Software testing taxonomy(Testing methods)

➢ Testing methods (or strategies) define the way for designing test cases. They can be responsibility based (black-box), implementation based (white box) also known as structural testing.

- Black-box techniques design test cases on the basis of the specified functionality of the item to be tested. White-box ones rely on source code analysis to develop test cases.

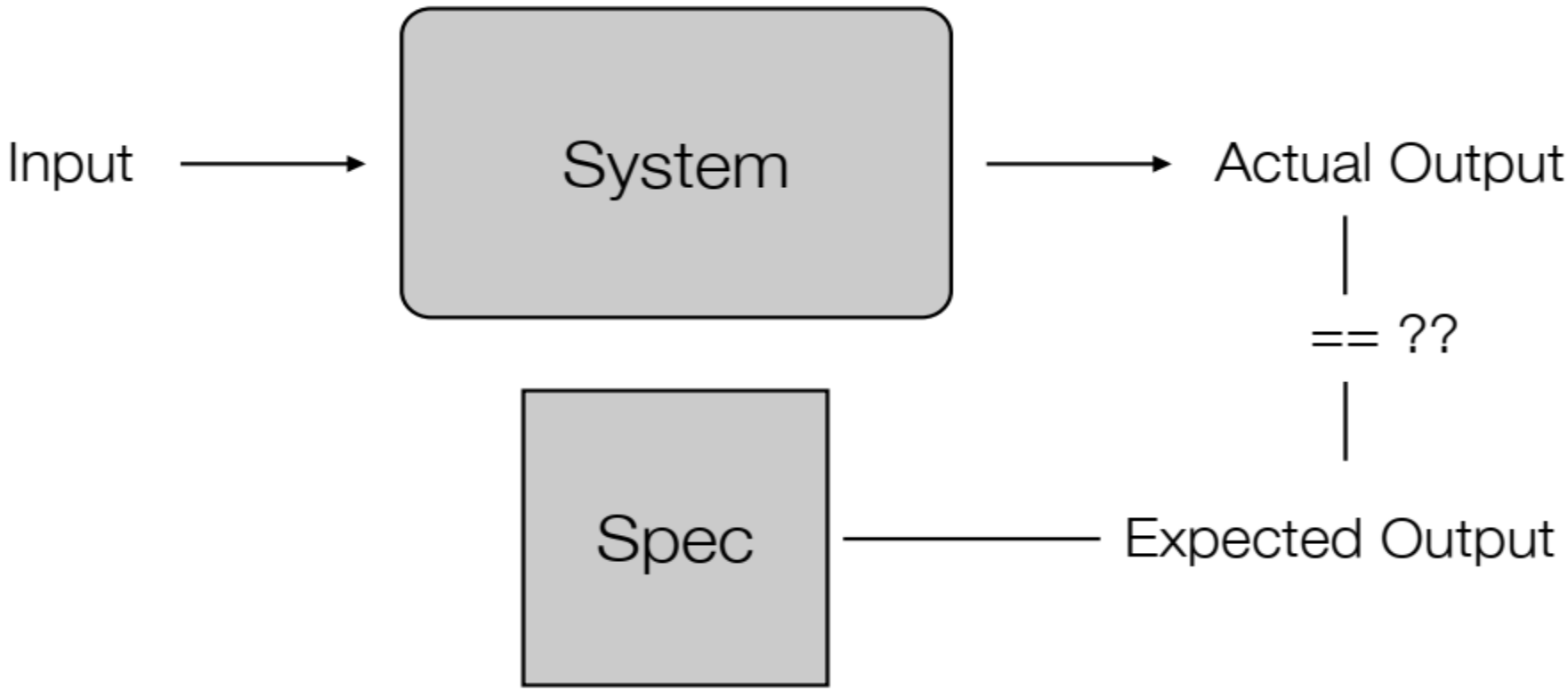
Black-box testing

➢ Black-box testing (also known as functional or behavioral testing) is based on requirements with no knowledge of the internal program structure or data.

➢ Black-box testing relies on the specification of the system or the component that is being tested to derive test cases.

➢ The system is a black- box whose behavior can only be determined by
studying its inputs and the related outputs.

Black Box Testing



A **black box test** passes **input** to a system, records the **actual output** and compares it to the **expected output**

Note: if you do not have a spec, then any behavior by the system is correct!

Some of the most well-known black-box testing techniques are:

Systematic testing: This refers to a complete testing approach in which SUT is shown to conform exhaustively to a specification, up to the testing assumptions.

➢ Some of the most commonly performed are equivalence partitioning and boundary value analysis, and also logic-based techniques, such as cause effect graphing and decision tables.

Random testing: This is literally the antithesis of systematic testing - the
sampling is over the entire input domain.

➢ Fuzz testing is a form of black-box random testing, which randomly
mutates well-formed inputs and tests the program on the resulting data.

➢ It delivers randomly sequenced and/or structurally bad data to a system to see if failures occur.

Model-based testing (MBT): This is a testing strategy in which test cases are derived in part from a model that describes some (if not all) aspects of the SUT.

➢ MBT is a form of black-box testing because tests are generated from a
model, which is derived from the requirements documentation.

➢ It can be done at different levels (unit, integration, or system).

Smoke testing: ensuring the critical functionality of the SUT.

➢ A smoke test case is the first to be run by testers before accepting a build for further testing.
➢ Failure of a smoke test case will mean that the software build is refused.

Sanity testing: This is the process of ensuring the basic functionality of the SUT.

➢ Similarly to smoke testing, sanity tests are performed at the beginning of the test process, but its objective is different.

White-box testing

➢ White-box testing (also known as structural testing) is based on knowledge of the internal logic of an application's code.
➢ It determines if the program-code structure and logic is faulty.
➢ White-box test cases are accurate only if the tester knows what the program is supposed to do.

Focus on

- Code coverage
- Proper error handling
- Working as documented (is method “foo” thread safe?)
- Proper handling of resources
- How does the software behave when resources become constrained?

- Code coverage defines the degree of source code, which has been tested, for example, in terms of percentage of LOCs. There are several criteria for the code coverage:

1. Statement coverage: The line of code coverage granularity.
2. Decision (branch) coverage: Control structure (for example, if- else) coverage granularity.
3. Condition coverage: Boolean expression (true-false) coverage granularity.
4. Paths coverage: Every possible route coverage granularity.
5. Function coverage: Program functions coverage granularity.
6. Entry/exit coverage: Call and return of the coverage granularity.

statement coverage
• write tests until all statements have been executed

branch coverage (a.k.a. edge coverage)
• write tests until each edge in a program's control flow graph has been
executed at least once (covers true/false conditions)

condition coverage
• like branch coverage but with more attention paid to the conditionals (if compound conditional, ensure that all combinations have been covered)

path coverage

• write tests until all paths in a program's control flow graph have been executed multiple times as dictated by heuristics, e.g.,

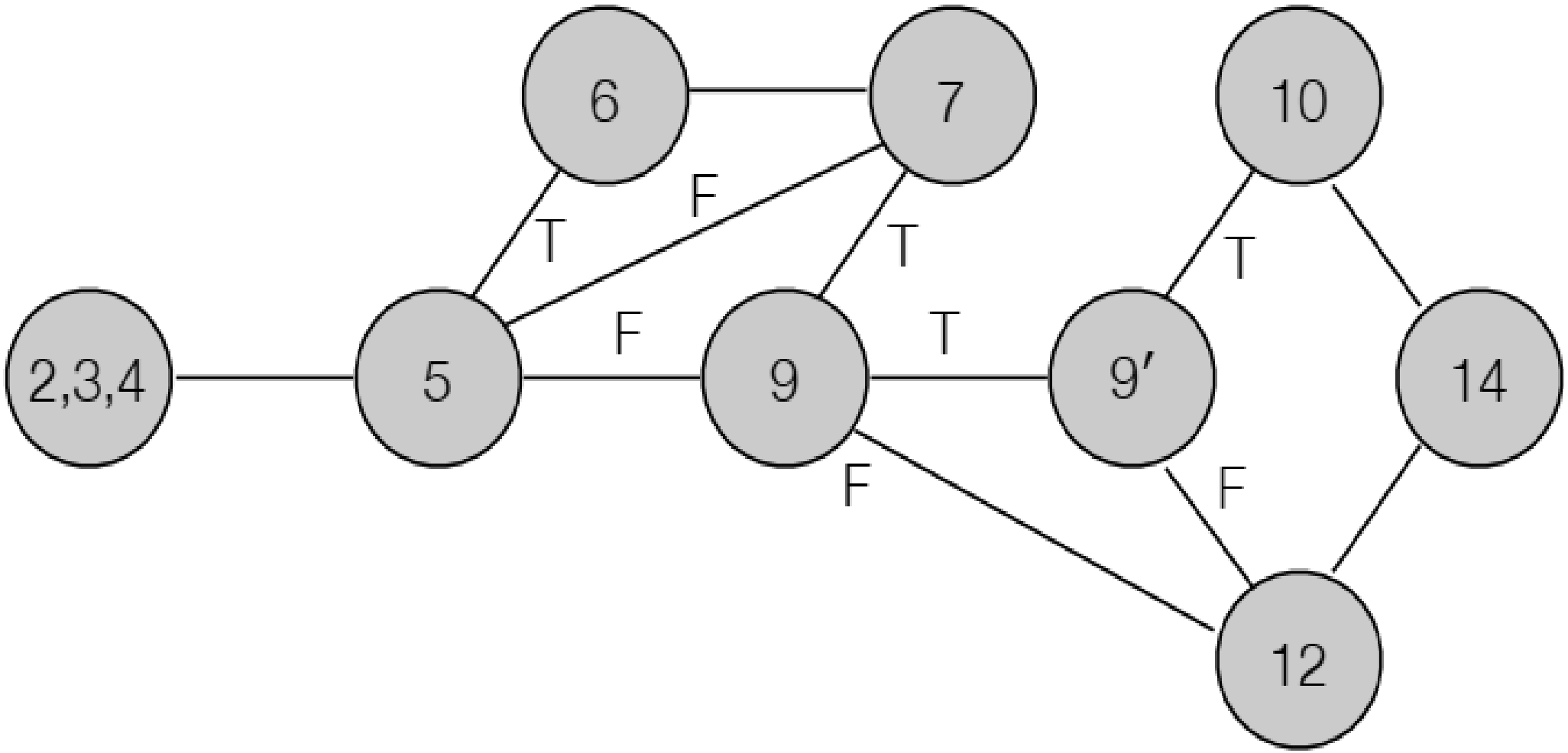
• for each loop, write a test case that executes the loop
• zero times (skips the loop)
• exactly one time
• more than once (exact number depends on context)

Control Flow Graphs(CFG)

- Control Flow Graphs represent the program's control structure graphically.
  - It has three components.

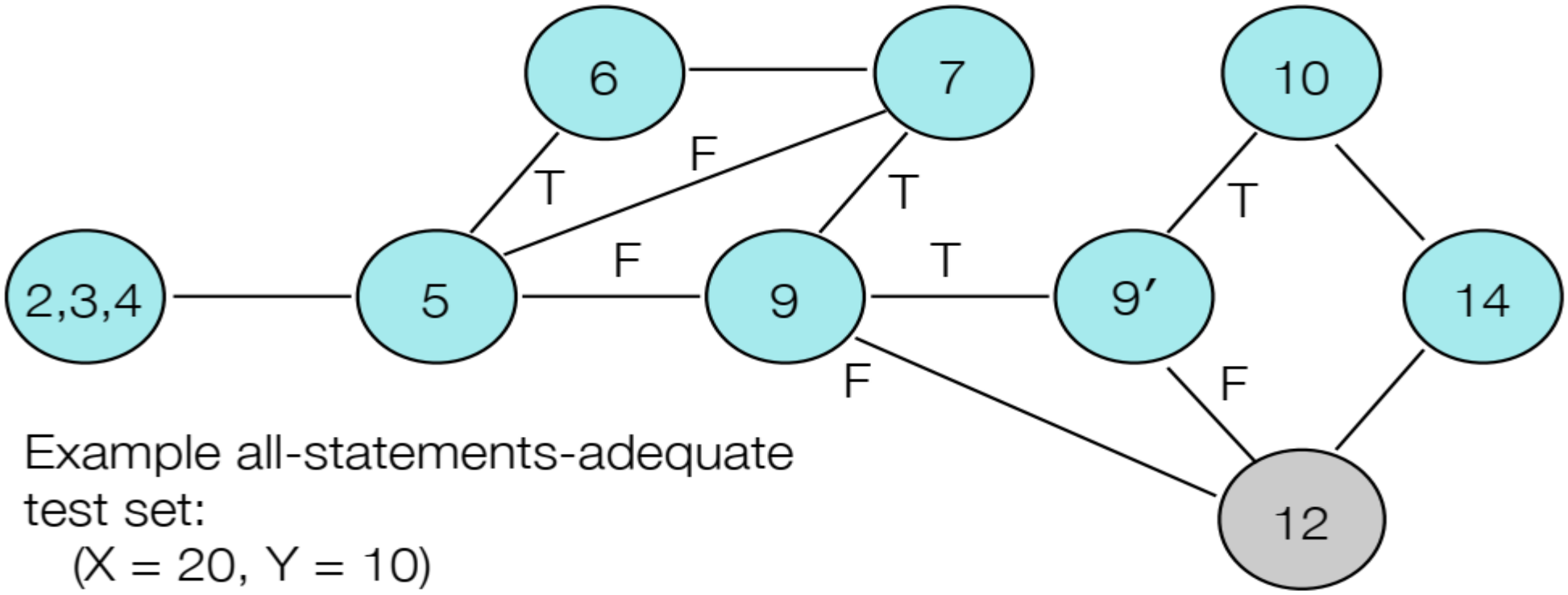
1. A decision: a program point at which the control can diverge.(e.g., if and case statements).
2. A junction: a program point where the control flow can merge. (e.g., end if, end loop, goto statement)
3. A process block : a sequence of program statements uninterrupted by either decisions or junctions. (i.e., straight-line code). A process has one entry and one exit.

```
1       function P return INTEGER is
2       begin
3          X, Y: INTEGER;
4          READ(X); READ(Y);
5          while (X > 10) loop
6             X := X – 10;
7             exit when X = 10;
8          end loop;
9          if (Y < 20 and then X mod 2 = 0) then
10            Y := Y + 20;
11         else
12            Y := Y – 20;
13         end if;
14         return 2 * X + Y;
15      end P;
```


6
7
10
T
F
T
T
T
2,3,4
5
F
9
T
9′
14
F
F
12

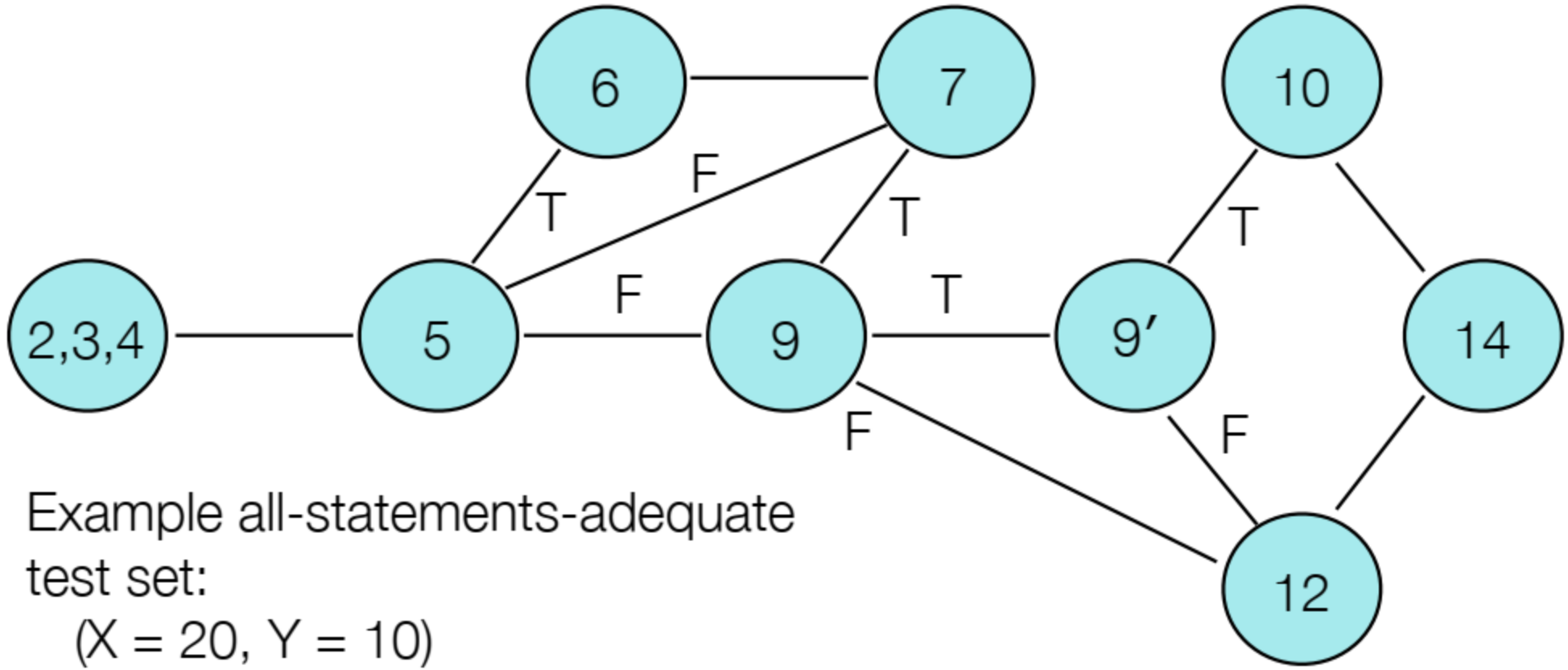
Statement Coverage

• Create a test set T such that by executing P for each t in T each elementary statement of P is executed at least once.



Example all-statements-adequate test set:

(X = 20, Y = 10)



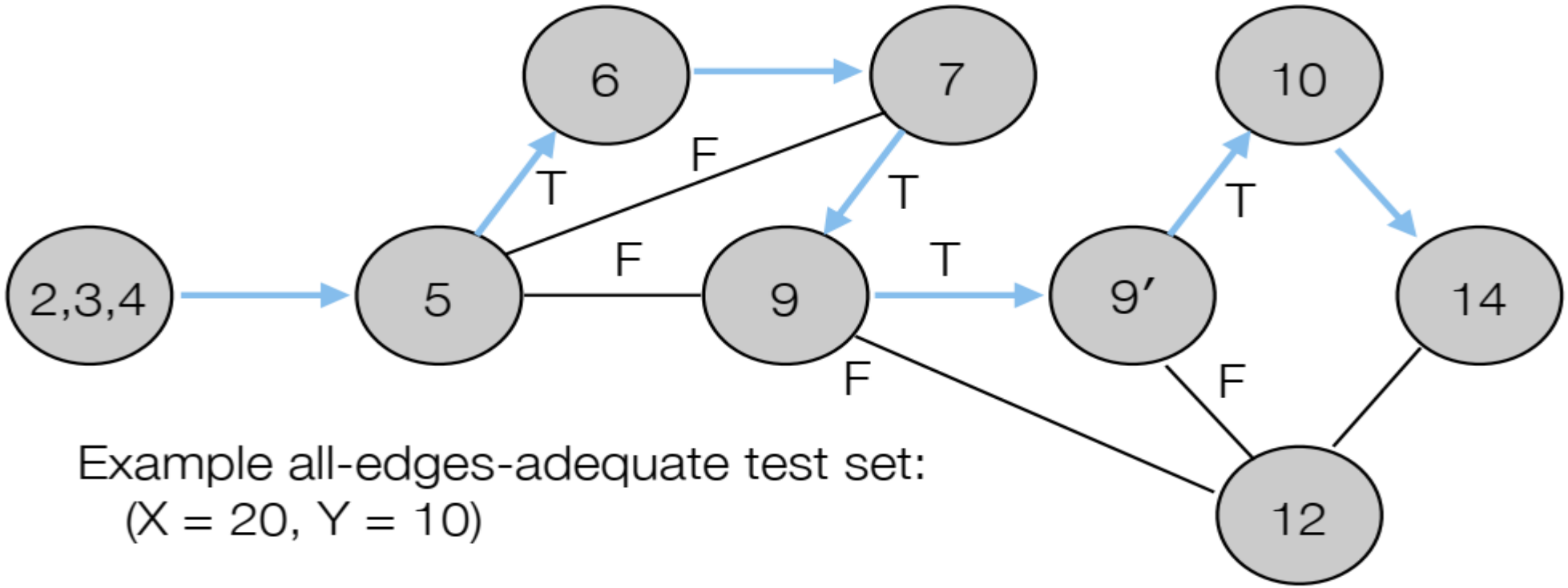
Example all-statements-adequate test set:

(X = 20, Y = 10)

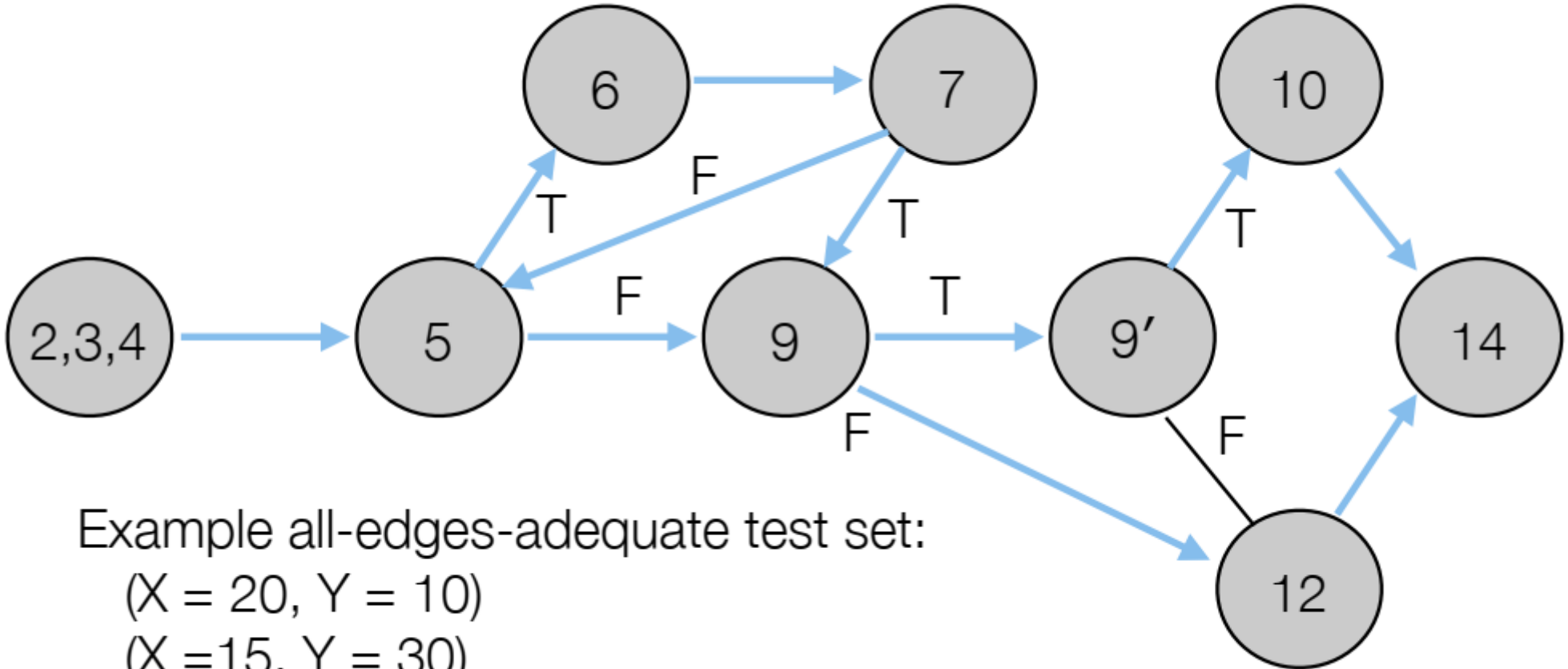
(X = 20, Y = 30)

Edge Coverage

• Select a test set T such that by executing P for each t in T each edge of P's control flow graph is traversed at least once.



Example all-edges-adequate test set:
(X = 20, Y = 10)



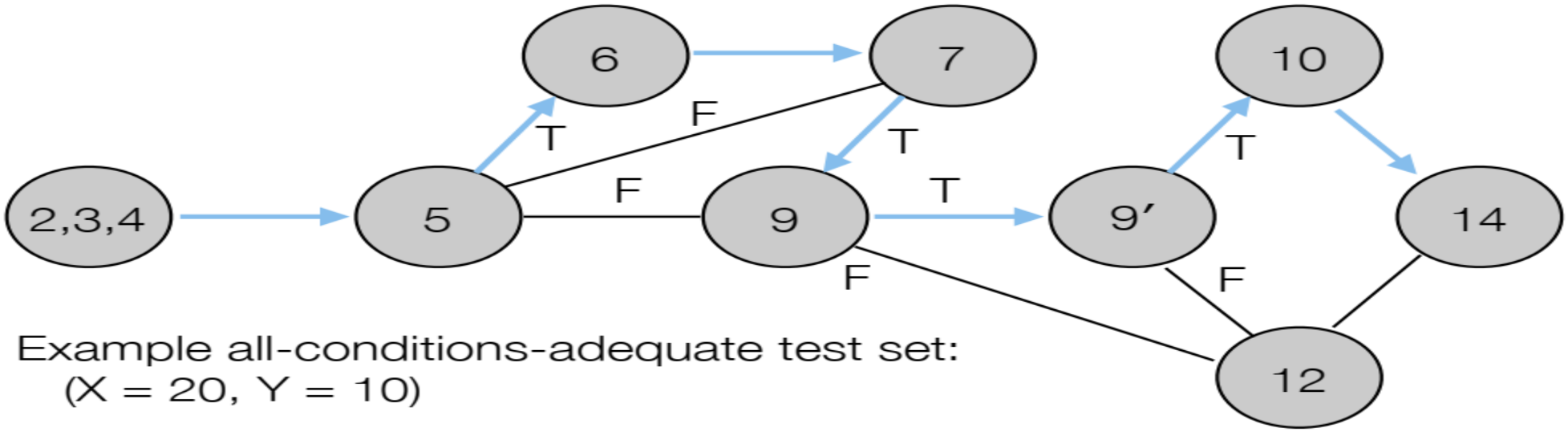
Example all-edges-adequate test set:

(X = 20, Y = 10)

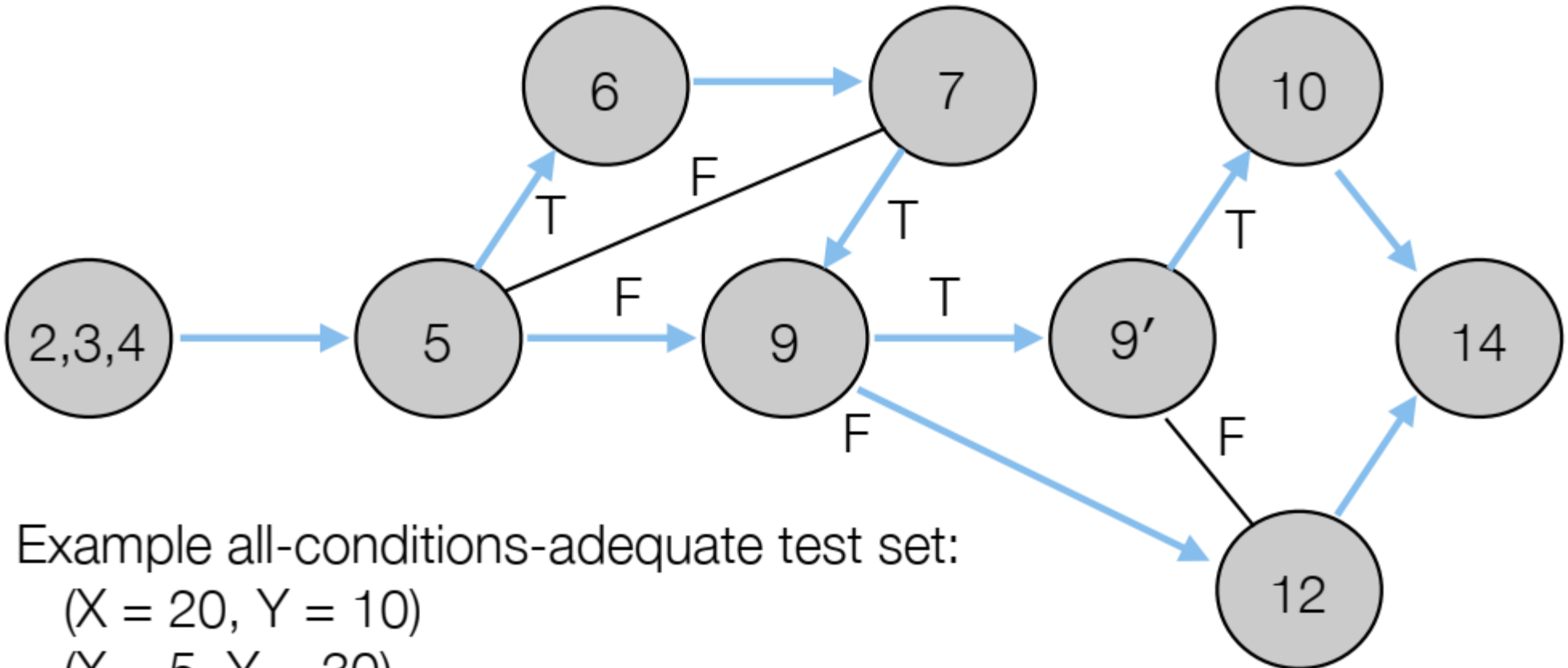
(X =15, Y = 30)

Condition Coverage

• Select a test set T such that by executing P for each t in T each edge of P's control flow graph is traversed at least once and all possible values of the constituents of compound conditions are exercised at least once .



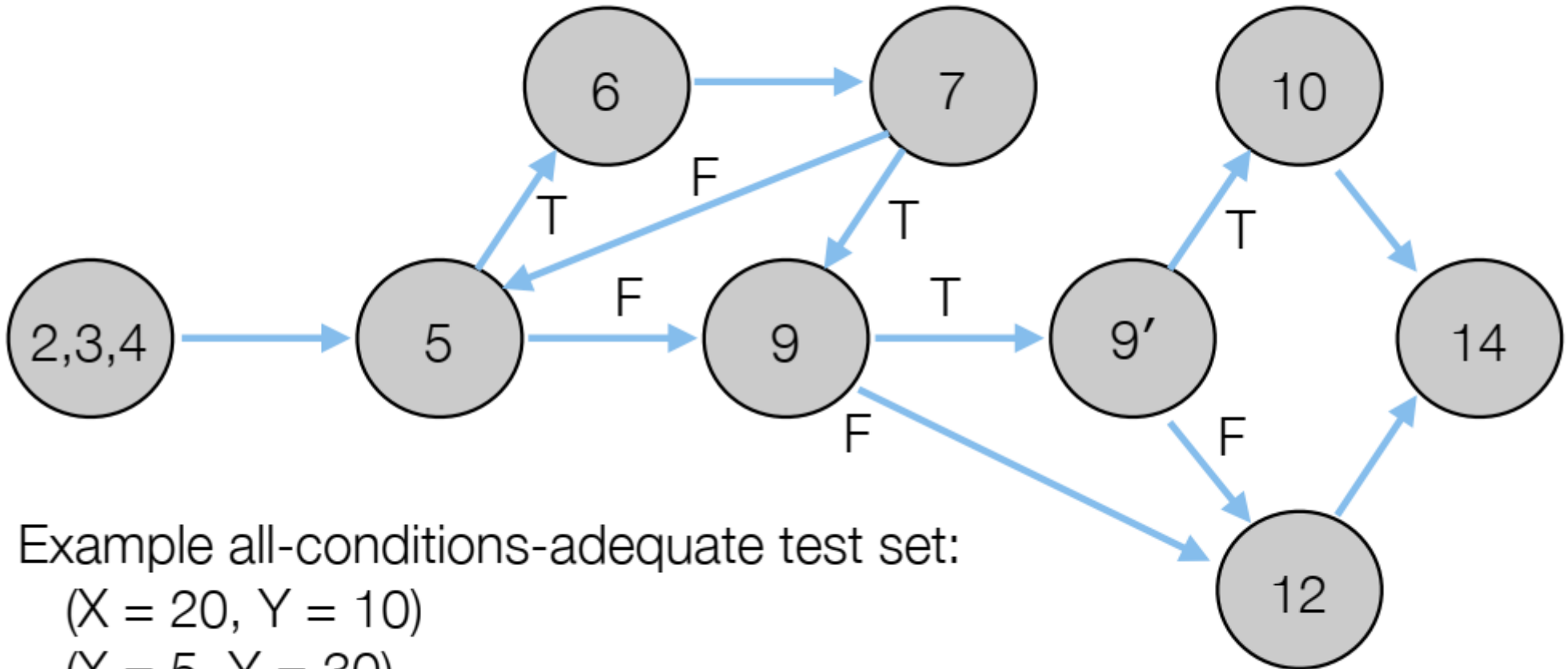
Example all-conditions-adequate test set:
(X = 20, Y = 10)



Example all-conditions-adequate test set:

(X = 20, Y = 10)

(X = 5, Y = 30)



Example all-conditions-adequate test set:

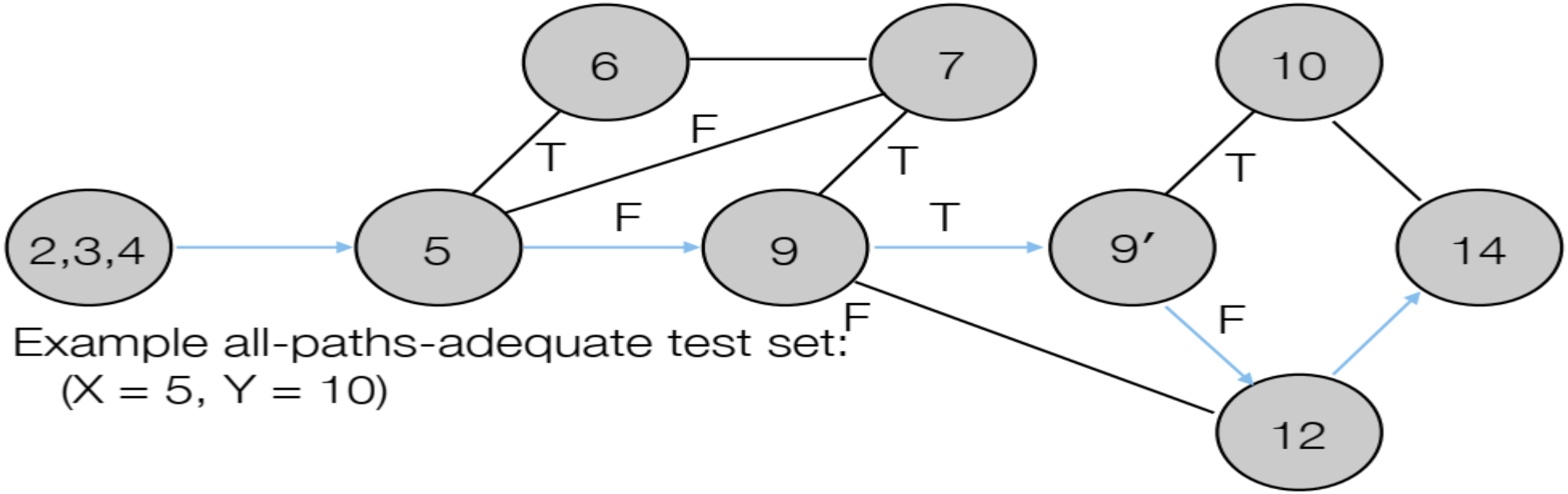
(X = 20, Y = 10)

(X = 5, Y = 30)

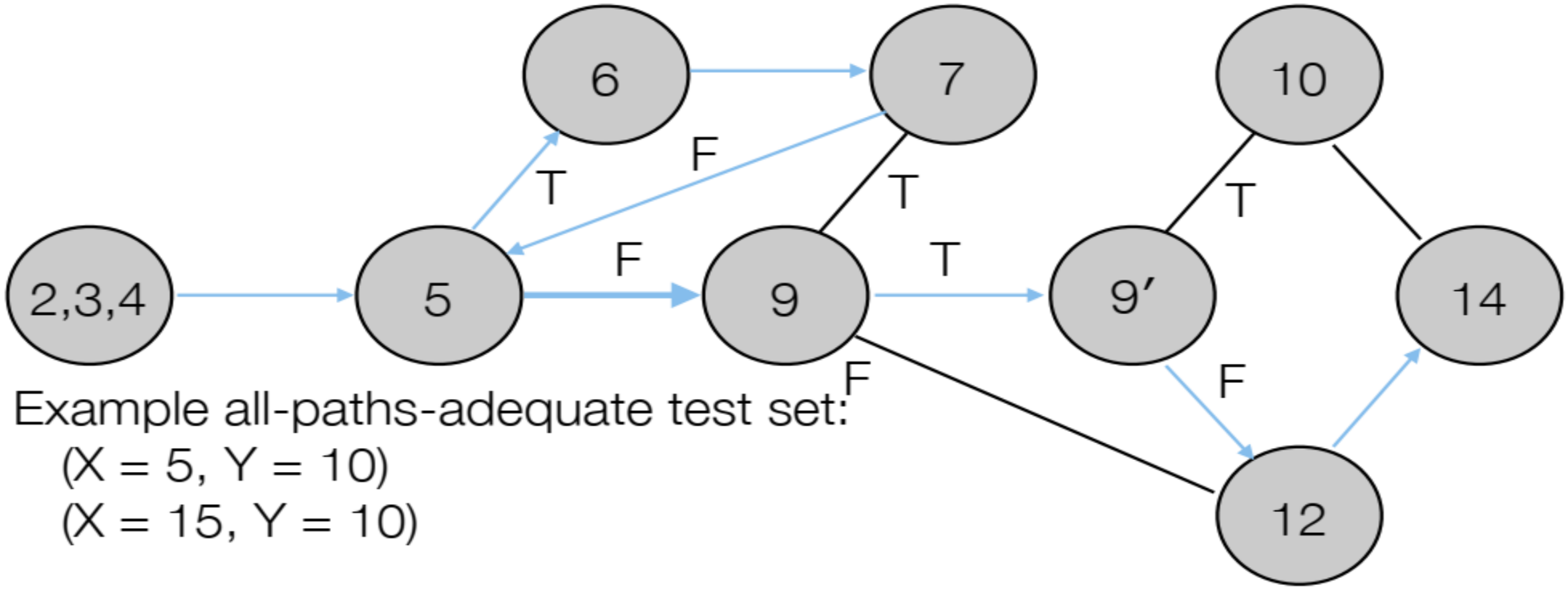
(X = 21, Y = 10)

Path Coverage

- Select a test set T such that by executing P for each t in T
- all paths leading from the initial to the final node of P's control flow graph are traversed at least once .



Path Coverage

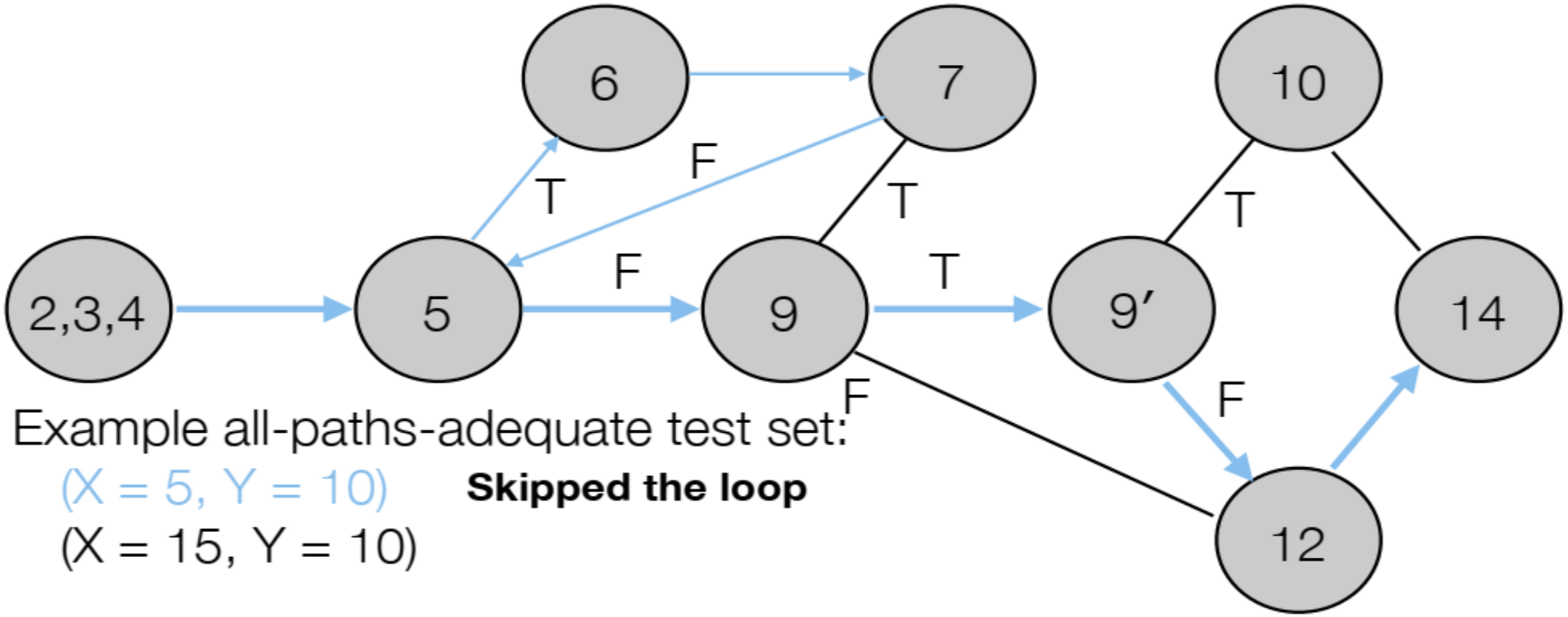


Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

Path Coverage

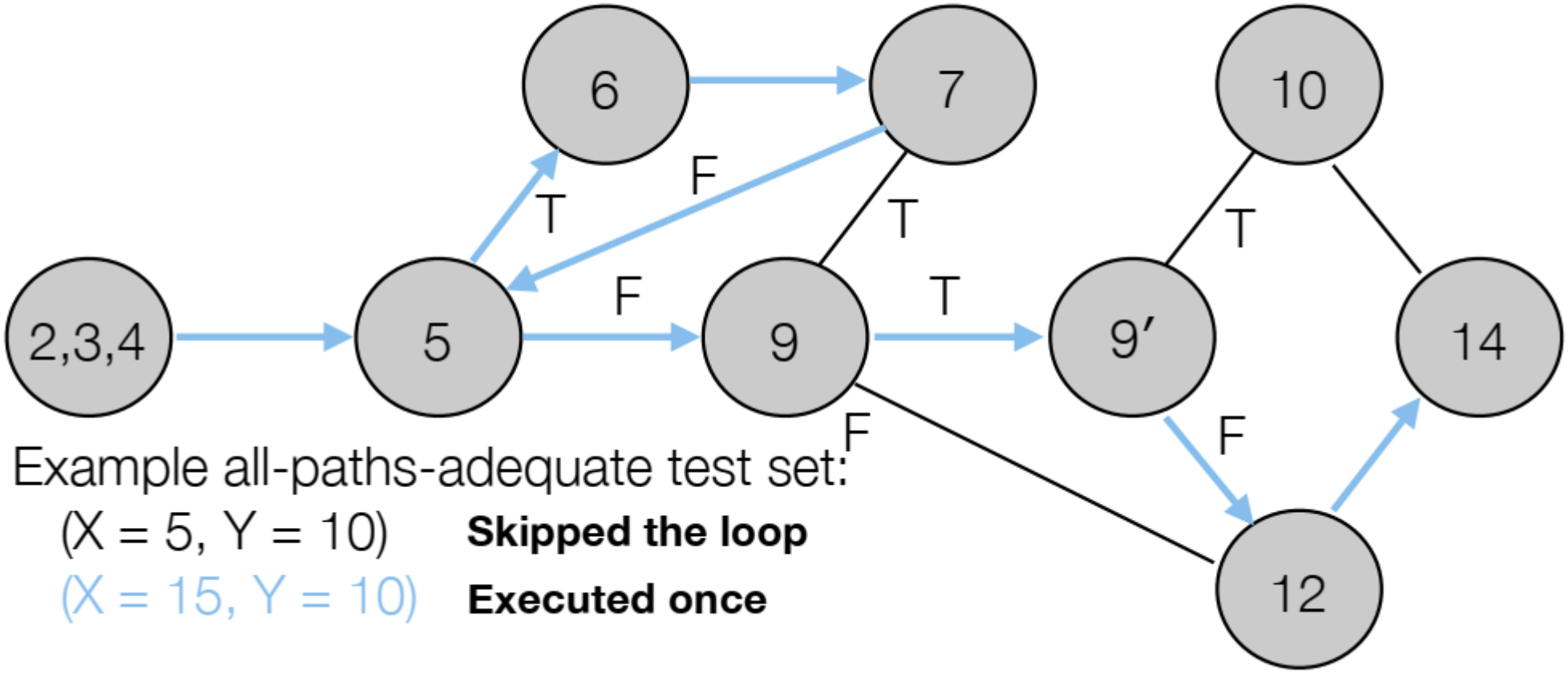


Example all-paths-adequate test set:

(X = 5, Y = 10) **Skipped the loop**

(X = 15, Y = 10)

Path Coverage

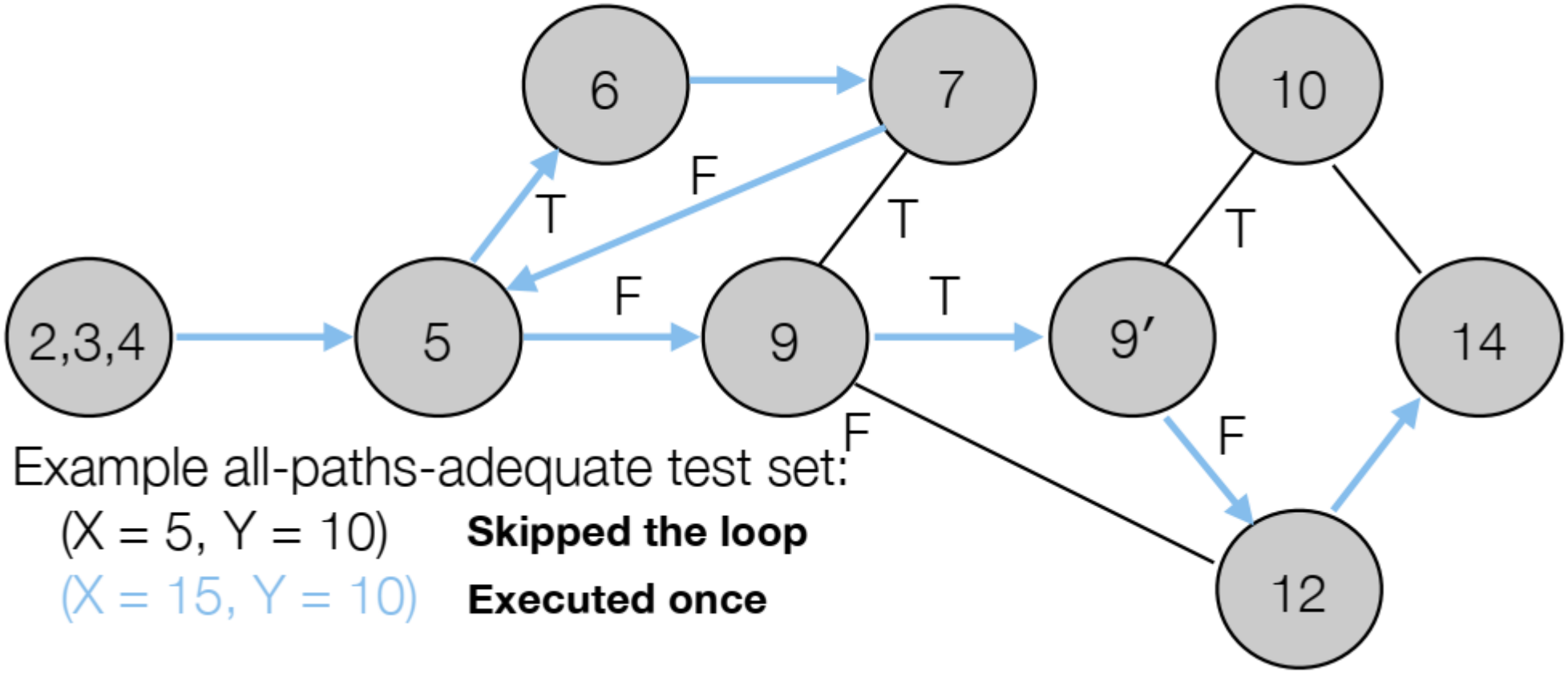


Example all-paths-adequate test set:

(X = 5, Y = 10) **Skipped the loop**

(X = 15, Y = 10) **Executed once**

Path Coverage

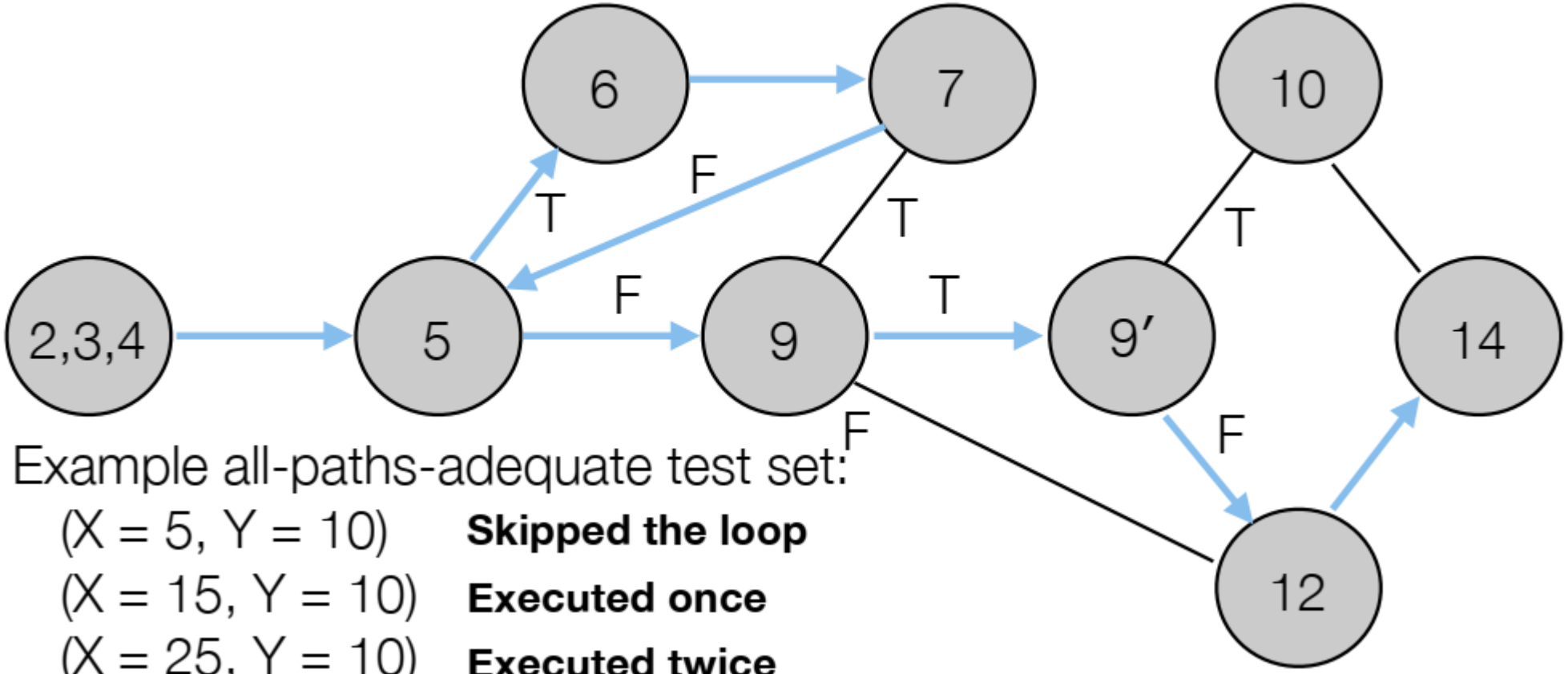


Example all-paths-adequate test set:

(X = 5, Y = 10) **Skipped the loop**

(X = 15, Y = 10) **Executed once**

Path Coverage



Example all-paths-adequate test set:

(X = 5, Y = 10)	**Skipped the loop**
(X = 15, Y = 10)	**Executed once**
(X = 25, Y = 10)	**Executed twice**

- Software testers and developers are responsible for path testing.
- Relies on Control Flow Graph(CFG)
- Cyclomatic Complexity: the minimum number of tests required for a unit of a program.
- Generate tests for each path of execution.
- Path testing is different from branch testing in that path testing is more inclusive. That is the test is thorough. On the other hand, branch testing tests outcomes of a decision.

Stages in path testing

- Design the CFG
- Calculate the cyclomatic complexity
- Generate set of paths
- Test Each path

Example: determine the type of triangle.

1.Input: Three sides of a triangle (side1,side2,side3)>0

2.if (side1>=(side2+side3) or side2>=(side1+side3) or side3>=(side2+side3))

3.output: not a triangle;

4.elseif(side1==side2 and side2==side3)

5.output:Equilateral

6.elseif(side1=side2 or side2==side3 or side1==side3)

7.output: Isoceles

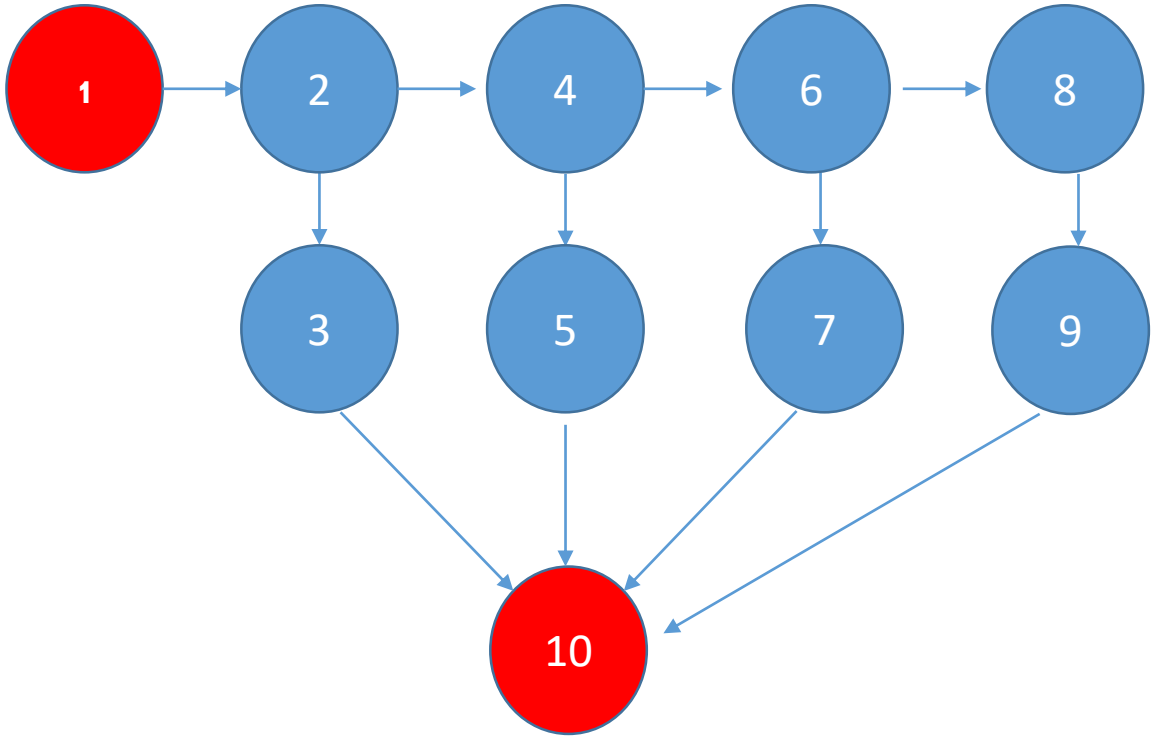
8.else

9.Output: Scalene

10.return output

Now, follow the steps.

1. Design the CFG



Nodes in red represent the entry and exit nodes.

2. Calculate the cyclomatic complexity:

Cyclomatic Complexity= Total No of Edges-Nodes+2

For the given example the total number of edges is 12 and there are 10 nodes. Therefore

CC=12-10+2, which is 4. That is, the minimum number of test required is 4.

3. Generate Test paths. Here generate test data for each output statement given in the program.

Path 1: for values that do not represent a triangle: (side1=12,side2= 3,side3= 4)

For this case, the path followed will be 1,2,3,10.

Path2: for values that represent an equilateral triangle:(side1=5,side2=5,side3=5)

The path followed will be 1,2,4,5,10.

Path3. for values that represent an isosceles triangle: Test with (side1=5,side2=5,side3=4)

The path followed will be 1,2,4,6,7,10.

Path4. for values that represent a scalene triangle : Test with (side1=4,side2=3,side3=2)

The path followed will be 1,2,4,6,8,9,10.