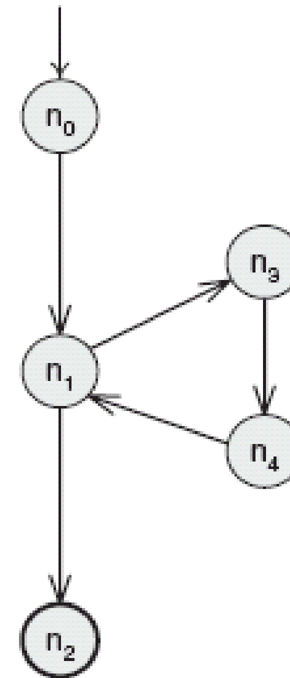
Prime path: A simple path of maximal length (it does not appear as a proper subpath of any other simple path)

Example of prime paths



Prime Paths = { $[n_0, n_1, n_3]$, $[n_0, n_2, n_3]$ }
path (t_1) = $[n_0, n_1, n_3]$
path (t_2) = $[n_0, n_2, n_3]$
$T_1 = \{t_1, t_2\}$
T_1 satisfies prime path coverage on the graph

(a) Prime Path Coverage on a Graph with No Loops



Prime Paths = { $[n_0, n_1, n_2]$,
$[n_0, n_1, n_3, n_4]$, $[n_1, n_3, n_4, n_1]$,
$[n_3, n_4, n_1, n_3]$, $[n_4, n_1, n_3, n_4]$,
$[n_3, n_4, n_1, n_2]$ }
path (t_3) = $[n_0, n_1, n_2]$
path (t_4) = $[n_0, n_1, n_3, n_4, n_1, n_3, n_4, n_1, n_2]$
$T_2 = \{t_3, t_4\}$
T_2 satisfies prime path coverage on the graph

(b) Prime Path Coverage on a Graph with Loops

Structural criteria: prime path coverage

Prime Path Coverage (PPC): for every prime path p in G, *p* in *path*(*T*) (TR contains every prime path).

Prime path coverage has two special cases:

A *round trip* path is a prime path of nonzero length that starts and ends at the same node.

Simple Round Trip Coverage (SRTC): *TR contains at least one round-trip path for each reachable node in G that begins and ends a round-trip path.*

Complete Round Trip Coverage (CRTC): *TR contains all roundtrip paths for each reachable node in G.*

Structural criteria: path coverage

Complete Path Coverage (CPC): *TR contains all paths in G.*

If a graph has a cycle an infinite number of paths are possible, and hence there are an infinite number of test requirements (**complete path coverage is not feasible**).

A **finite approximation** of this criterion is, however, useful where a set of path are given in a parametric way (for ex., these paths given by a customer in the form of usage scenarios).

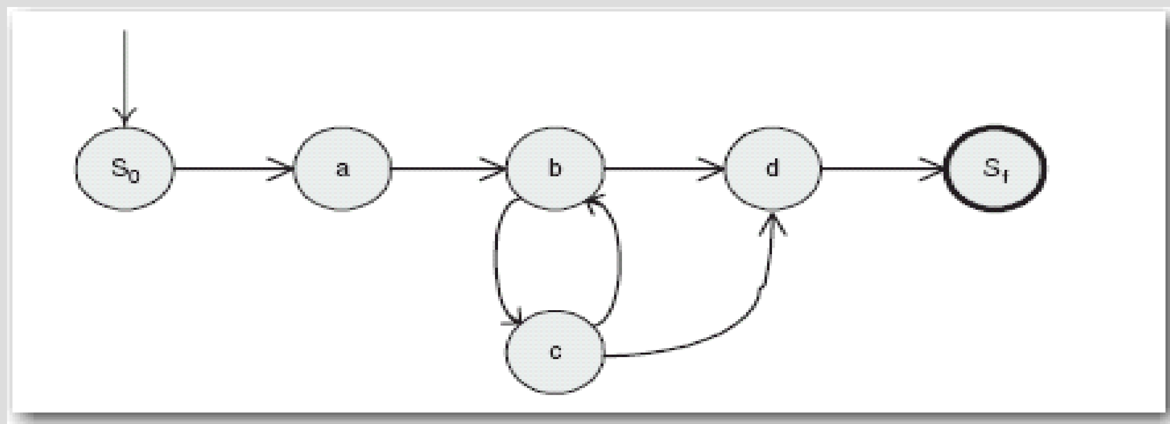
Specified Path Coverage (SPC): *TR contains a set S of test paths, where S is supplied as a parameter.*

Structural criteria: Touring problems

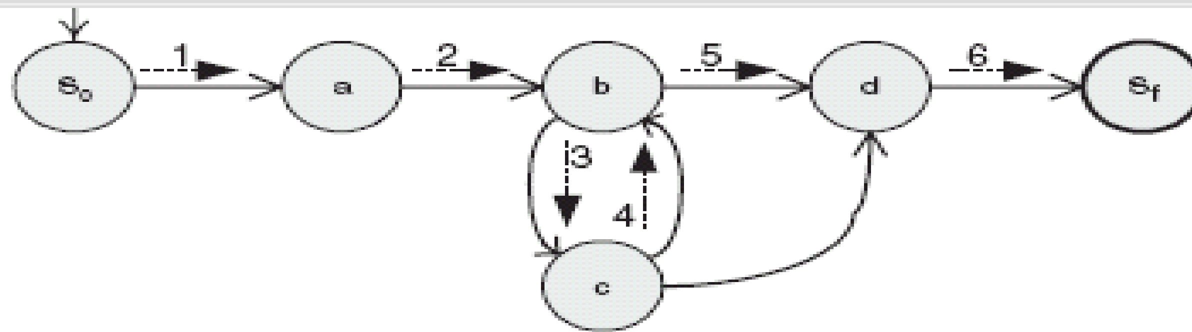
Simple paths do not have internal loops, whereas test paths that tour a simple path may have loops.

Requiring that the simple path is a subpath of the step path may result in a too strong requirement.

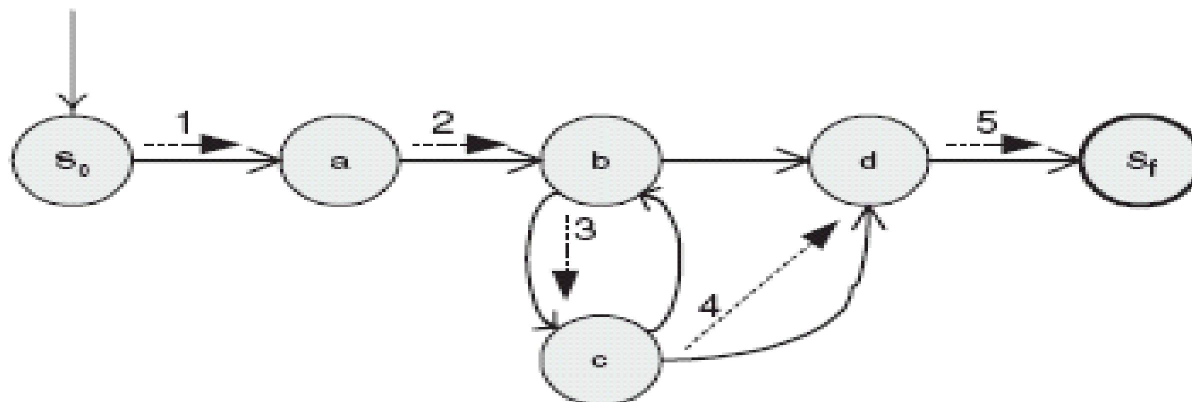
Ex. If it required to tour the **subpath** $q = [a, b, d]$, the requirement is not net by any path containing c, as $p = [s_0, a, b, c, b, d, s f]$



Tour weakening: sidetrips and detours



(a) Graph being toured with a sidetrip



(b) Graph being toured with a detour

Tour weakening: sidetrips and detours

***Tour**:* Test path *p* is said to *tour* subpath *q* if and only if *q* is a subpath of *p*.

***Tour with Sidetrips**:* Test path *p* is said to *tour* subpath *q with sidetrips* if and only if every **edge** in *q* is also in *p* in the same order.

***Tour with Detours**:* Test path *p* is said to *tour* subpath *q with detours* if and only if every **node** in *q* is also in *p* in the same order.

A graph coverage criterion can be specialized with a choice of touring.

For ex., prime path coverage could be defined **strictly** in terms of tours, **less strictly** to allow sidetrips, or **even less strictly** to allow detours.

Data flow criteria

Testing criteria based on the assumption that to test a program adequately, one should **focus on the flows of data values**:

try to ensure that the **values are created and used correctly**.

A ***definition (def)*** is a location where a value for a variable is stored into memory

A ***use*** is a location where a variable's value is accessed.

Data flow testing criteria use the fact that values are carried from defs to uses: ***du-pairs***

The idea of data flow criteria is to **exercise du-pairs** in various ways over the data flow graph (a control flow graph with nodes and edges annotated with sets of def and use variables).

Constructing a Data Flow Graph (DFG)

Data flow graph (DFG) a control flow graph annotated with locations where variable are accessed (Def and Use)

A **Def for a variable X** is a location in the program where a value for X is stored into memory (assignment, input, etc.):

- X appears on the left side of an assignment statement
- X is an actual parameter in a call site and its value is changed within the method
- X is a formal parameter of a method (an implicit def when the method begins execution)
- X is an input to the program

If a variable has multiple definitions in a single basic block, **the last definition is the only one that is relevant** to data flow analysis.

Constructing a Data Flow Graph (DFG)

A **Use** may occur for a variable X in the following situations:

X appears on the right side of an assignment statement

X appears in a conditional test (note that such a test is always associated with at least two edges)

X is an actual parameter to a method

X is an output of the program

X is an output of a method in a Return statement or returned as a parameter

DFG: local and global Use

Example: Y = Z; X = Y + 2;

The use of Y is called **local use**; it is impossible for a def in another basic block to reach the use in **X = Y + 2**.

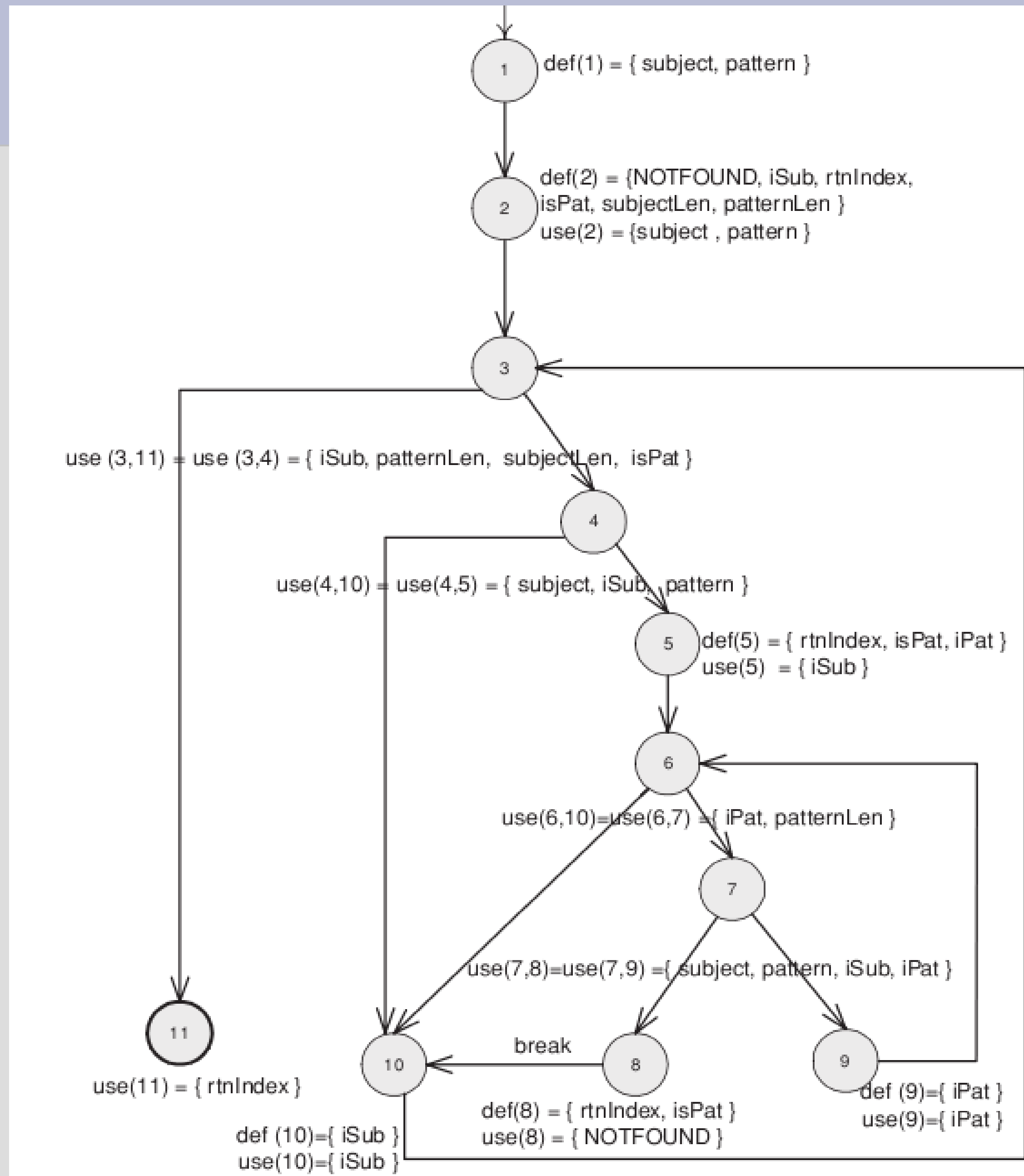
The use of **Z** is called **global use** because the definition of **Z** used in this basic block must originate in some other basic block.

Data flow analysis only considers global uses.

Defs and Uses at each node in the CFG for TestPat

node	def	use
1	{subject, pattern}	
2	{NOTFOUND, isPat, iSub, rtnIndex, subjectLen, patternLen}	{subject, pattern}
3		
4		
5	{rtnIndex, isPat, iPat}	{iSub}
6		
7		
8	{rtnIndex, isPat}	{NOTFOUND}
9	{iPat}	{iPat}
10	{iSub}	{iSub}
11		{rtnIndex}

DFG: Annotated CFG for TestPath with Defs and Uses



CF and DF Graphs

1 subject, pattern are forwarded parameters

2
NOTFOUND = -1
iSub = 0
rtnIndex = NOTFOUND
isPat = false
subjectLen = subject.length
patternLen = pattern.length

3
iSub + patternLen - 1 < subjectLen && isPat = = false
(iSub + patternLen - 1 >= subjectLen || isPat != false)

4
subject [iSub] == pattern [0]
(subject [iSub] != pattern [0])

5
rtnIndex = iSub
isPat = true
iPat = 1

6
iPat < patternLen
iPat >= patternLen

7
subject[iSub + iPat] != pattern[iPat]
subject[iSub + iPat] == pattern[iPat]

8
break
rtnIndex = NOTFOUND
isPat = false;

9
iPat++

10
iSub++

11
return (rtnIndex)

---

1 def(1) = { subject, pattern }

2
def(2) = {NOTFOUND, iSub, rtnIndex, isPat, subjectLen, patternLen }
use(2) = {subject , pattern }

3
use (3,11) = use (3,4) = { iSub, patternLen, subjectLen, isPat }

4
use(4,10) = use(4,5) = { subject, iSub, pattern }

5
def(5) = { rtnIndex, isPat, iPat }
use(5) = { iSub }

6
use(6,10)=use(6,7) ={ iPat, patternLen }

7
use(7,8)=use(7,9) ={ subject, pattern, iSub, iPat }

8
break
def(8) = { rtnIndex, isPat }
use(8) = { NOTFOUND }

9
def (9)={ iPat }
use(9)={ iPat }

10
def (10)={ iSub }
use(10)={ iSub }

11
use(11) = { rtnIndex }

Data flow criteria

Basic information: does a def of a variable reach a particular use?

A def of a variable v at location l_i *does not* reach a use at location l_j because

- no path goes from l_i to l_j .
- the value of v is changed by another def before it reaches location location l_j .

A path from l_i to l_j is ***def-clear* with respect to *v*** if for every node n_k and every edge e_k on the path, $k \neq i$ and $k \neq j$, v is not in $def(n_k)$ or in $def(n_k)$.

If a def-clear path goes from l_i to l_j with respect to v, it is said that the def of v at $\boldsymbol{l_i}$ ***reaches* the use at $\boldsymbol{l_j}$**.

Data flow criteria

A ***du-path* with respect to a variable *v*** is a simple path that is def-clear with respect to *v* from a node n_i for which *v* is in *def*(n_i) to a node n_j for which *v* is in *use*(n_j).

The test criteria for data flow are defined as sets of du-paths (def-path sets).

***du*(*ni, v*)** be the set of du-paths with respect to variable *v* that start at node n_i.

Paths starting with the definition of variable v.

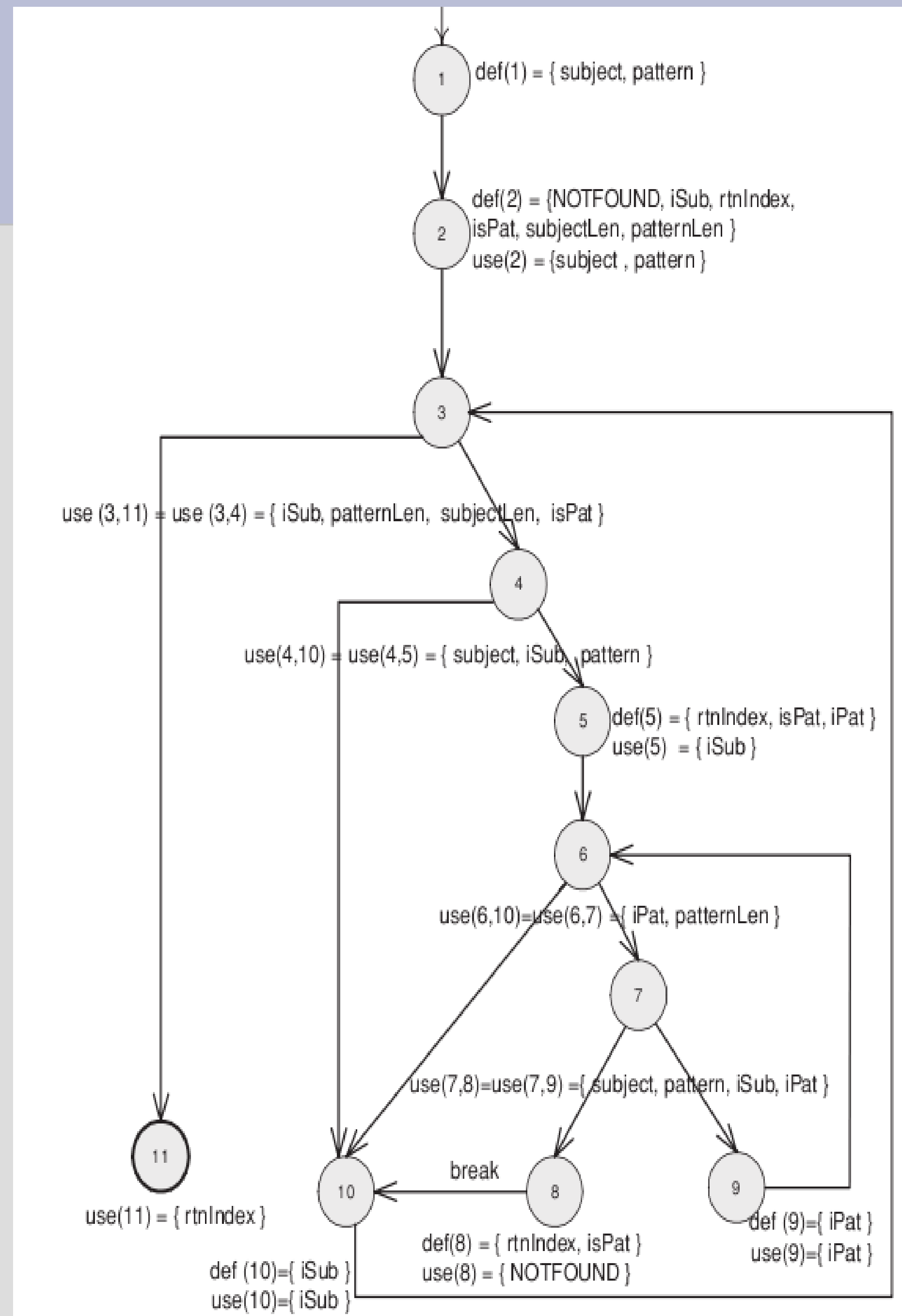
***du*(*ni , nj, v*)** be the set of dupaths with respect to variable *v* that start at node n_i and end at node n_j.

Paths from definition to use of variable v.

Sets of du paths

***du*(10, *i Sub*) =** {[10, 3, 4], [10, 3, 4, 5], [10, 3, 4, 5, 6, 7, 8], [10, 3, 4, 5, 6, 7, 9], [10, 3, 4, 5, 6, 10], [10, 3, 4, 5, 6, 7, 8, 10], [10, 3, 4, 10], [10, 3, 11]}

***du*(10, 4, *iSub*) =** {[10, 3, 4]}
***du*(10, 5, *iSub*) =** {[10, 3, 4, 5]}
***du*(10, 8, *iSub*) =** {[10, 3, 4, 5, 6, 7, 8]}
***du*(10, 9, *iSub*) =** {[10, 3, 4, 5, 6, 7, 9]}
***du*(10, 10, *iSub*) =** {[10, 3, 4, 5, 6, 10], [10, 3, 4, 5, 6, 7, 8, 10], [10, 3, 4, 10]}
***du*(10, 11, *iSub*) =** {[10, 3, 11]}



Coverage criteria for Data Flow

A test path p is said to ***du tour* subpath *d*** with respect to v if p tours d and the portion of p to which d corresponds is def-clear with respect to v.

All-Defs Coverage (ADC): *For each def-path set $S = du(n, v)$, TR contains at least one path d in S.*

Since def-path set $du(n, v)$ represents all def-clear simple paths from n to all uses of v *the criterion* requires to tour **at least one path to at least one use**.

Coverage criteria for Data Flow

All-Uses Coverage (AUC): *For each def-pair set*

$S = du(n_i , n_j ,v)$, TR contains at least one path d in S.

Since def-path set *du*(*ni* , *nj*, *v*) represents all the def-clear simple paths from a def of *v* at n_i to a use of *v* at n_j the criterion All-Uses requires to **tour at least one path for every def-use pair**.

All-du-Paths Coverage (ADUPC): *For each def-pair set S = du(ni , nj, v), TR contains every path d in S.*

Every du-path should be included-

Example of data flow coverage criteria