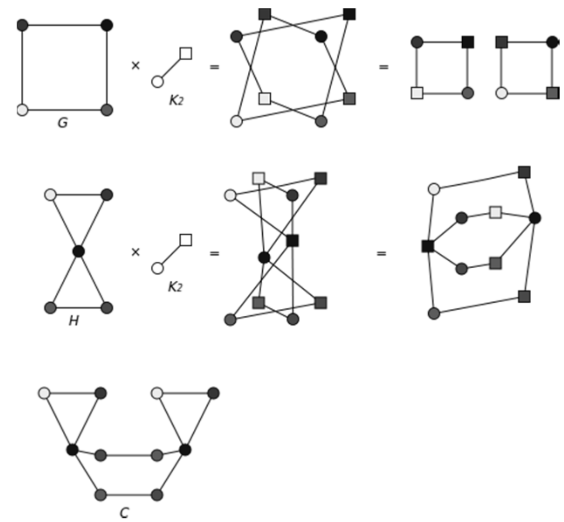
Coverage

- Literature of software testing is primarily concerned with various notions of *coverage*
- Four basic kinds of coverage:
  - **Graph coverage**
  - **Logic coverage**
  - **Input space partitioning**
  - **Syntax-based coverage**



$$\neg(p \wedge \neg p)$$

- Two purposes: to know what we have & haven’t tested, and to know when we can “safely” stop testing

Need to Abstract Testing

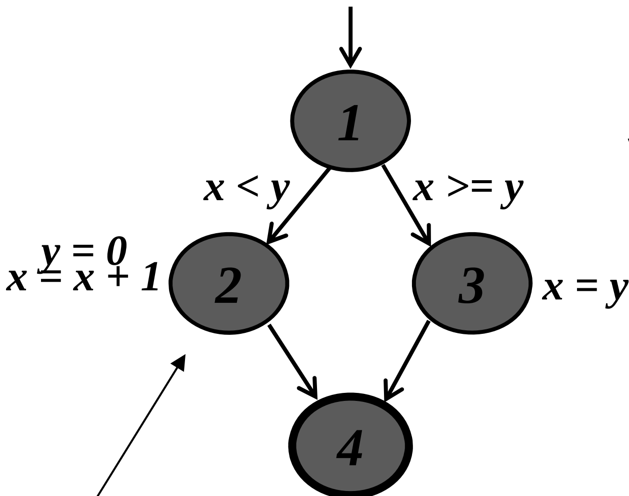
- As we have seen, we can't try all possible executions of a program
- How can we *measure* "how much testing" we have done and look for more things to test?
  - Could talk about modules we have and have not tested, or use cases explored
  - Could also talk *structurally* – what aspects of the *source code* have we tested?

Graph Coverage

- *Cover all the nodes, edges, or paths of some graph related to the program*
- *Examples:*
  - *Statement coverage*
  - *Branch coverage*
  - *Path coverage*
  - *Data flow (def-use) coverage*
  - *Model-based testing coverage*
  - *Many more – most common kind of coverage, by far*

Statement/Basic Block Coverage

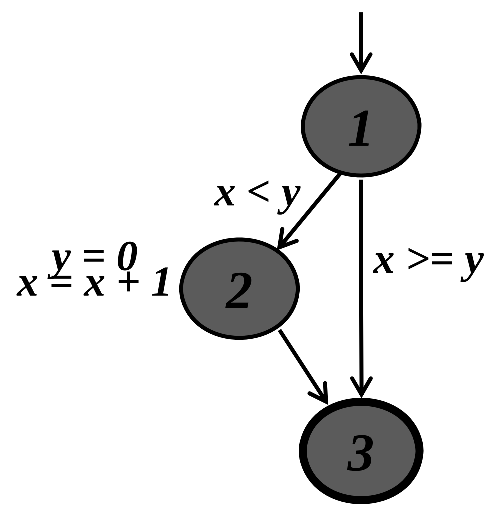
```
if (x < y)
{
  y = 0;
  x = x + 1;
}
else
{
  x = y;
}
```



Statement coverage:
Cover every node of these graphs

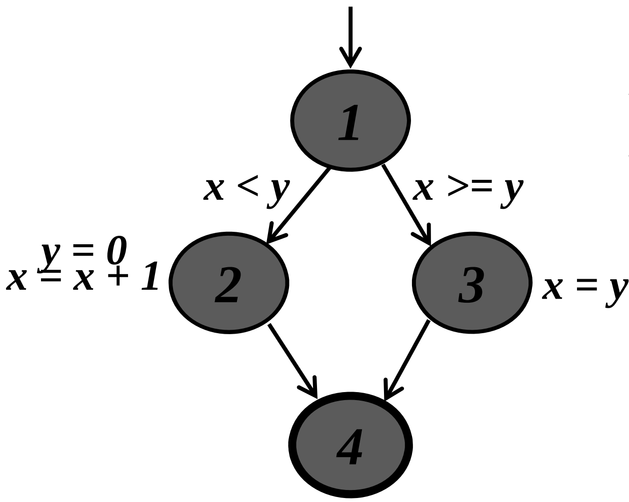
Treat as one node because if one statement executes the other must also execute (code is a basic block)

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



Branch Coverage

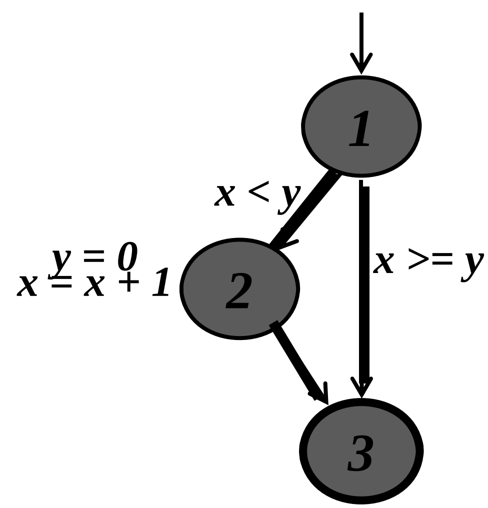
```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
```



Branch coverage vs. statement coverage: Same for if-then-else

But consider this if-then structure. For branch coverage can't just cover all nodes, but must cover all edges – get to node 3 both after 2 and without executing 2!

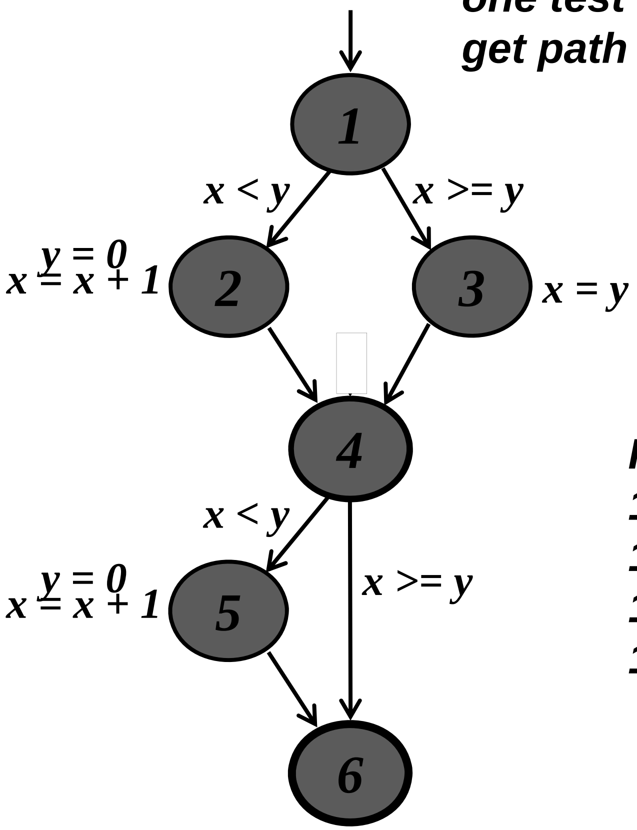
```
if (x < y)
{
   y = 0;
   x = x + 1;
}
```



Path Coverage

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

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



How many paths through this code are there? Need one test case for each to get path coverage

To get statement and branch coverage, we only need two test cases:
1 2 4 5 6 and 1 3 4 6

Path coverage needs two more:
1 2 4 5 6
1 3 4 6
1 2 4 6
1 3 4 5 6

In general: exponential in the number of conditional branches!

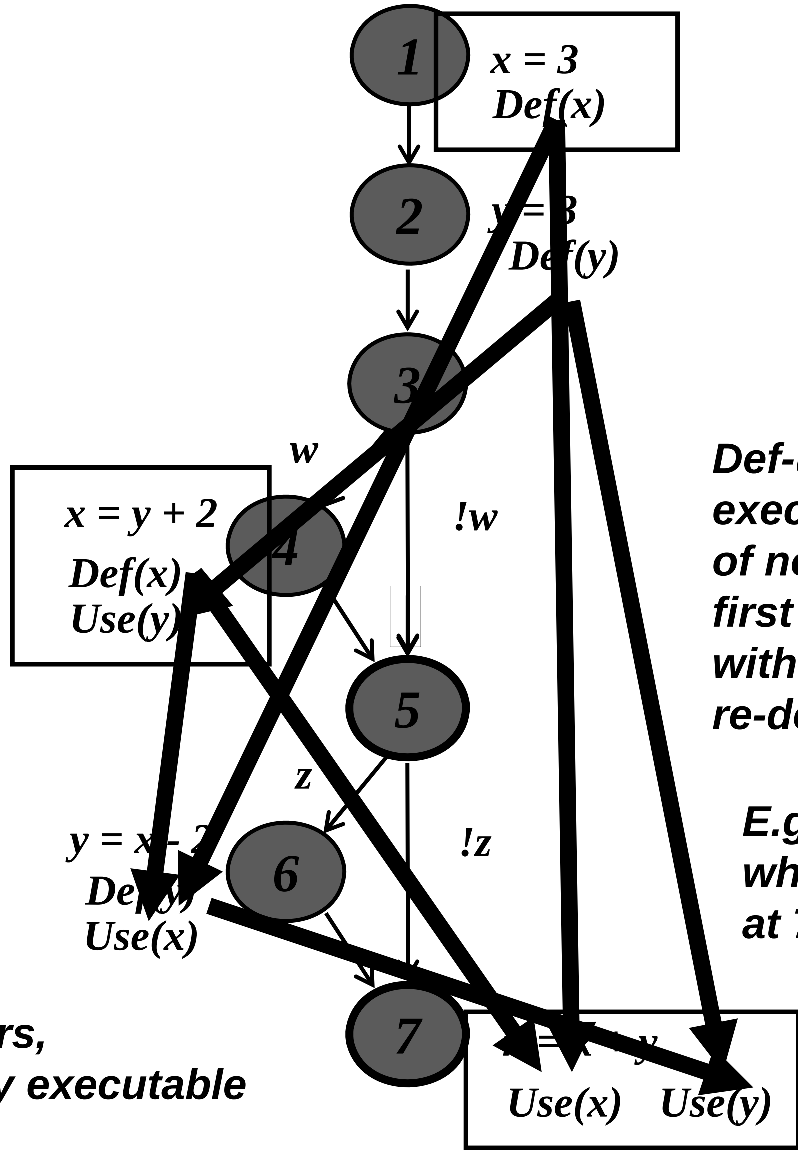
Data Flow (Def-Use) Coverage

```
x = 3;
y = 3;

if (w) {
   x = y + 2;
}

if (z) {
  y = x – 2;
}

n = x + y
```



Annotate program with locations where variables are defined and used (very basic static analysis)

Def-use pair coverage requires executing all possible pairs of nodes where a variable is first defined and then used, without any intervening re-definitions

E.g., this path covers the pair where x is defined at 1 and used at 7: 1 2 3 5 6 7

May be many pairs, some not actually executable

But this path does NOT: 1 2 3 4 5 6 7

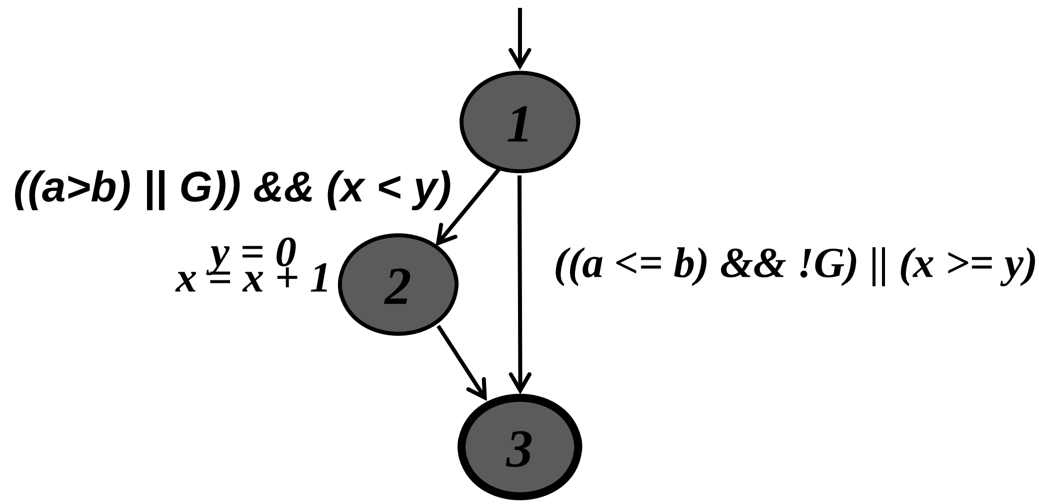
Logic Coverage

What if, instead of:

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

we have:

```
if (((a>b) || G)) && (x < y))
{
  y = 0;
  x = x + 1;
}
```



Now, branch coverage will guarantee that we cover all the edges, but does not guarantee we will do so for all the different logical reasons

We want to test the logic of the guard of the if statement

Active Clause Coverage

	((a > b)	or G)	and (x < y)	
1	T	F	T	T
2	F	F	T	F
3	F	T	T	T
4	F	F	T	F
5	T	T	T	T
6	T	T	F	F

With these values for G and (x<y), (a>b) determines the value of the predicate

With these values for (a>b) and (x<y), G determines the

With these values for (a>b) and G, (x<y) determines the value of the predicate

duplicate

9

Input Domain Partitioning

- **<u>Partition scheme</u> *q* of domain *D***
- **The partition *q* defines a <u>set of blocks</u>, $Bq = b_1$, b_2 , … b_Q**
- **The partition must satisfy two properties:**
  1. **blocks must be <u>pairwise disjoint</u> (no overlap)**
  2. **together the blocks <u>cover</u> the domain *D* (complete)**

Coverage then means using at least one input from each of b_1, b_2, b_3, . . .

Syntax-Based Coverage

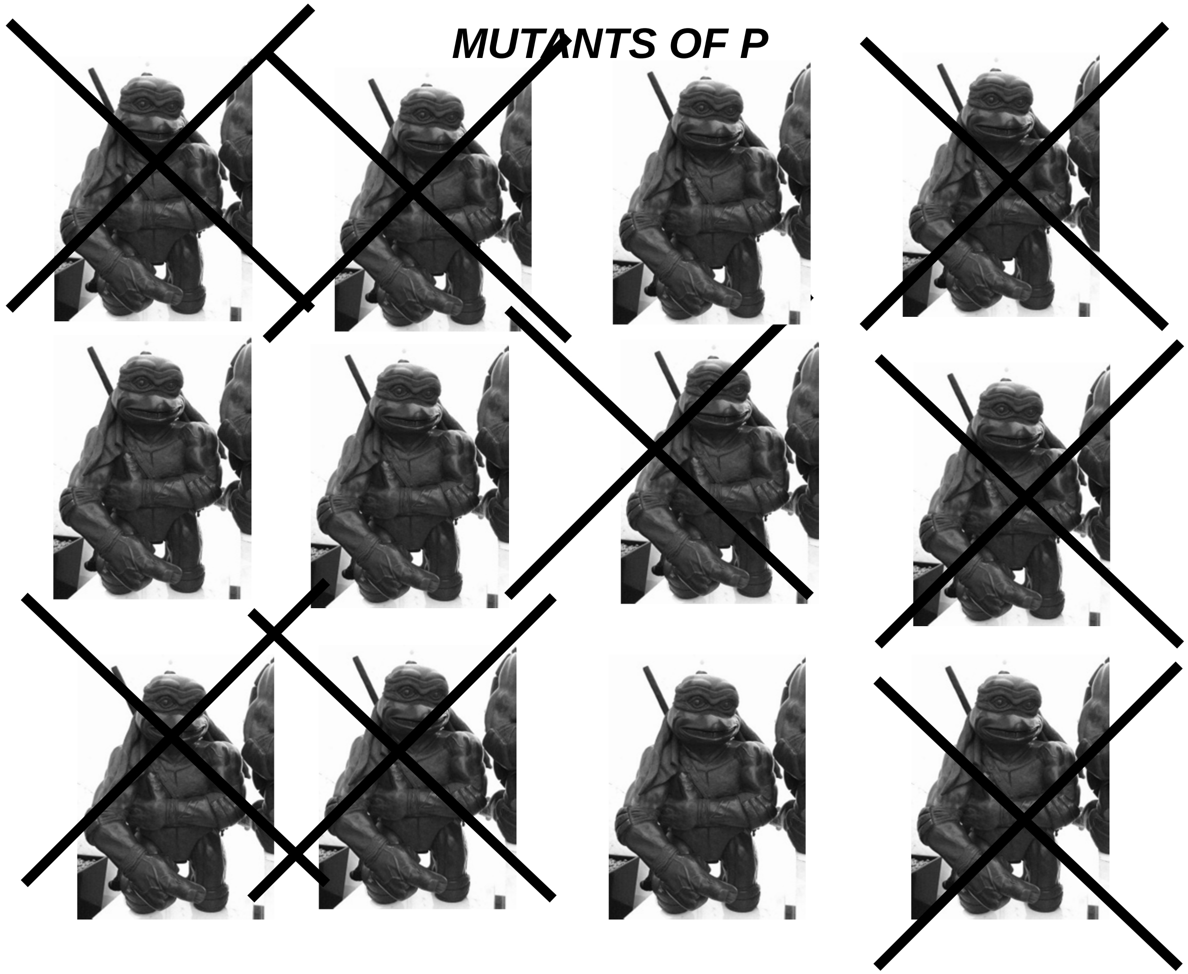
- Usually known as *mutant testing*
- Bit different kind of creature than the other coverages we've looked at
- Idea: generate many syntactic *mutants* of the original program
- Coverage: how many mutants does a test suite kill (detect)?

Syntax-Based Coverage

MUTANTS OF P

Program P

P



***100% coverage
means you kill
all the mutants with
your test suite***

12