

# Testing & Continuous Integration

Kenneth M. Anderson

University of Colorado, Boulder

CSCI 5828 — Lecture 20 — 03/19/2010

© University of Colorado, 2010

# Goals

2

- ▶ Review material from Chapter 7 of Pilone & Miles
  - ▶ Testing of Systems
    - ▶ unit tests, integration tests, system tests, acceptance tests
  - ▶ Testing of Code
    - ▶ Black Box
    - ▶ Gray Box
    - ▶ White Box
    - ▶ Code Coverage
  - ▶ Continuous Integration

# Testing

- ▶ Testing is a critical element of a larger software engineering concern / process known by many names
  - ▶ software quality control / software quality assurance
  - ▶ validation and verification
    - ▶ **validation**: are we building the right product?
    - ▶ **verification**: does “foo” meet its specification?
      - ▶ where “foo” can be code, a model, a design diagram, a requirement, ...
  - ▶ At each stage, we need to verify that the thing we produce accurately represents its specification

# Terminology

- ▶ An *error* is a mistake made by an engineer
- ▶ A *fault* is a manifestation of that error in the code
- ▶ A *failure* is an incorrect output/behavior that is caused by executing a fault
- ▶ Testing attempts to surface failures in our software systems
  - ▶ Debugging attempts to associate failures with faults so they can be removed from the system
- ▶ If a system passes all of its tests, is it free of all faults?

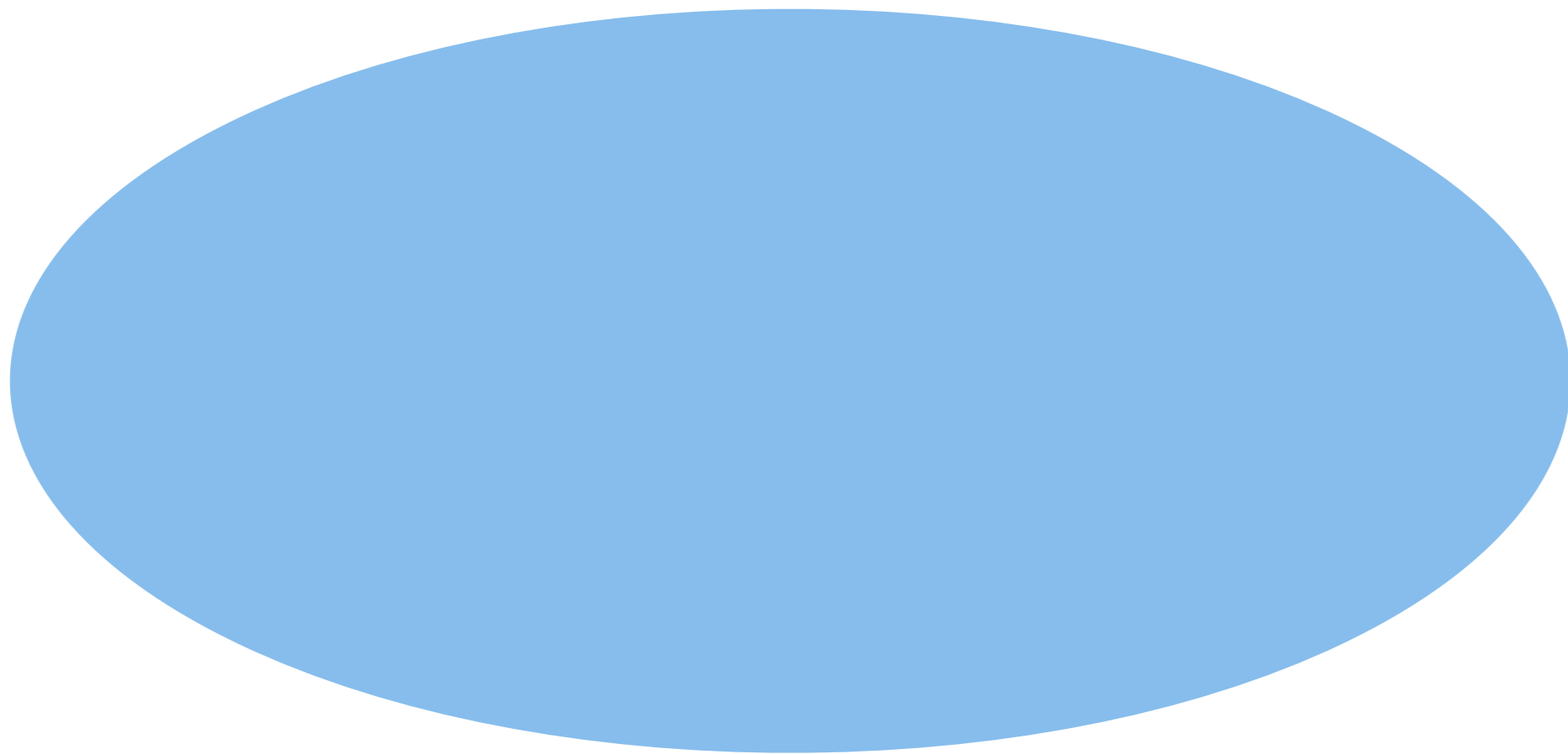
# No!

5

- ▶ Faults may be hiding in portions of the code that only rarely get executed
  - ▶ “Testing can only be used to prove the existence of faults not their absence” or “Not all faults have failures”
    - ▶ Sometimes faults mask each other; this is particularly insidious
- ▶ However, if we do a good job in creating a test set that
  - ▶ covers all functional capabilities of a system
  - ▶ covers all code using a metric such as “branch coverage”
- ▶ Then, having all tests pass increases our confidence that our system has high quality and can be deployed

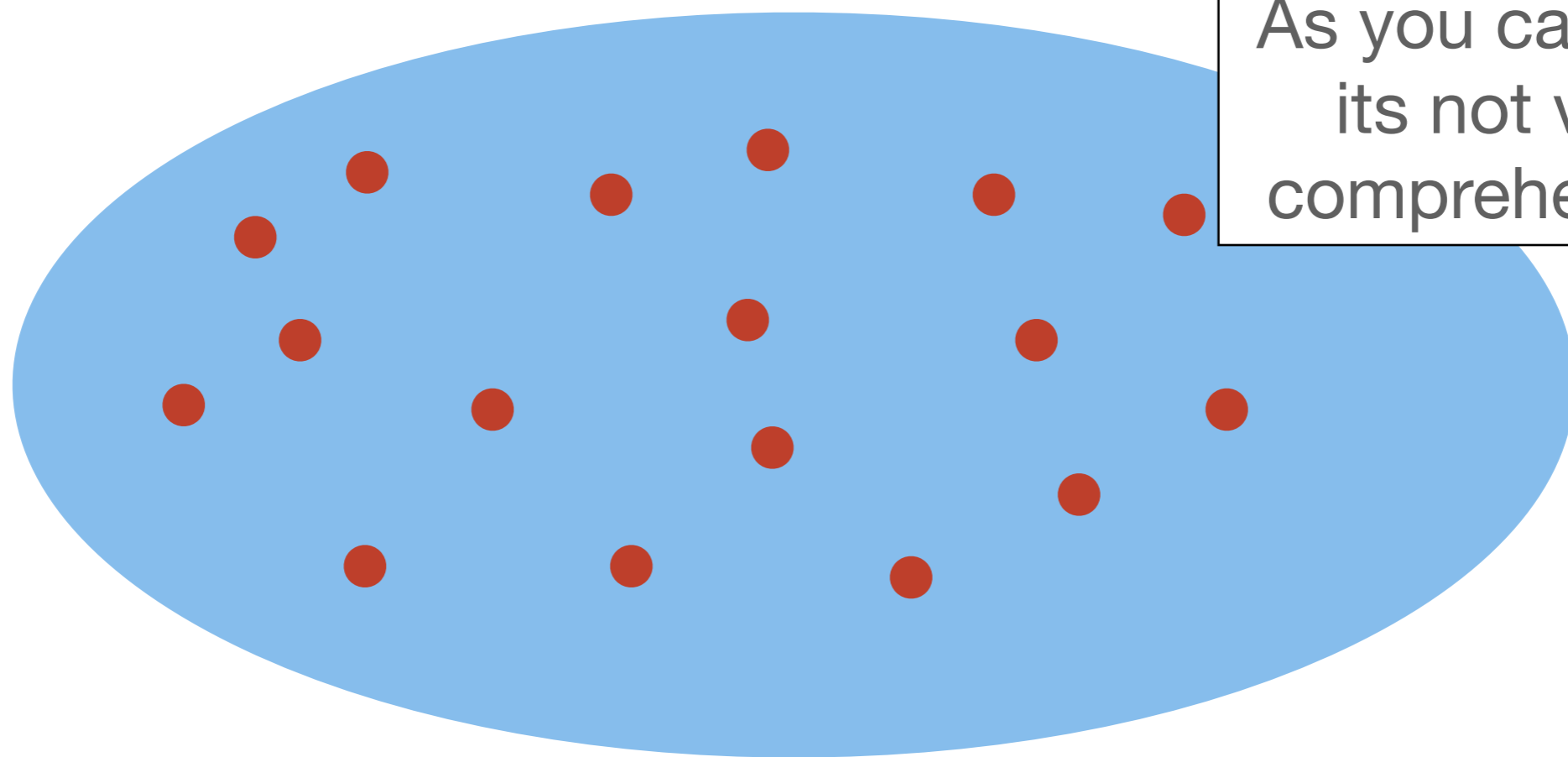
# Looking for Faults

6



All possible states/behaviors of a system

# Looking for Faults

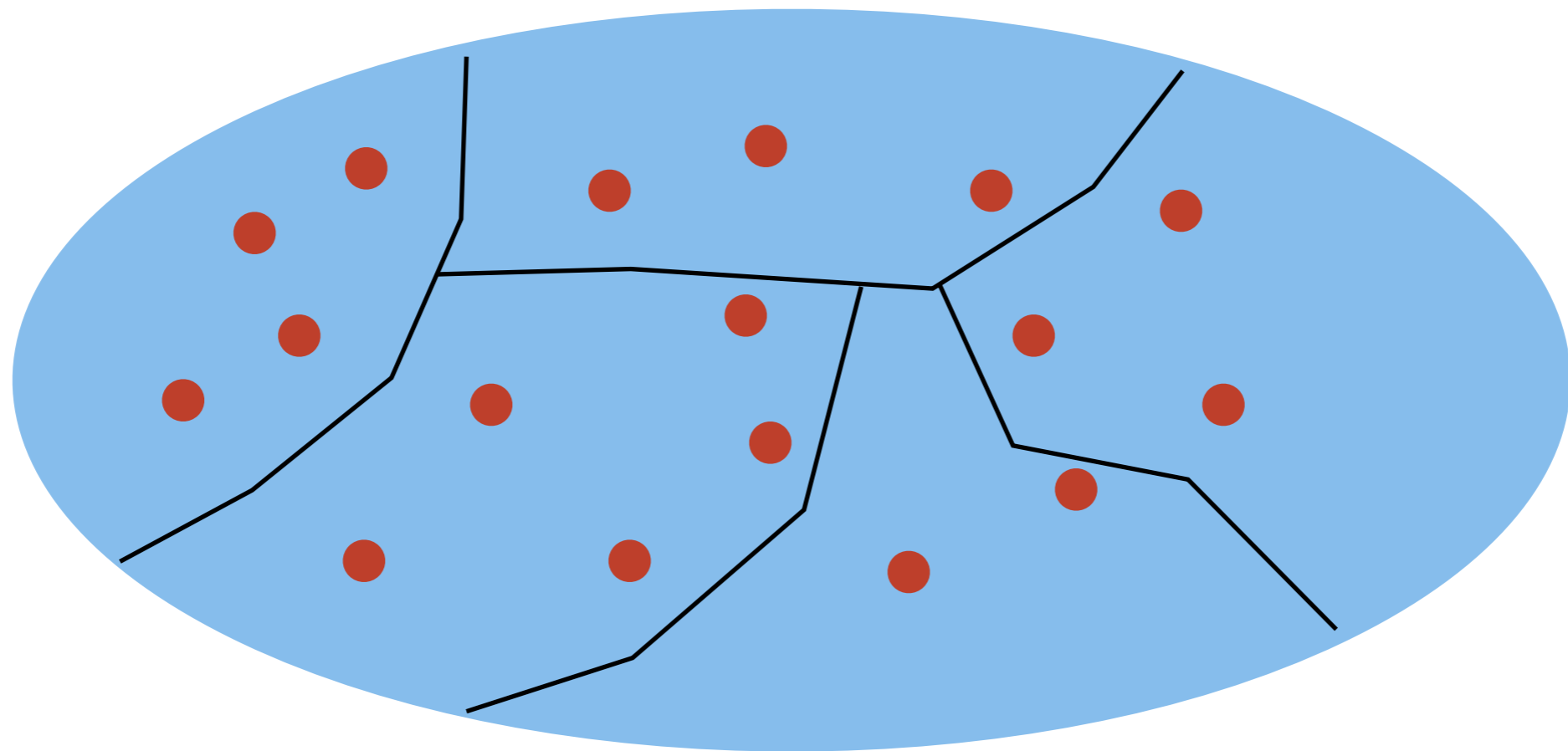


As you can see,  
its not very  
comprehensive

Tests are a way of sampling the behaviors of a software system, looking for failures

# One way forward? Fold

8



The testing literature advocates folding the space into equivalent behaviors and then sampling each partition



# What does that mean?

9

- ▶ Consider a simple example like the greatest common denominator function
  - ▶ `int gcd(int x, int y)`
    - ▶ At first glance, this function has an infinite number of test cases
  - ▶ But lets fold the space
    - ▶ `x=6 y=9`, returns 3, tests common case
    - ▶ `x=2 y=4`, returns 2, tests when x is the GCD
    - ▶ `x=3 y=5`, returns 1, tests two primes
    - ▶ `x=9 y=0`, returns ?, tests zero
    - ▶ `x=-3 y=9`, returns ?, tests negative

# Completeness

10

- ▶ From this discussion, it should be clear that “completely” testing a system is impossible
  - ▶ So, we settle for heuristics
    - ▶ attempt to fold the input space into different functional categories
      - ▶ then create tests that sample the behavior/output for each functional partition
- ▶ As we will see, we also look at our coverage of the underlying code; are we hitting all statements, all branches, all loops?

# Continuous Testing

11

- ▶ Testing is a continuous process that should be performed at every stage of a software development process
  - ▶ Recall our requirements gathering process that continually queried the user, “Did we get this right?”
  - ▶ Recall our emphasis on iteration throughout the entire development process
    - ▶ at the end of each iteration, we check our results to see if what we built is meeting our requirements

# Testing the System (I)

12

## ▶ Unit Tests

- ▶ Tests that cover low-level aspects of a system
  - ▶ For each module, does each operation perform as expected

## ▶ Integration Tests

- ▶ Tests that check that modules work together in combination
- ▶ Most projects on schedule until they hit this point
  - ▶ All sorts of hidden assumptions are surfaced when code written by different developers are used in tandem
- ▶ Lack of integration testing has led to spectacular failures

# Testing the System (II)

13

## ▶ System Tests

- ▶ Tests performed by the developer to ensure that all major functionality has been implemented
  - ▶ Have all user stories been implemented and function correctly?

## ▶ Acceptance Tests

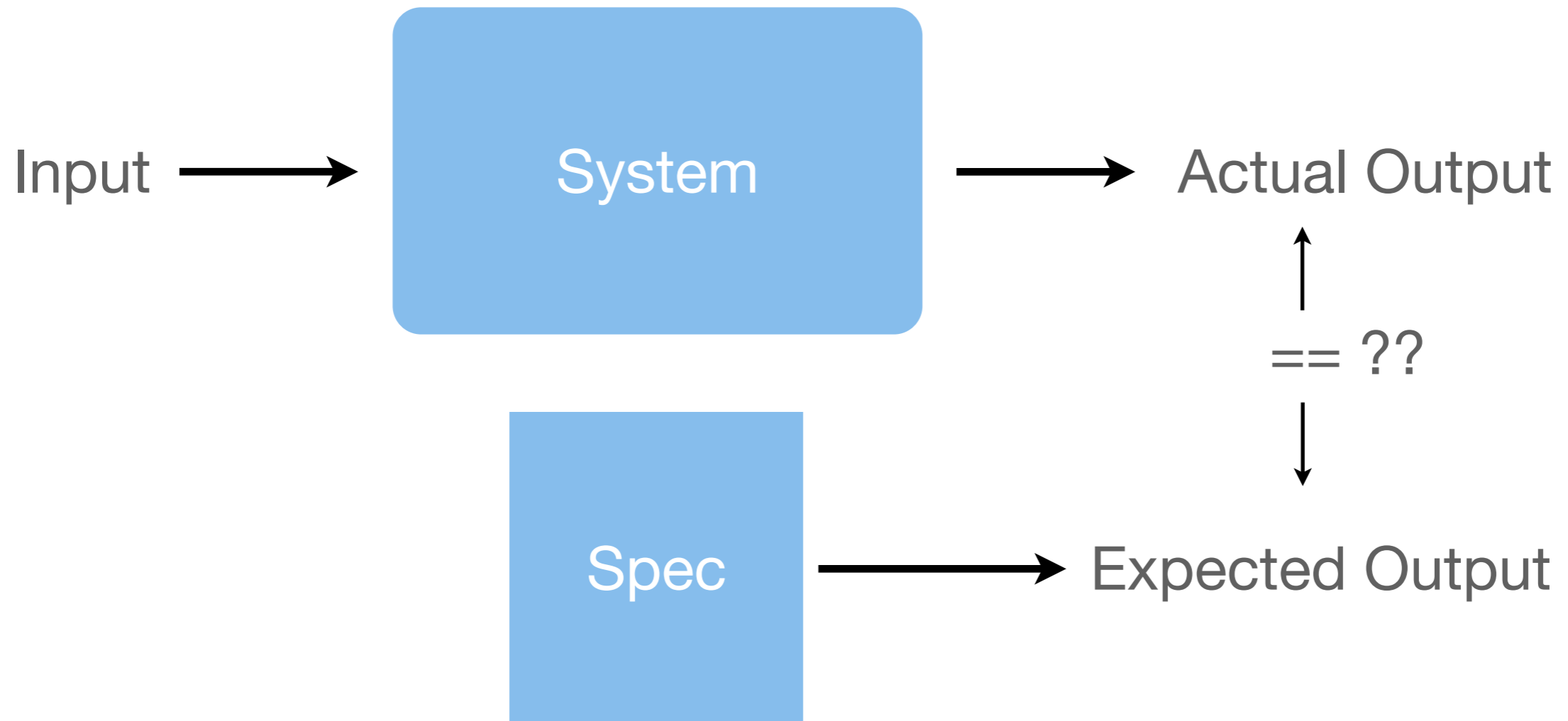
- ▶ Tests performed by the user to check that the delivered system meets their needs
  - ▶ In large, custom projects, developers will be on-site to install system and then respond to problems as they arise

# Multi-Level Testing

14

- ▶ Once we have code, we can perform three types of tests
  - ▶ Black Box Testing
    - ▶ Does the system behave as predicted by its specification
  - ▶ Grey Box Testing
    - ▶ Having a bit of insight into the architecture of the system, does it behave as predicted by its specification
  - ▶ White Box Testing
    - ▶ Since, we have access to most of the code, lets make sure we are covering all aspects of the code: statements, branches, ...

# Black Box Testing



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

# Results

16

- ▶ if actual output == expected output
  - ▶ TEST PASSED
- ▶ else
  - ▶ TEST FAILED
- ▶ Process
  - ▶ Write at least one test case per functional capability
  - ▶ Iterate on code until all tests pass
- ▶ Need to automate this process as much as possible



# Black Box Categories

17

- ▶ **Functionality**
  - ▶ User input validation (based off specification)
  - ▶ Output results
  - ▶ State transitions
    - ▶ are there clear states in the system in which the system is supposed to behave differently based on the state?
  - ▶ Boundary cases and off-by-one errors

# Grey Box Testing

18

- ▶ Use knowledge of system's architecture to create a more complete set of black box tests
  - ▶ Verifying auditing and logging information
    - ▶ for each function is the system really updating all internal state correctly
  - ▶ Data destined for other systems
  - ▶ System-added information (timestamps, checksums, etc.)
  - ▶ “Looking for Scraps”
    - ▶ Is the system correctly cleaning up after itself
      - ▶ temporary files, memory leaks, data duplication/deletion

# White Box Testing

19

- ▶ Writing test cases with complete knowledge of code
  - ▶ Format is the same: input, expected output, actual output
- ▶ But, now we are looking at
  - ▶ code coverage (more on this in a minute)
  - ▶ 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 (I)

20

- ▶ A criteria for knowing white box testing is “complete”
  - ▶ statement coverage
    - ▶ write tests until all statements have been executed
  - ▶ branch coverage (aka 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)

# Code Coverage (II)

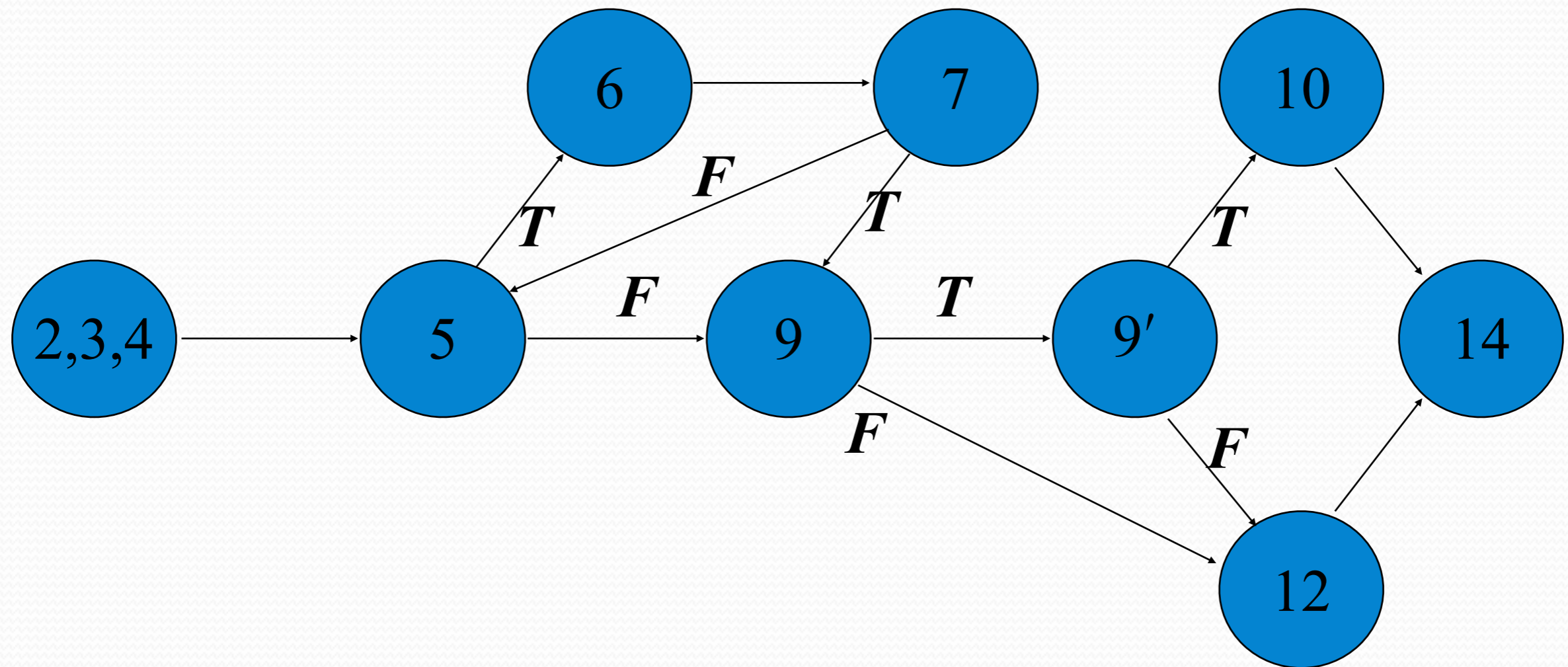
21

- ▶ A criteria for knowing white box testing is “complete”
  - ▶ 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)

# A Sample Ada Program to Test

```
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;
```

# P's Control Flow Graph (CFG)



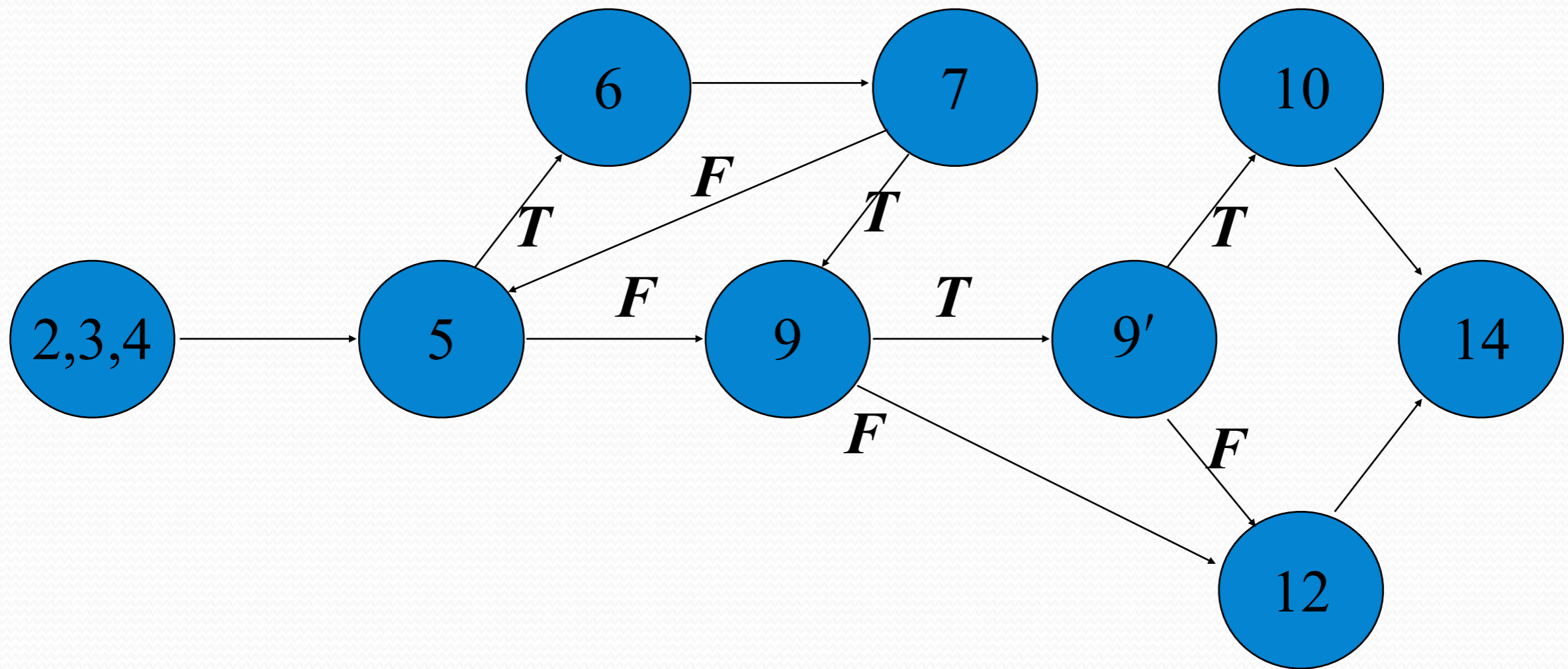
# White-box Testing Criteria

- Statement Coverage

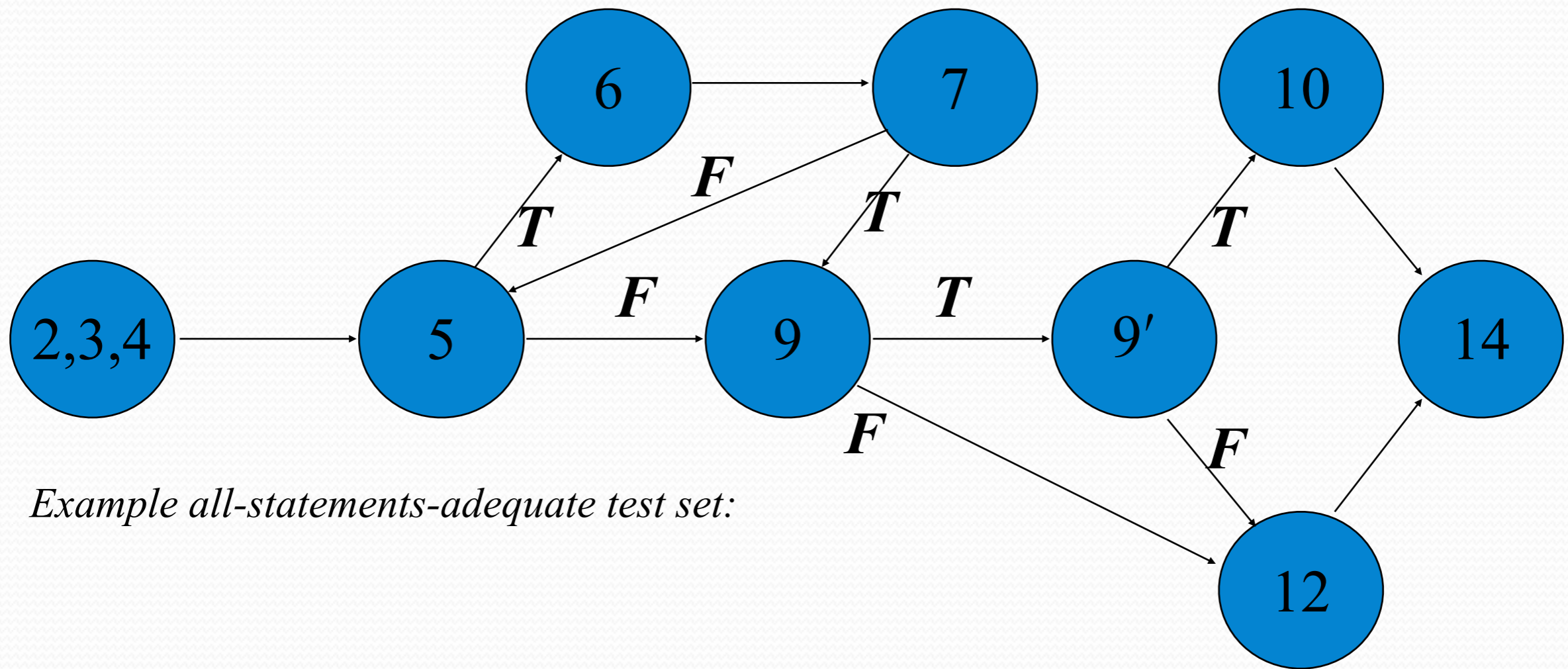
Select a test set  $T$  such that, by executing  $P$  for each  $d$  in  $T$ , each elementary statement of  $P$  is executed at least once



# All-Statements Coverage of P

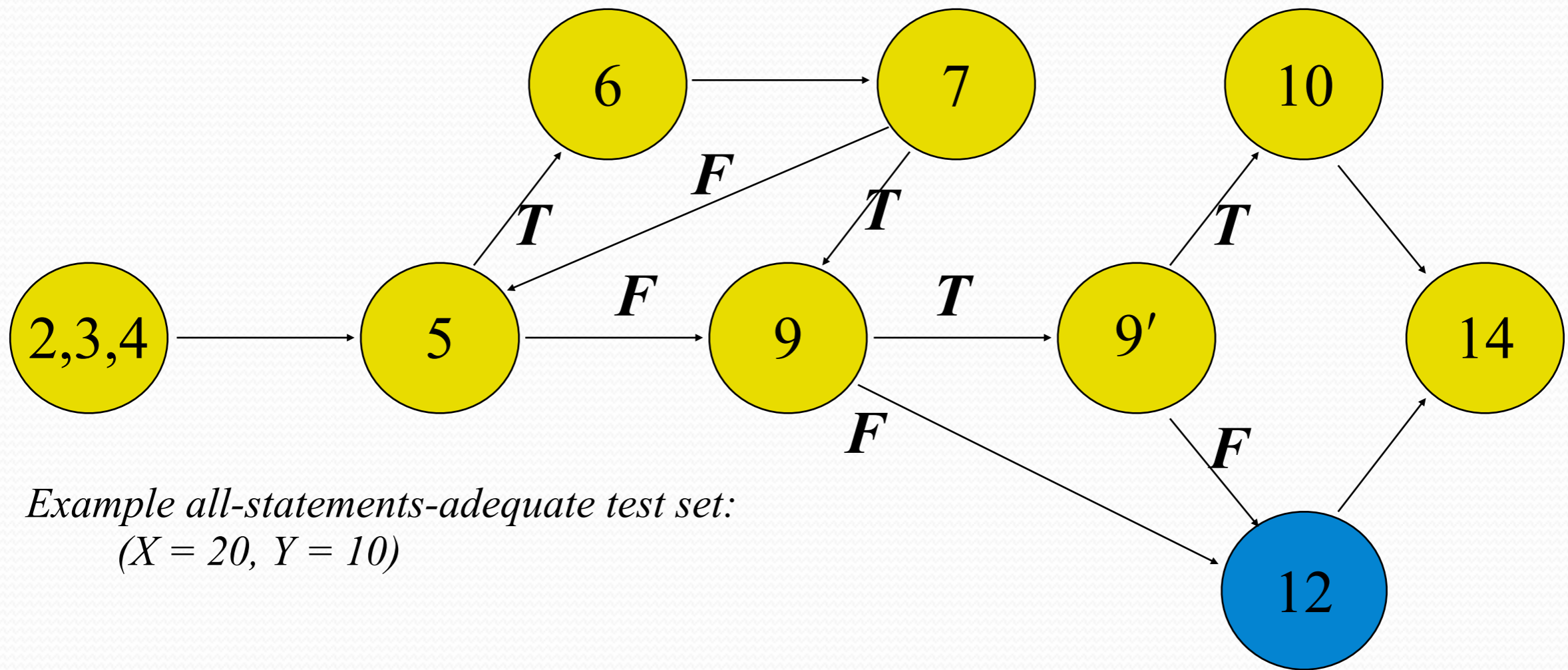


# All-Statements Coverage of P

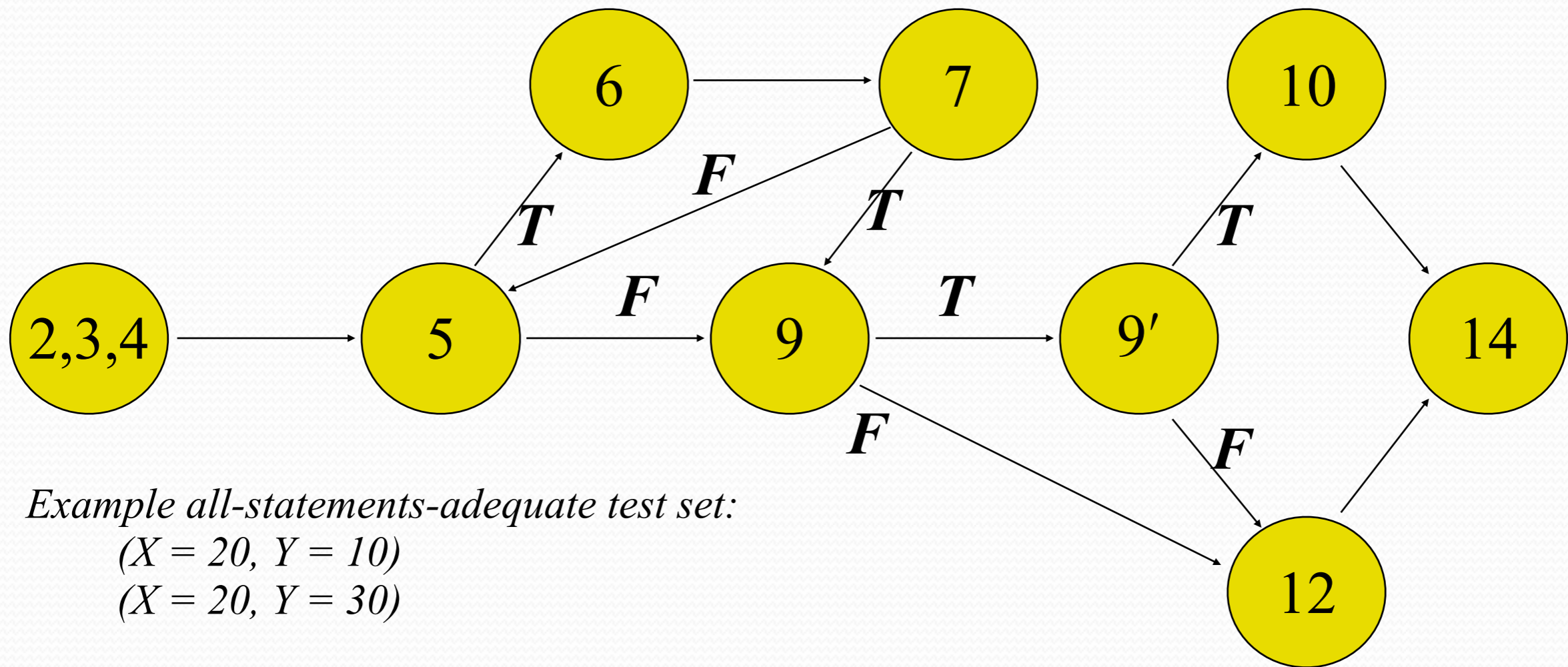


*Example all-statements-adequate test set:*

# All-Statements Coverage of P



# All-Statements Coverage of P

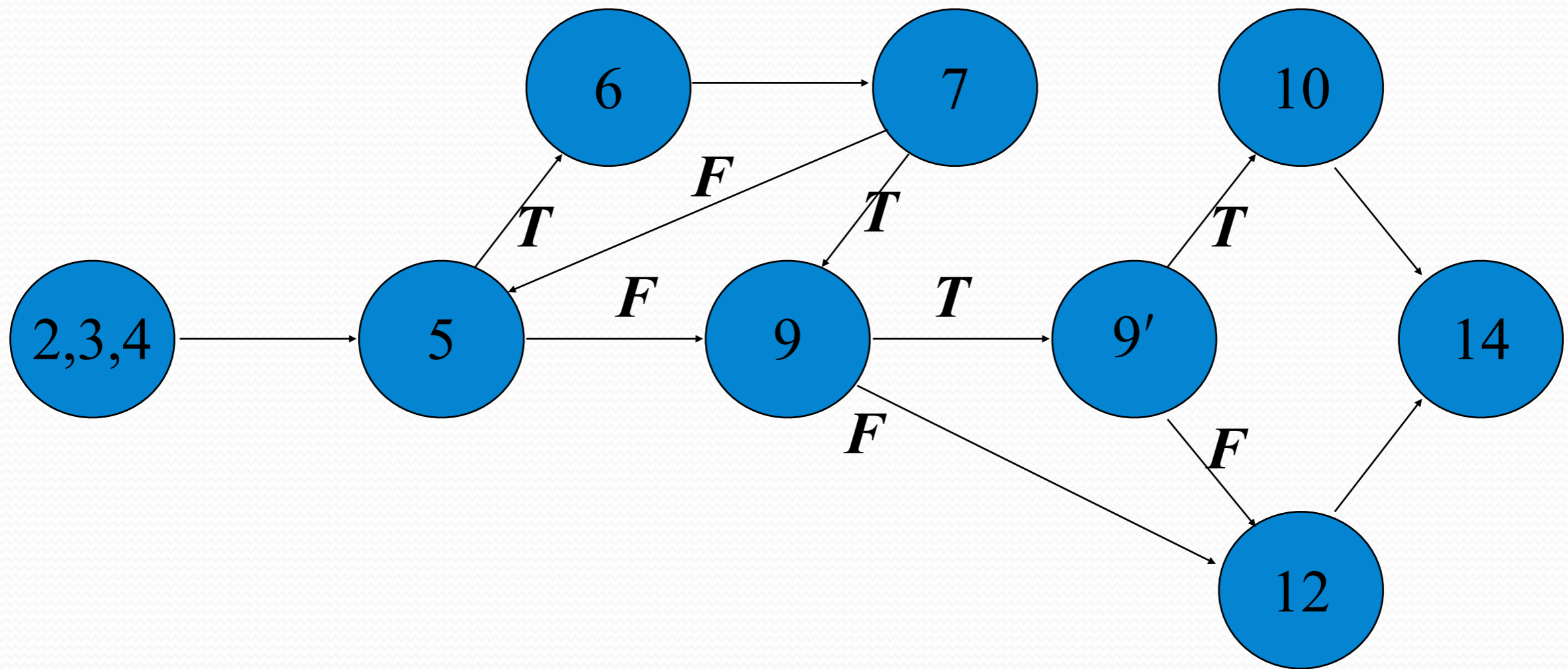


# White-box Testing Criteria

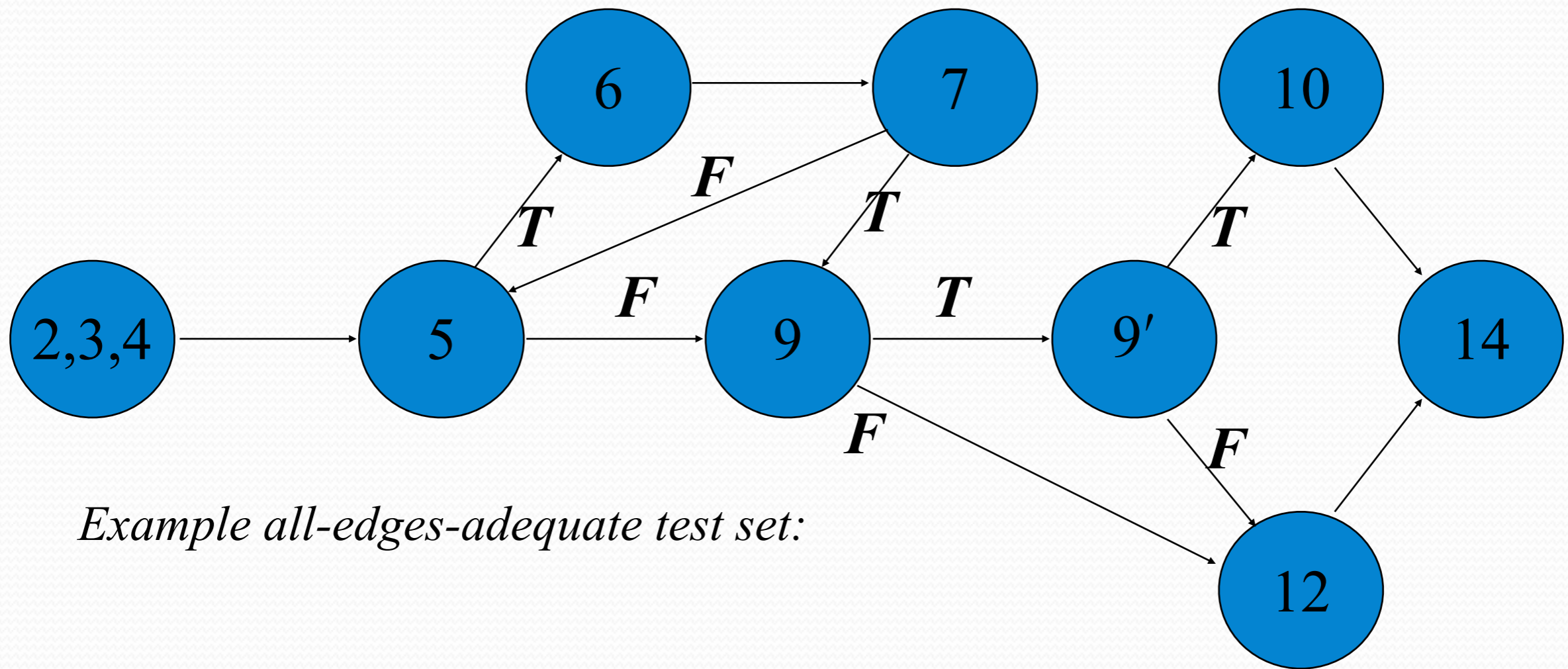
- Edge Coverage

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

# All-Edges Coverage of P

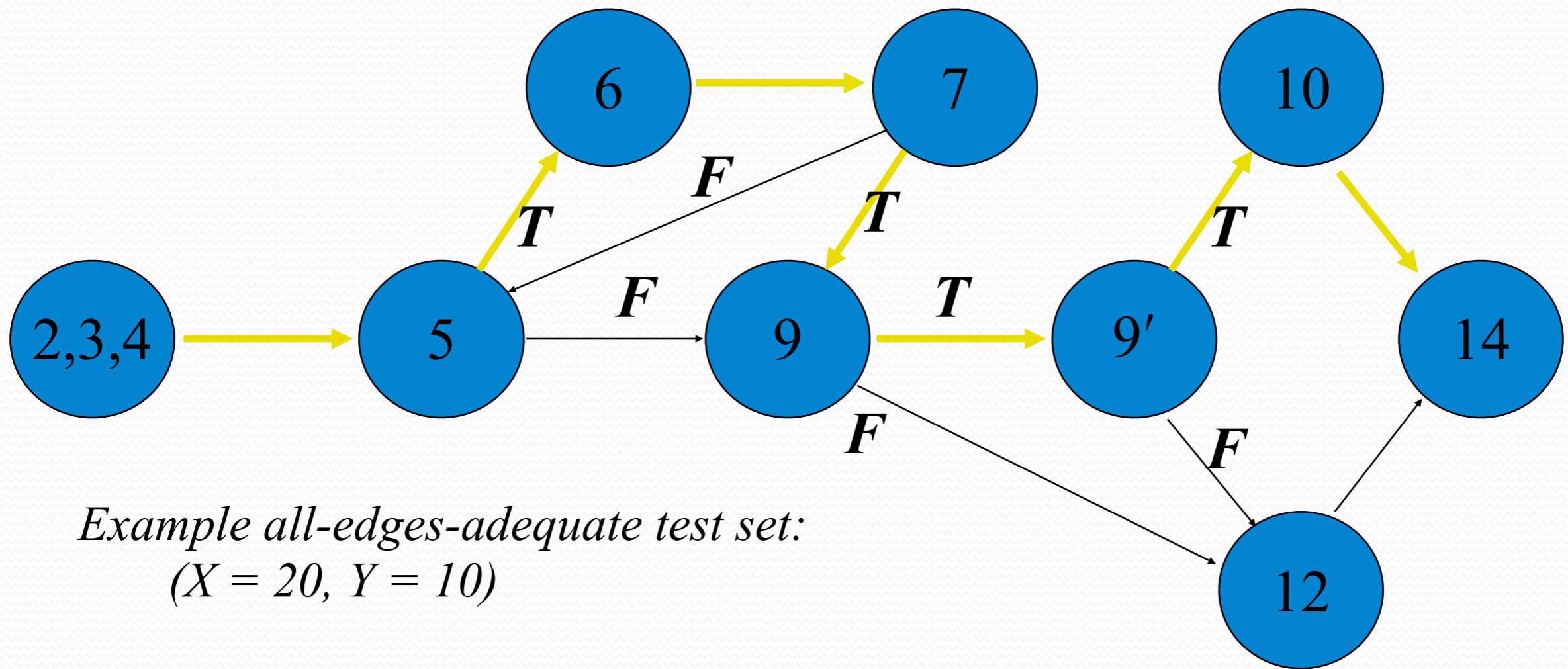


# All-Edges Coverage of P



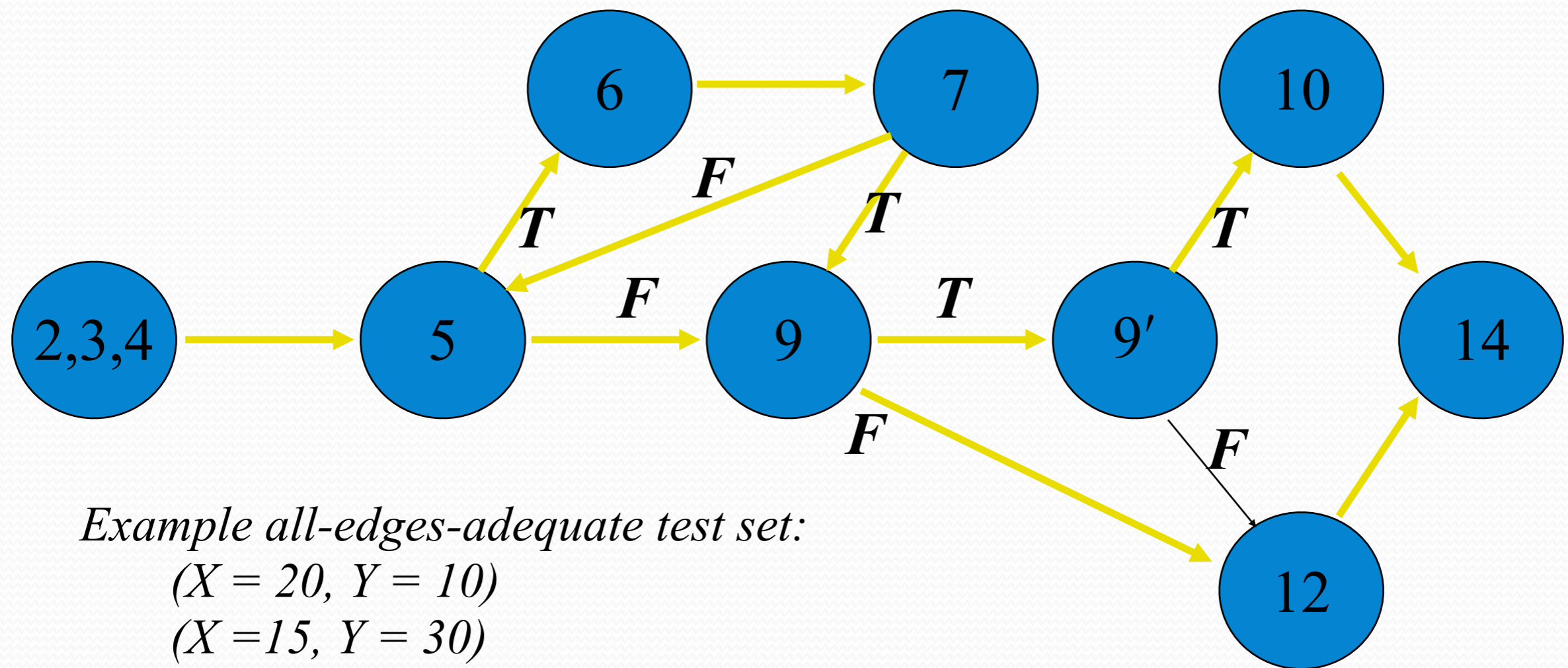
*Example all-edges-adequate test set:*

# All-Edges Coverage of P





# All-Edges Coverage of P

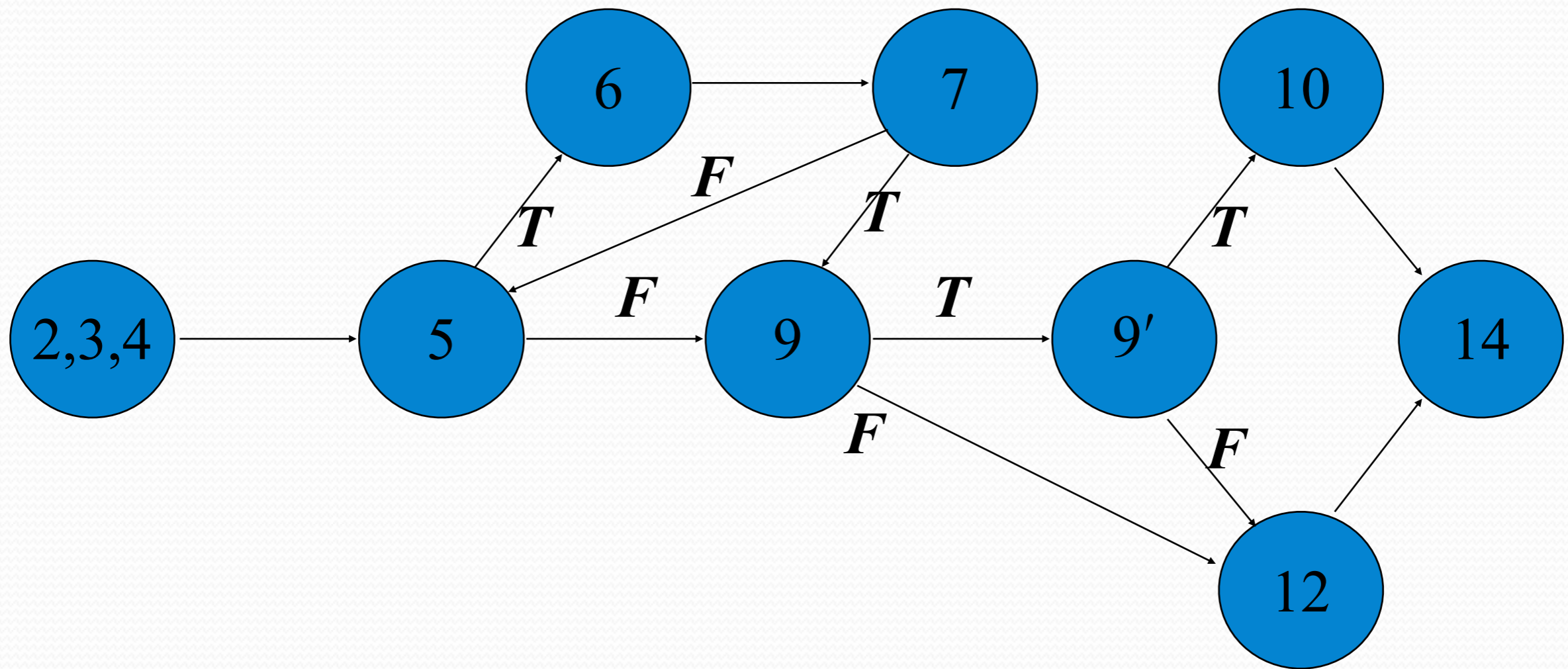


# White-box Testing Criteria

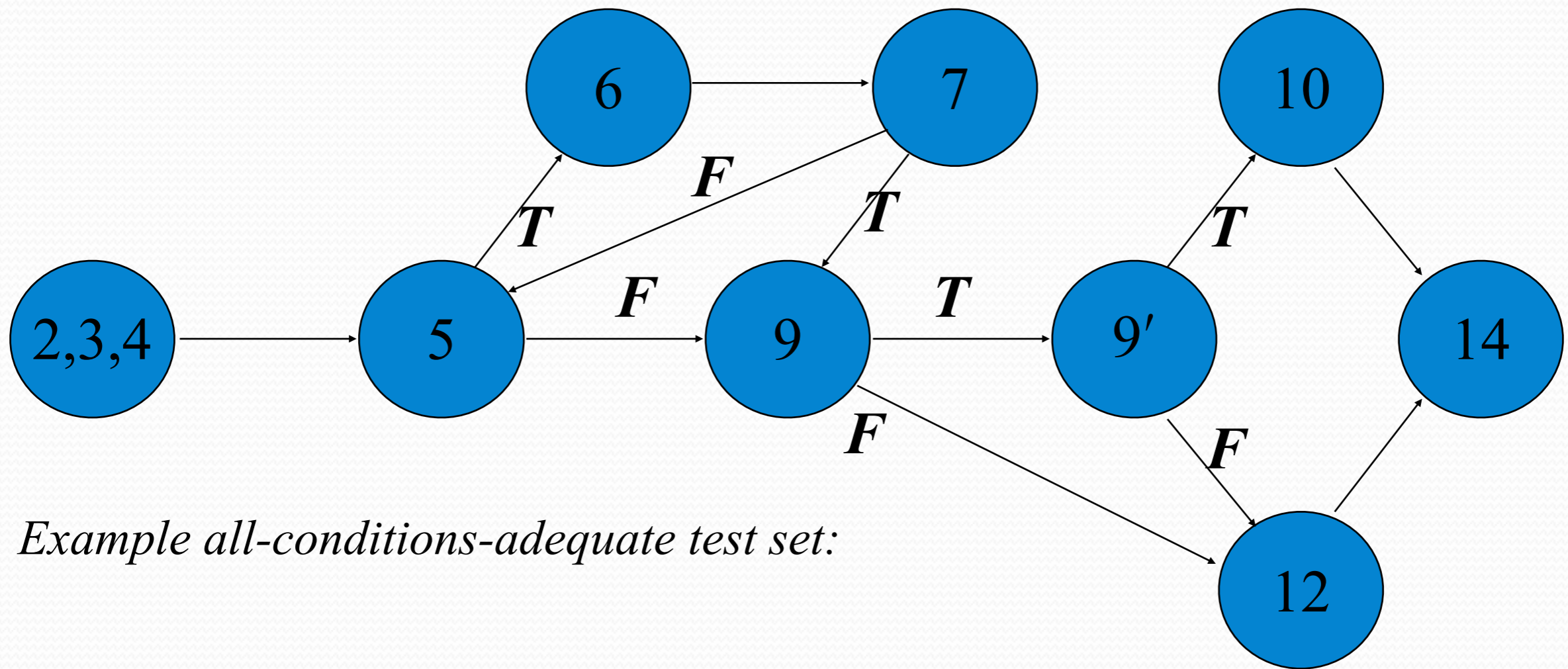
- Condition Coverage

Select a test set  $T$  such that, by executing  $P$  for each  $d$  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

# All-Conditions Coverage of P

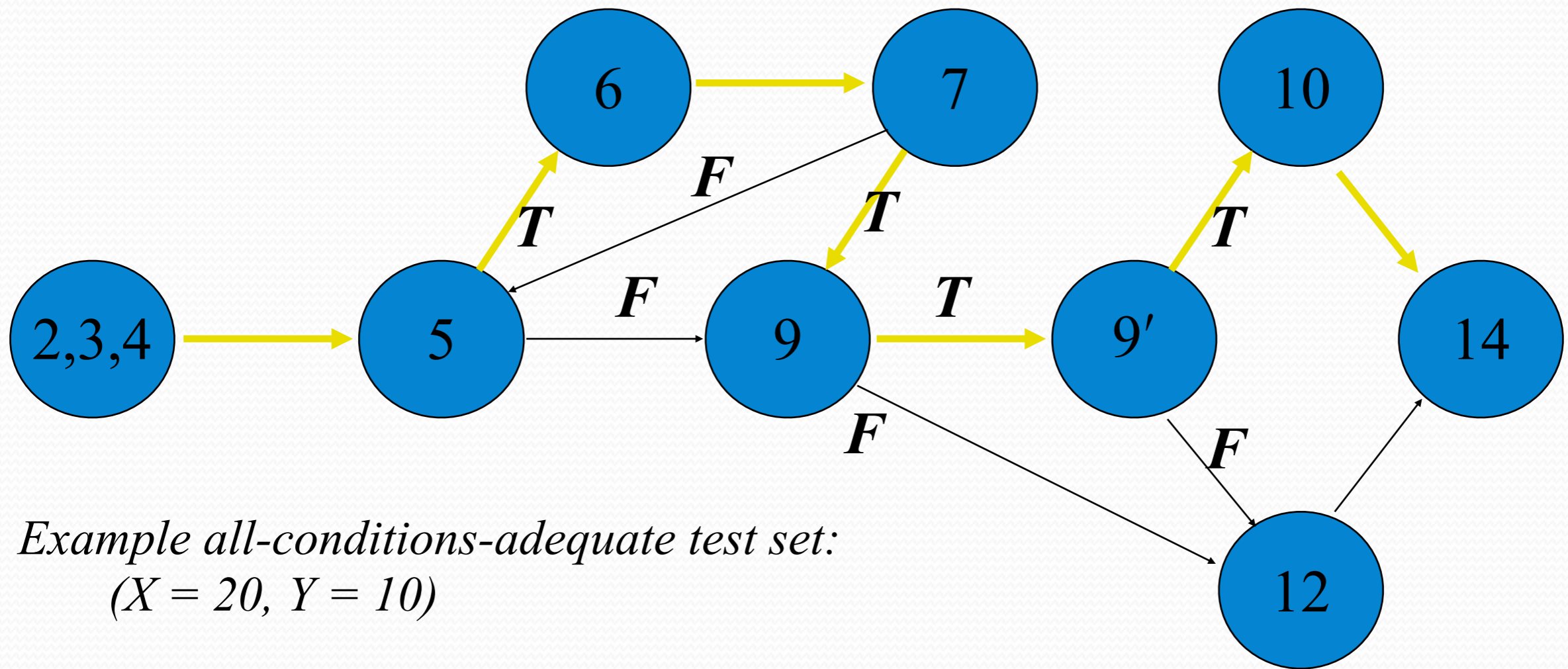


# All-Conditions Coverage of P



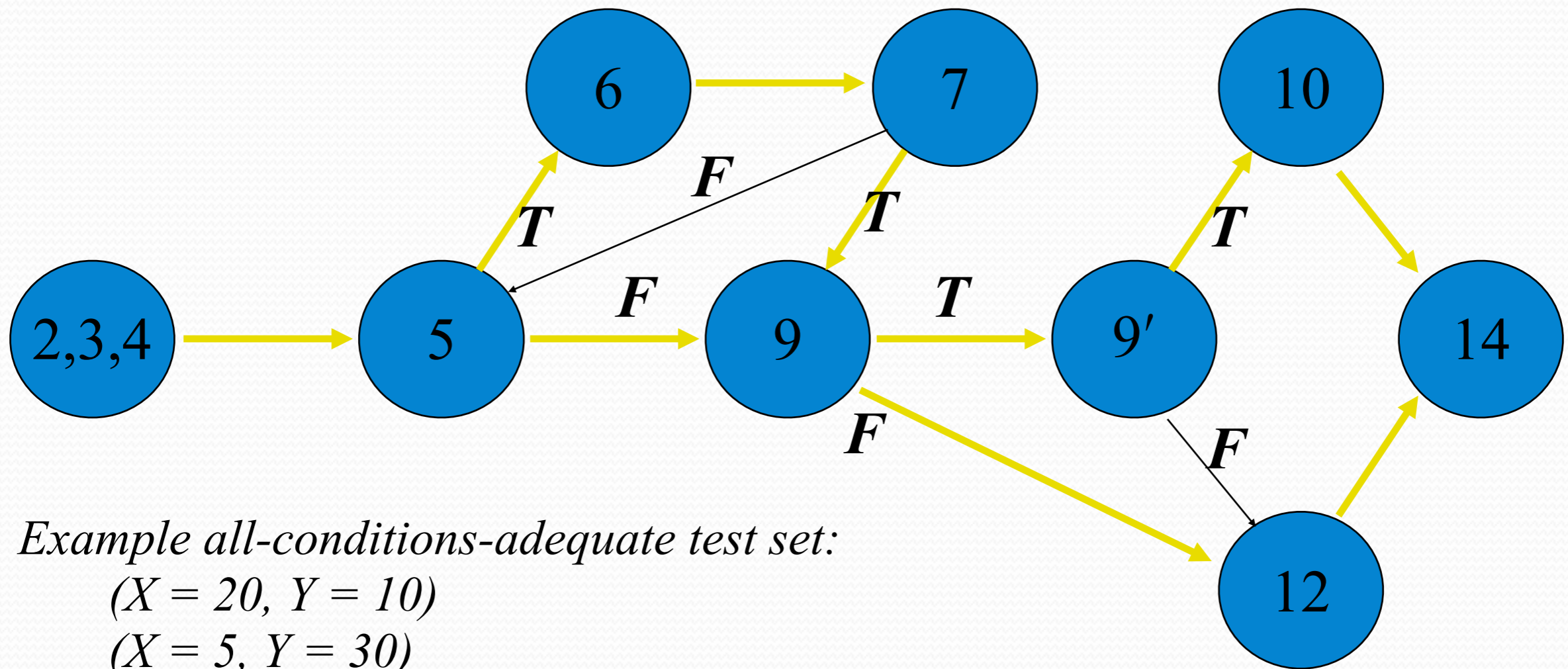
*Example all-conditions-adequate test set:*

# All-Conditions Coverage of P

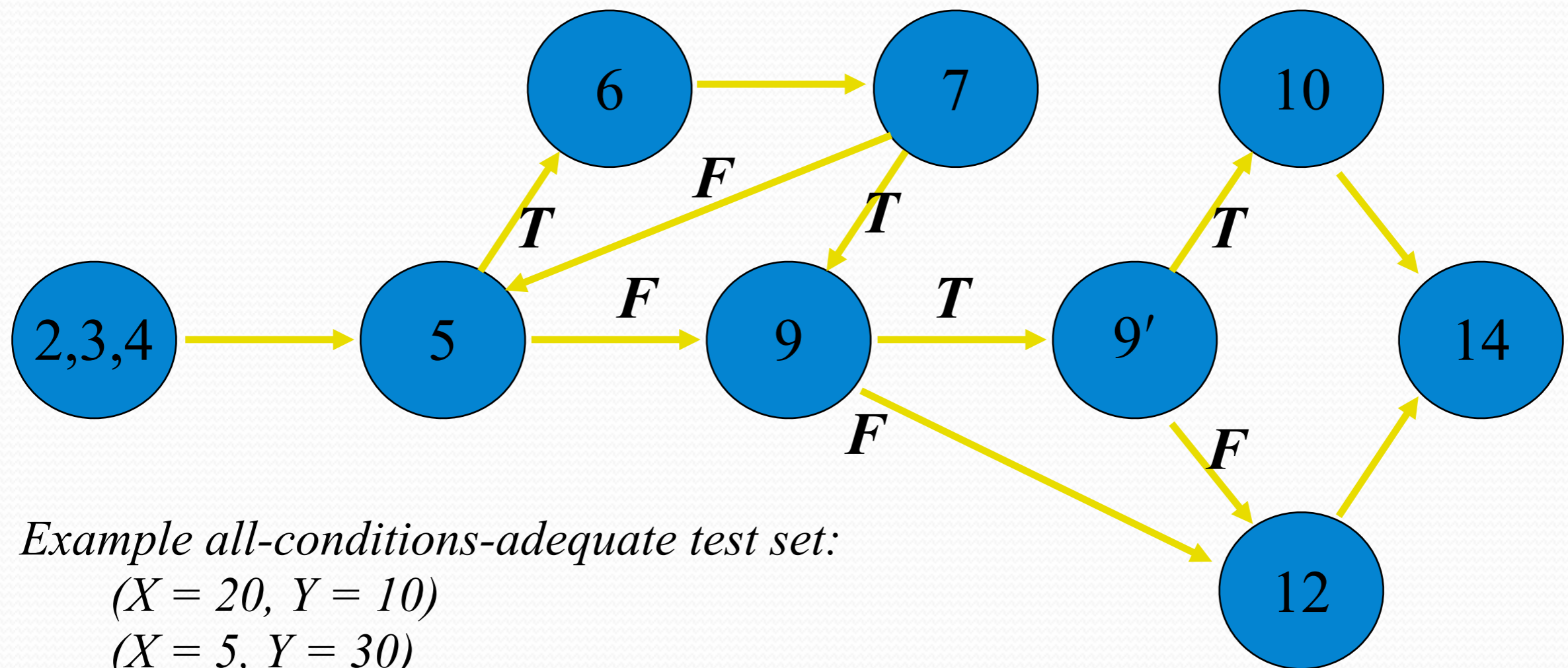


*Example all-conditions-adequate test set:  
(X = 20, Y = 10)*

# All-Conditions Coverage of P



# All-Conditions Coverage of P



*Example all-conditions-adequate test set:*

$(X = 20, Y = 10)$

$(X = 5, Y = 30)$

$(X = 21, Y = 10)$

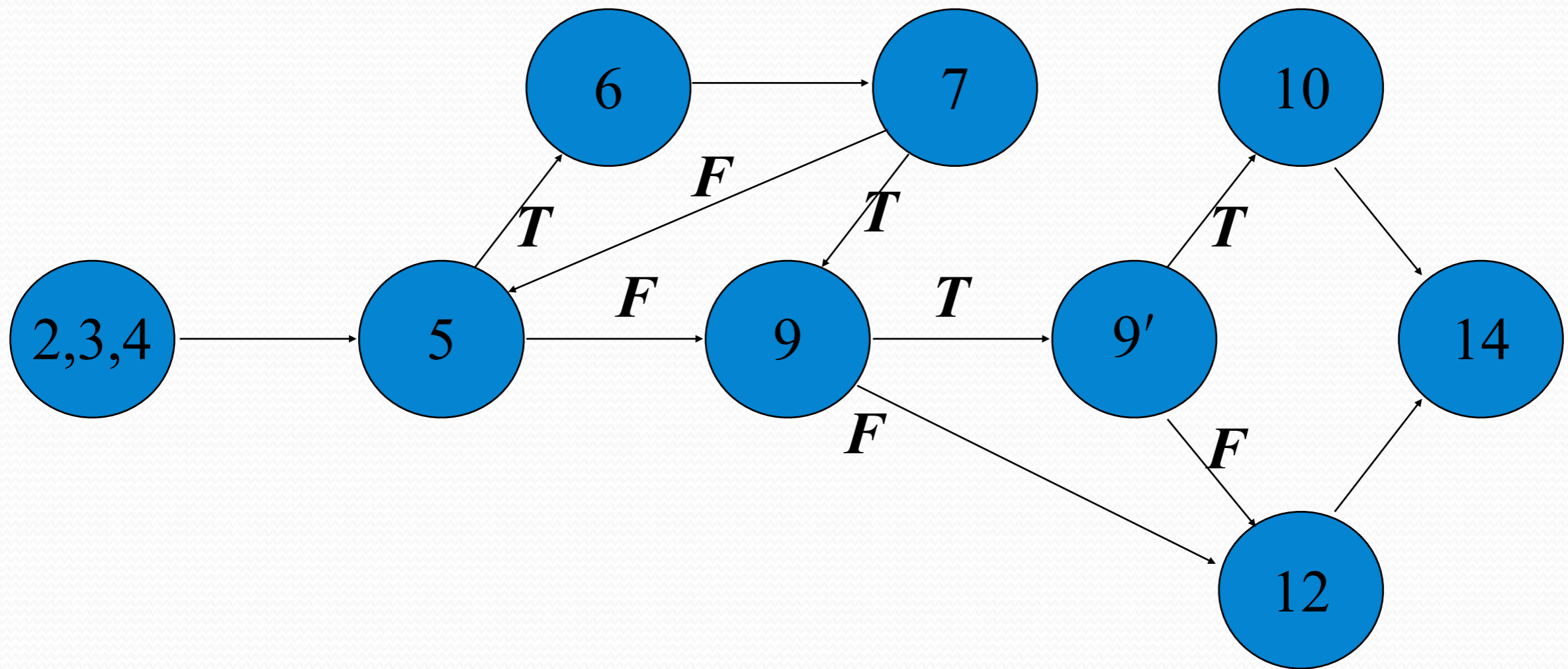
# White-box Testing Criteria

- Path Coverage

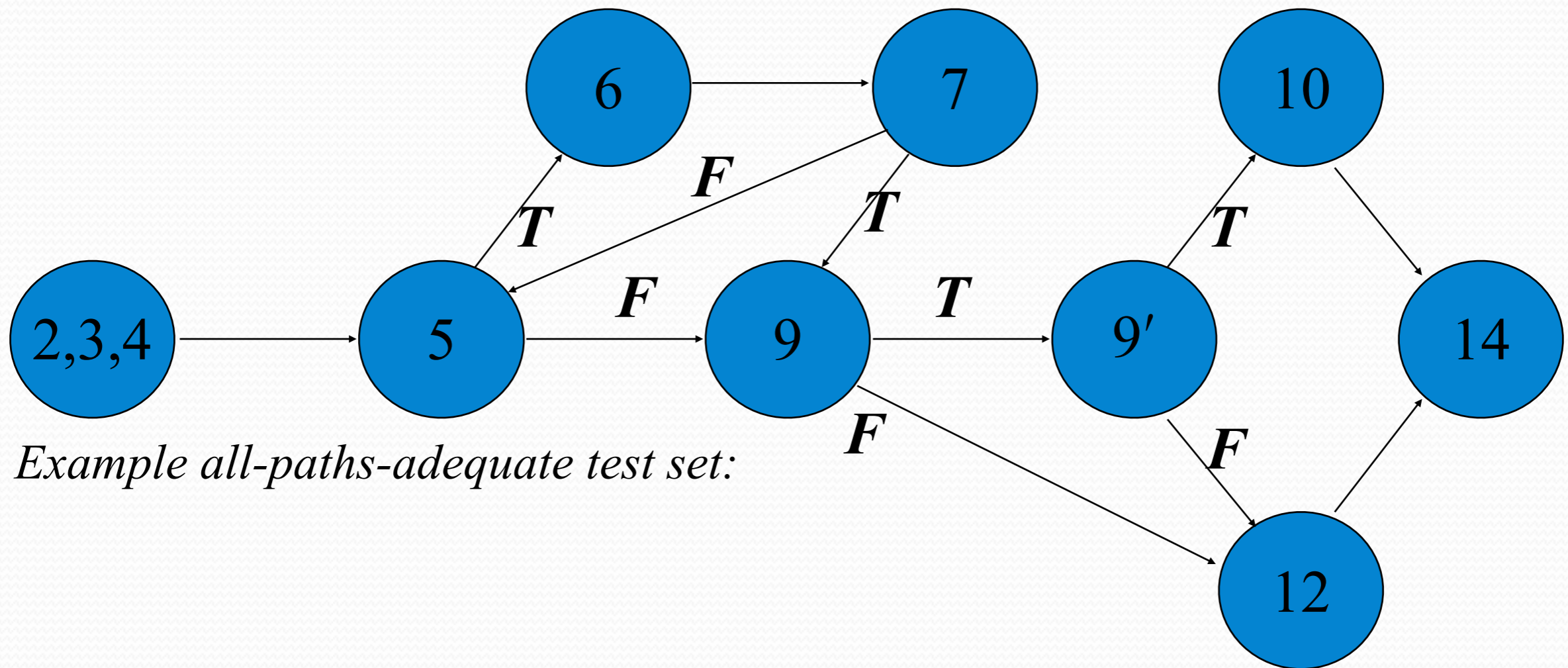
Select a test set  $T$  such that, by executing  $P$  for each  $d$  in  $T$ , all paths leading from the initial to the final node of  $P$ 's control flow graph are traversed at least once



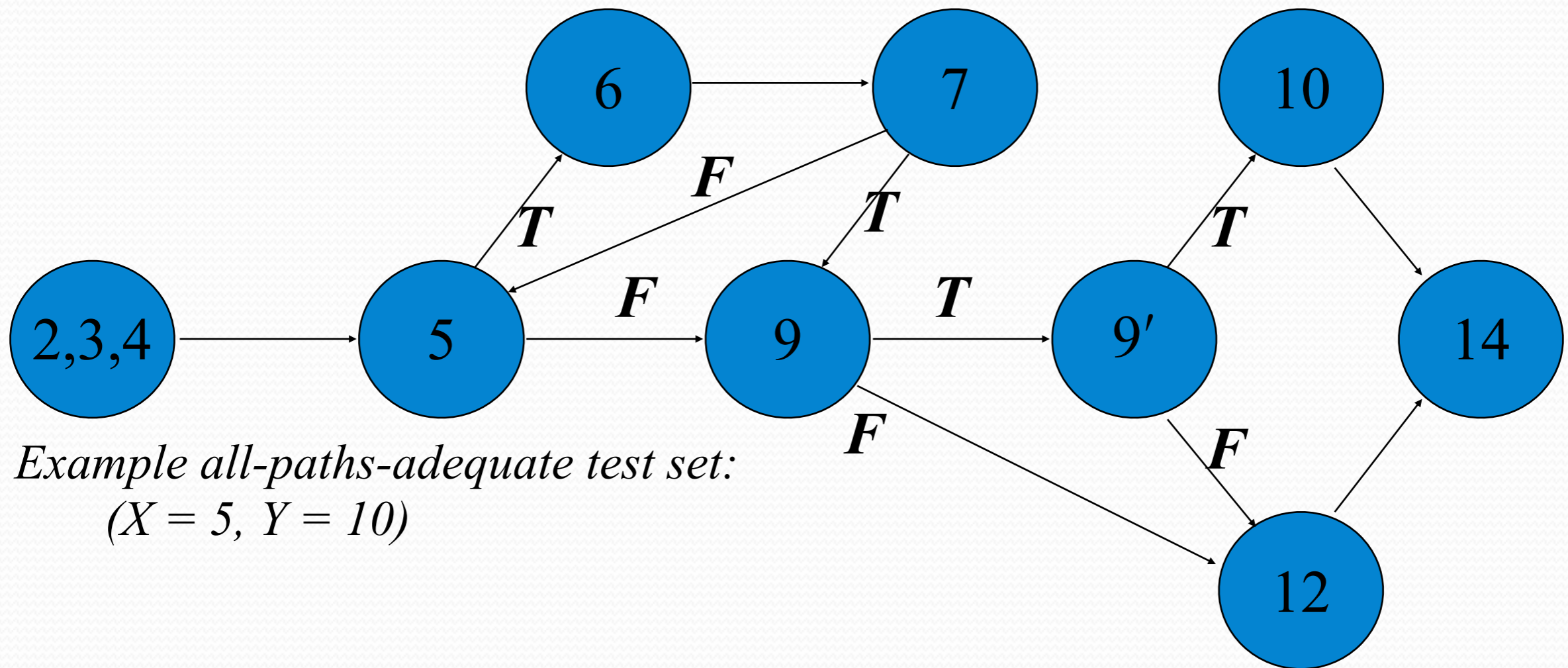
# All-Paths Coverage of P



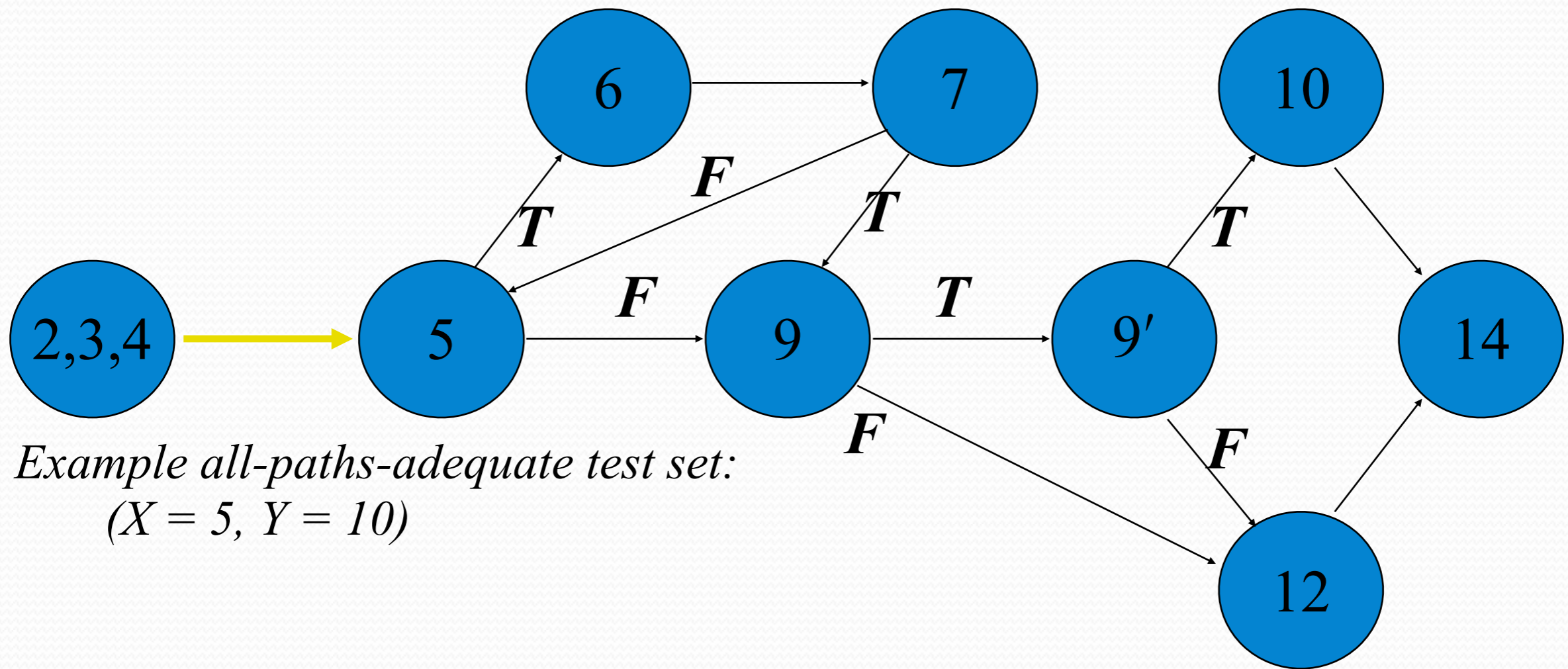
# All-Paths Coverage of P



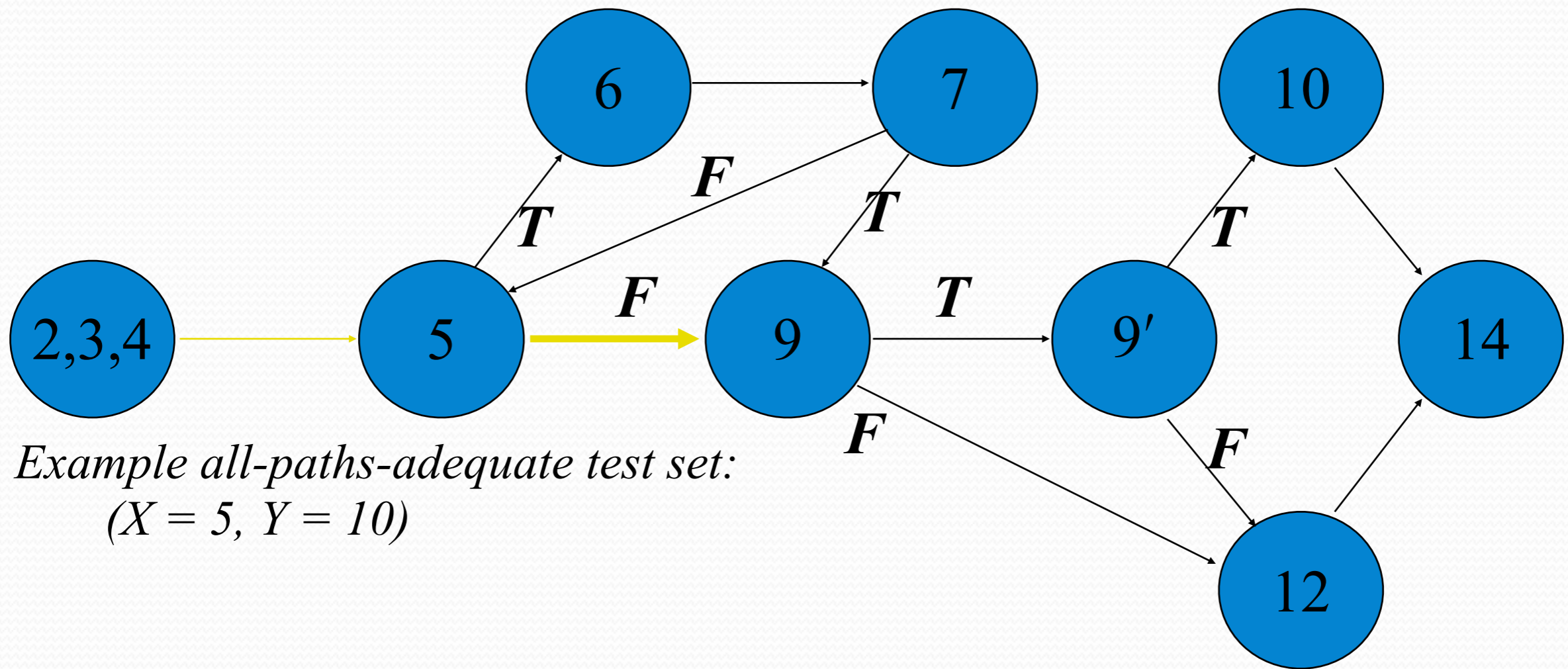
# All-Paths Coverage of P



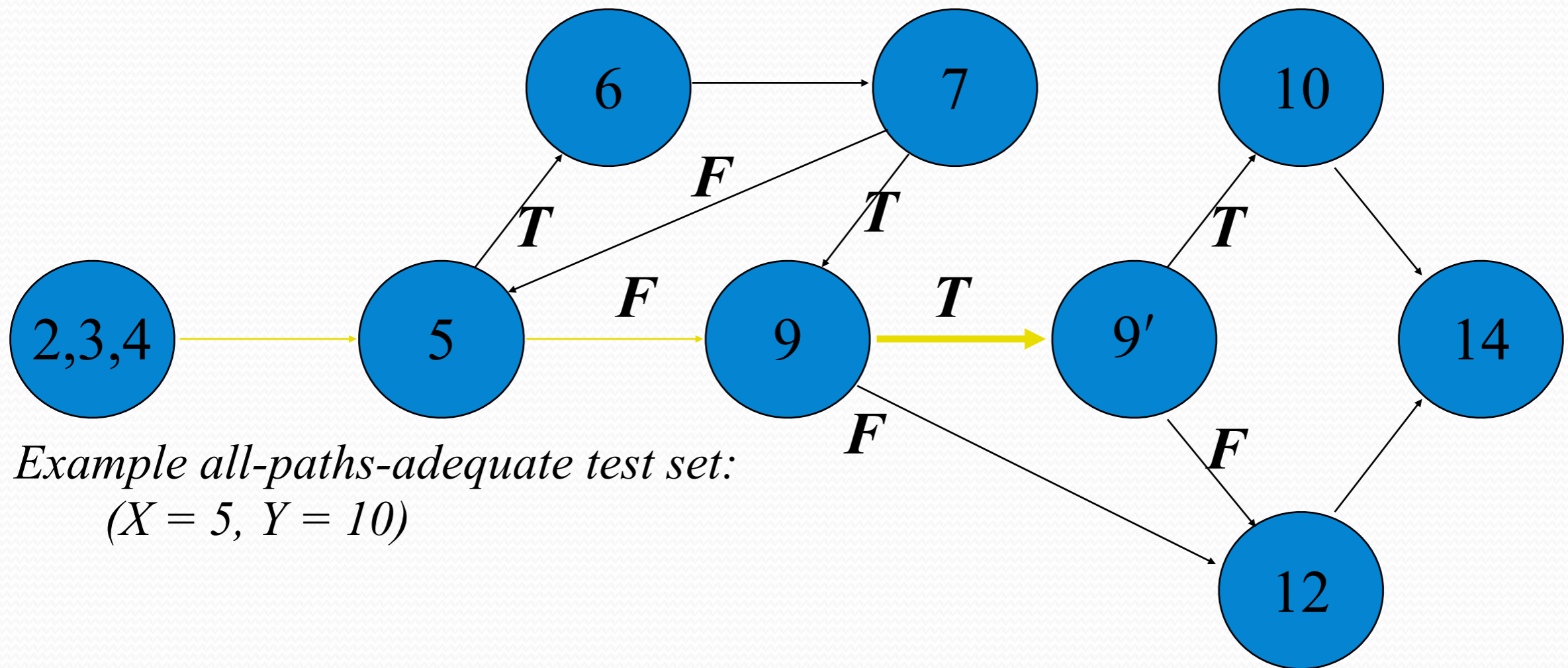
# All-Paths Coverage of P



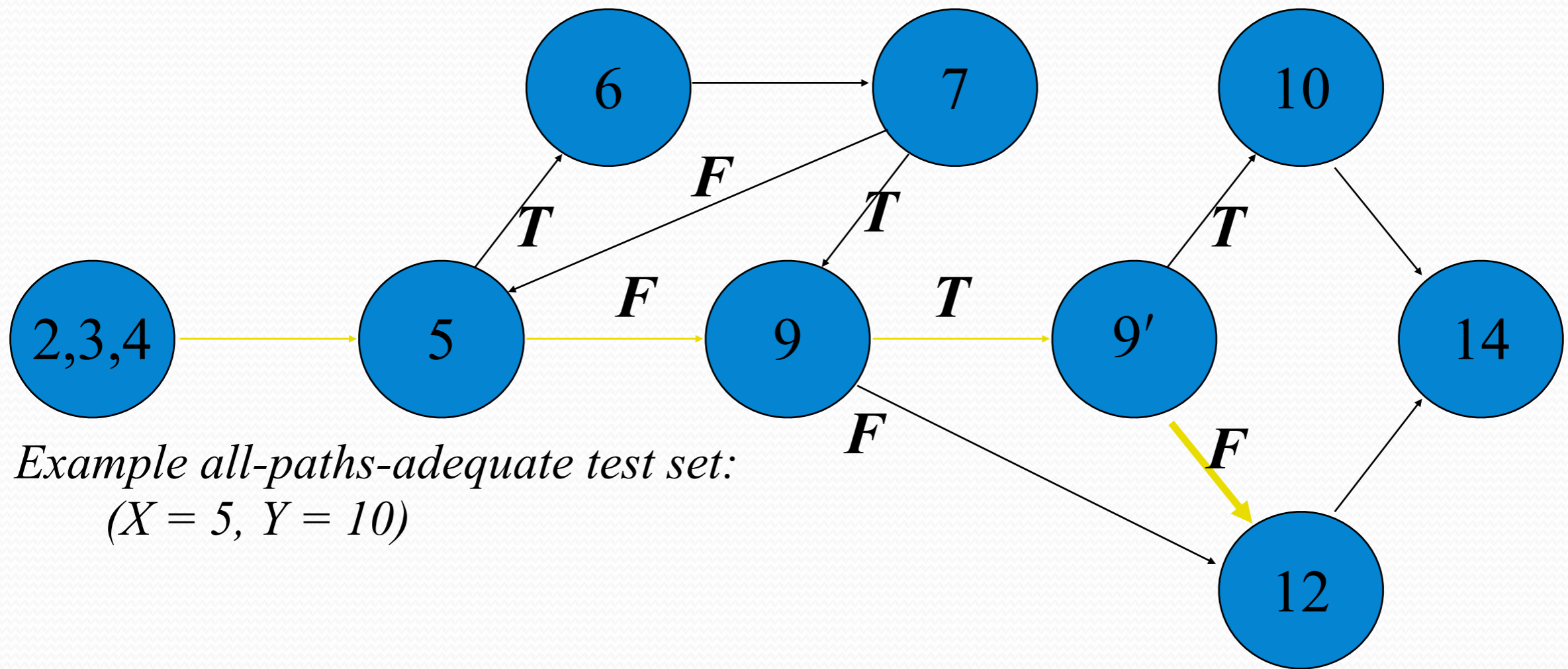
# All-Paths Coverage of P



# All-Paths Coverage of P

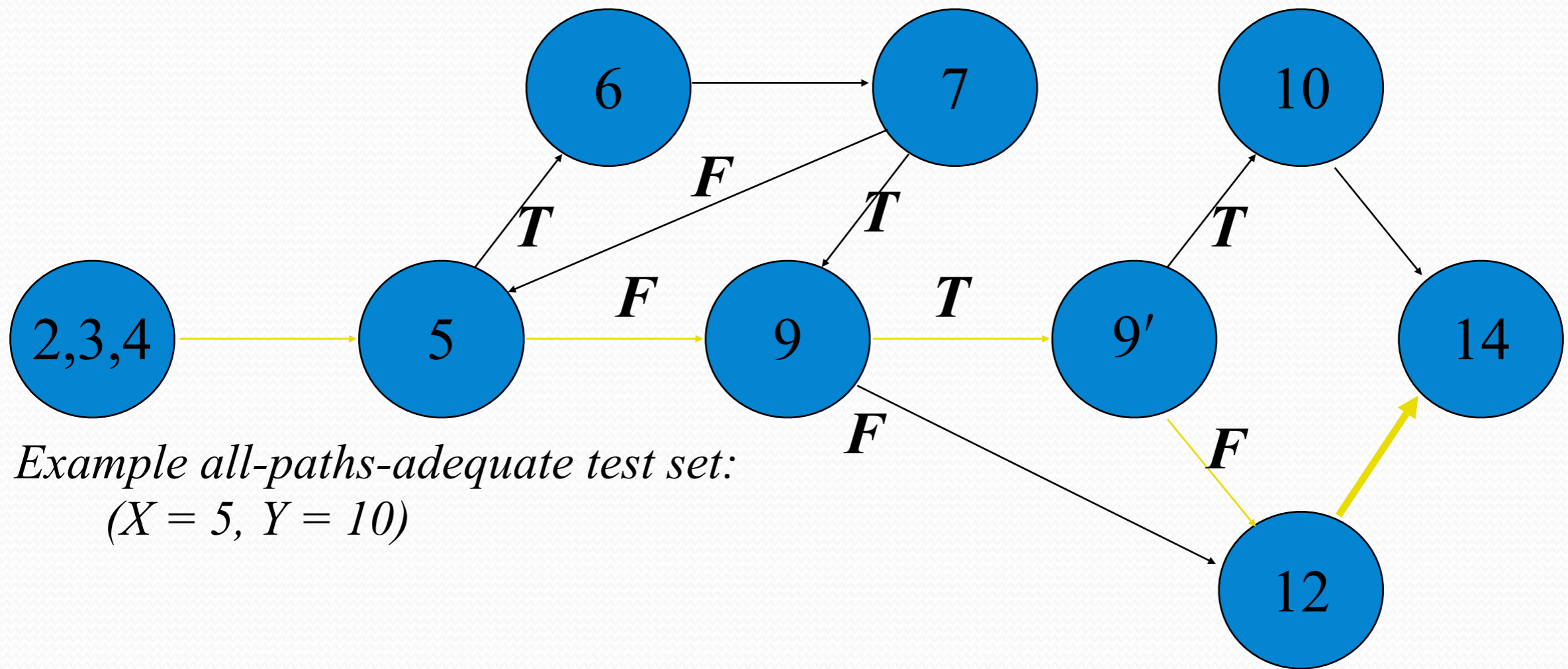


# All-Paths Coverage of P



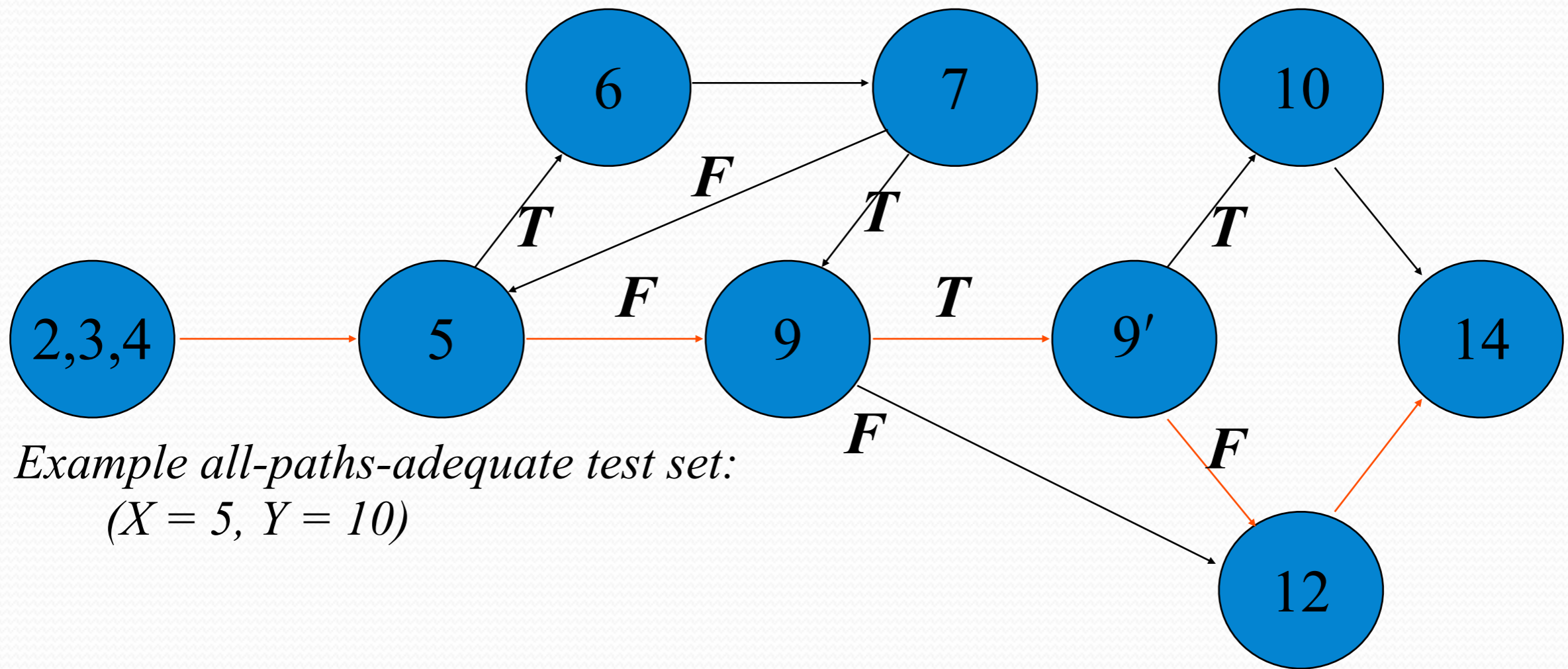
*Example all-paths-adequate test set:  
( $X = 5, Y = 10$ )*

# All-Paths Coverage of P

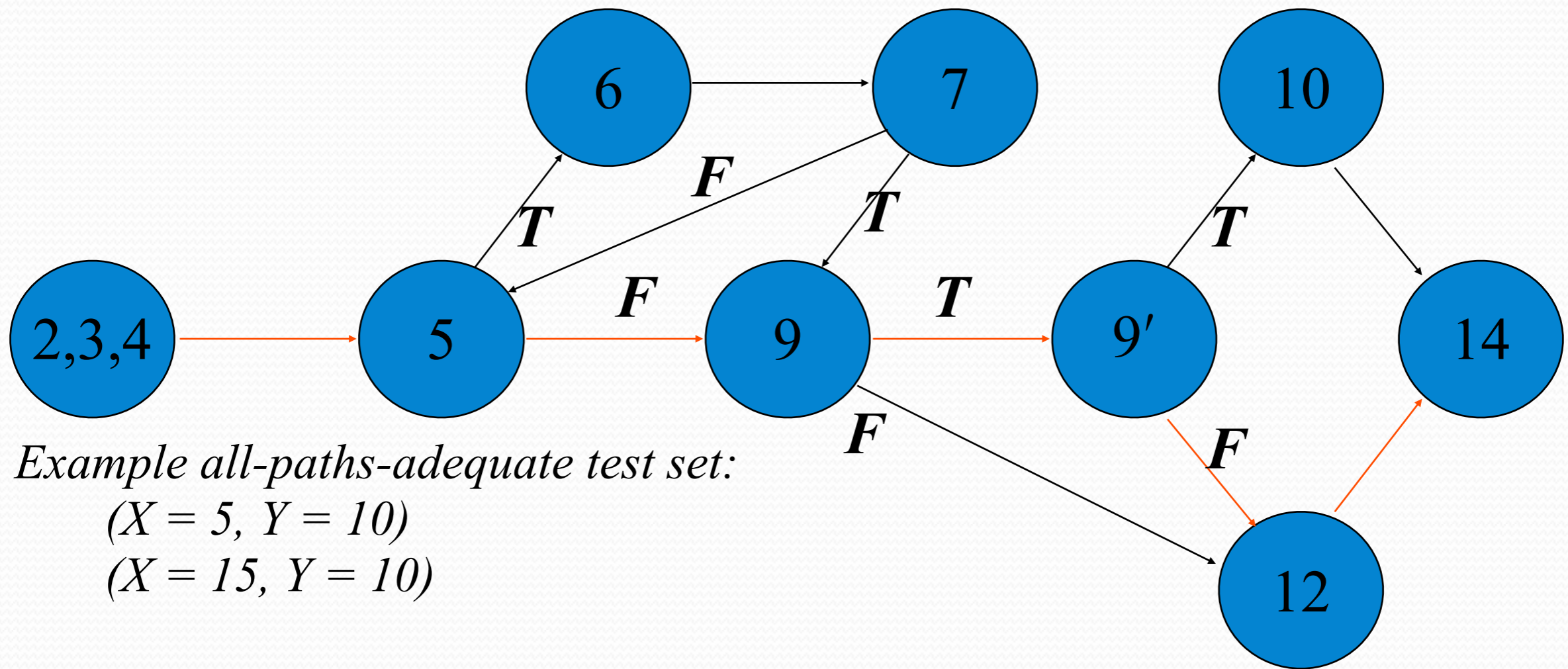




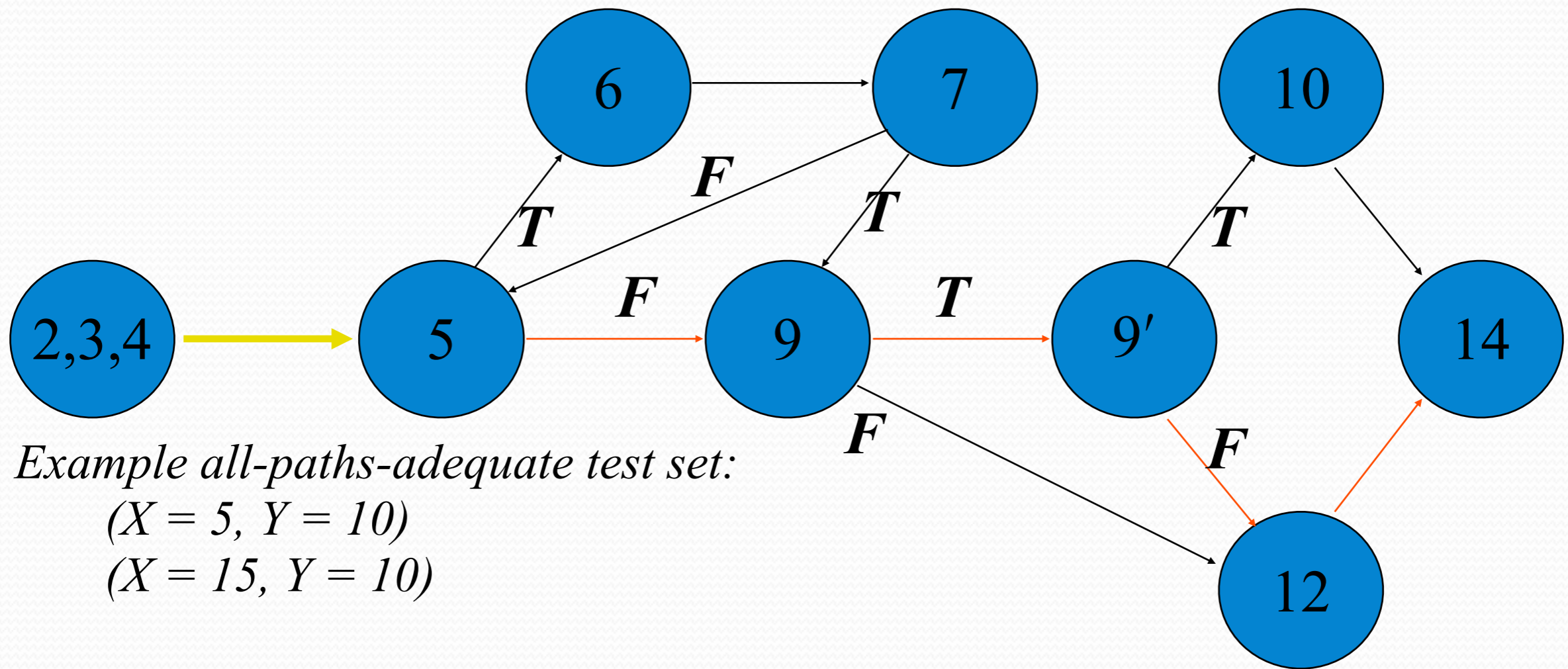
# All-Paths Coverage of P



# All-Paths Coverage of P



# All-Paths Coverage of P

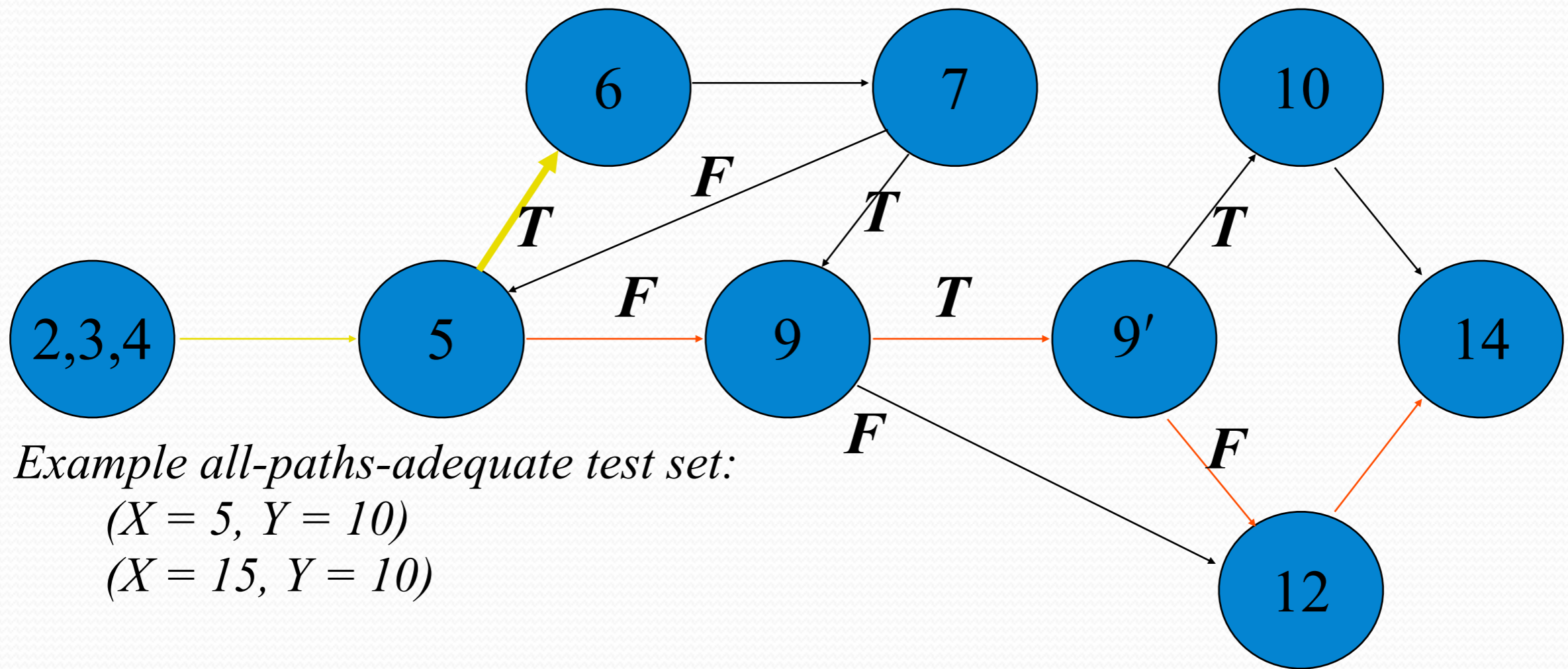


*Example all-paths-adequate test set:*

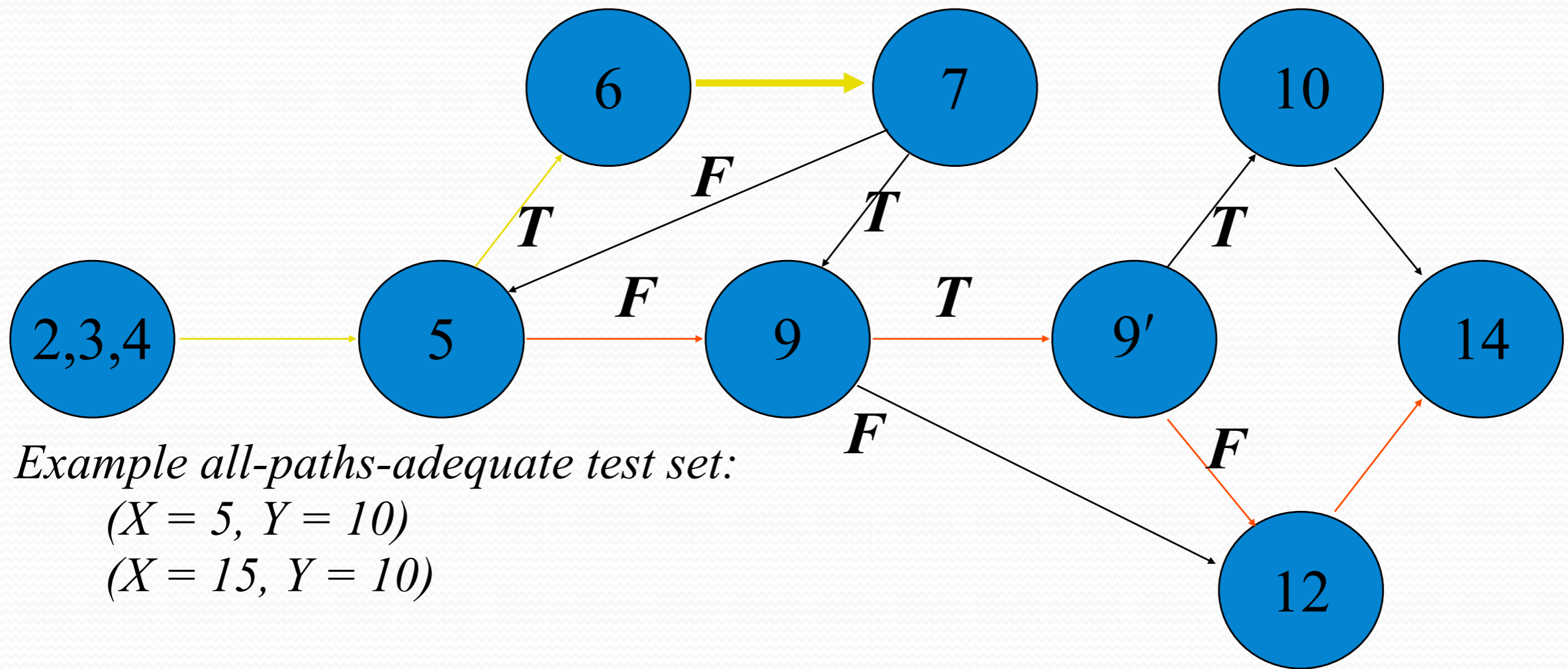
*( $X = 5, Y = 10$ )*

*( $X = 15, Y = 10$ )*

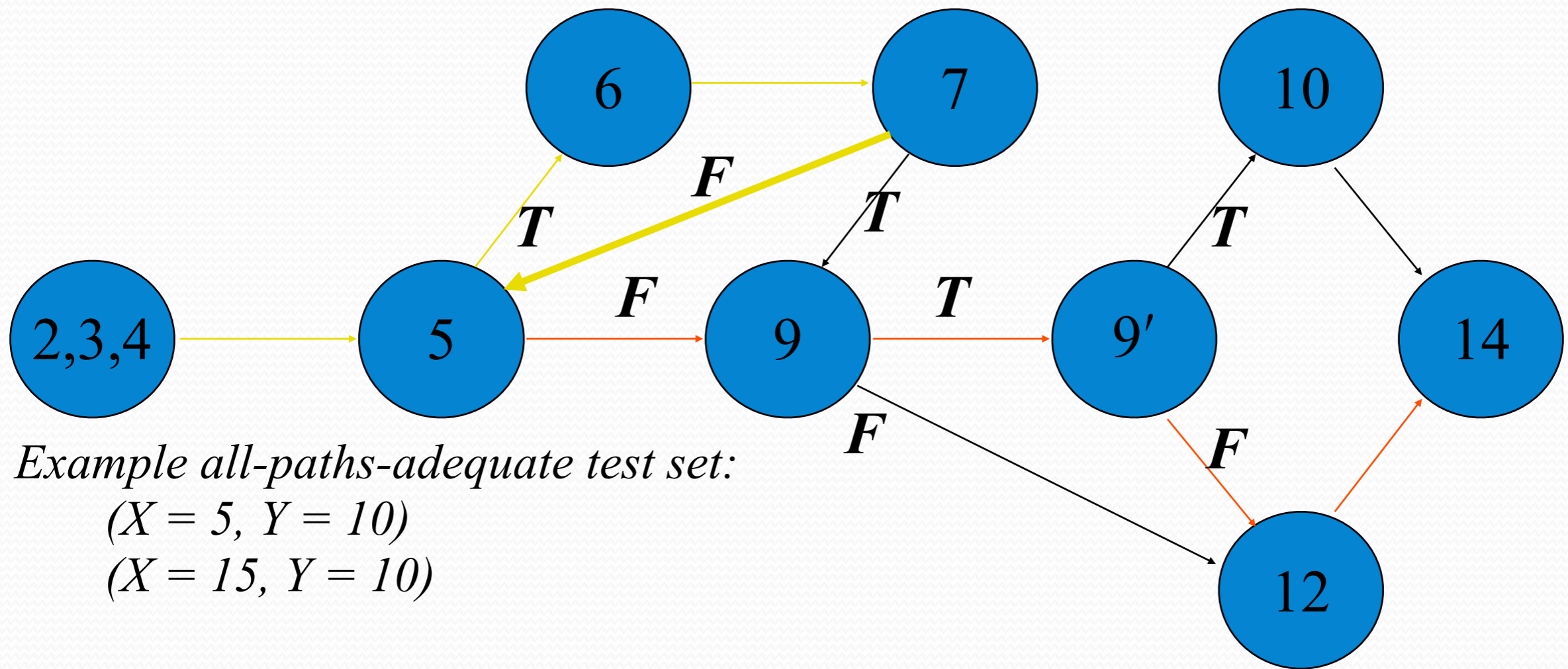
# All-Paths Coverage of P



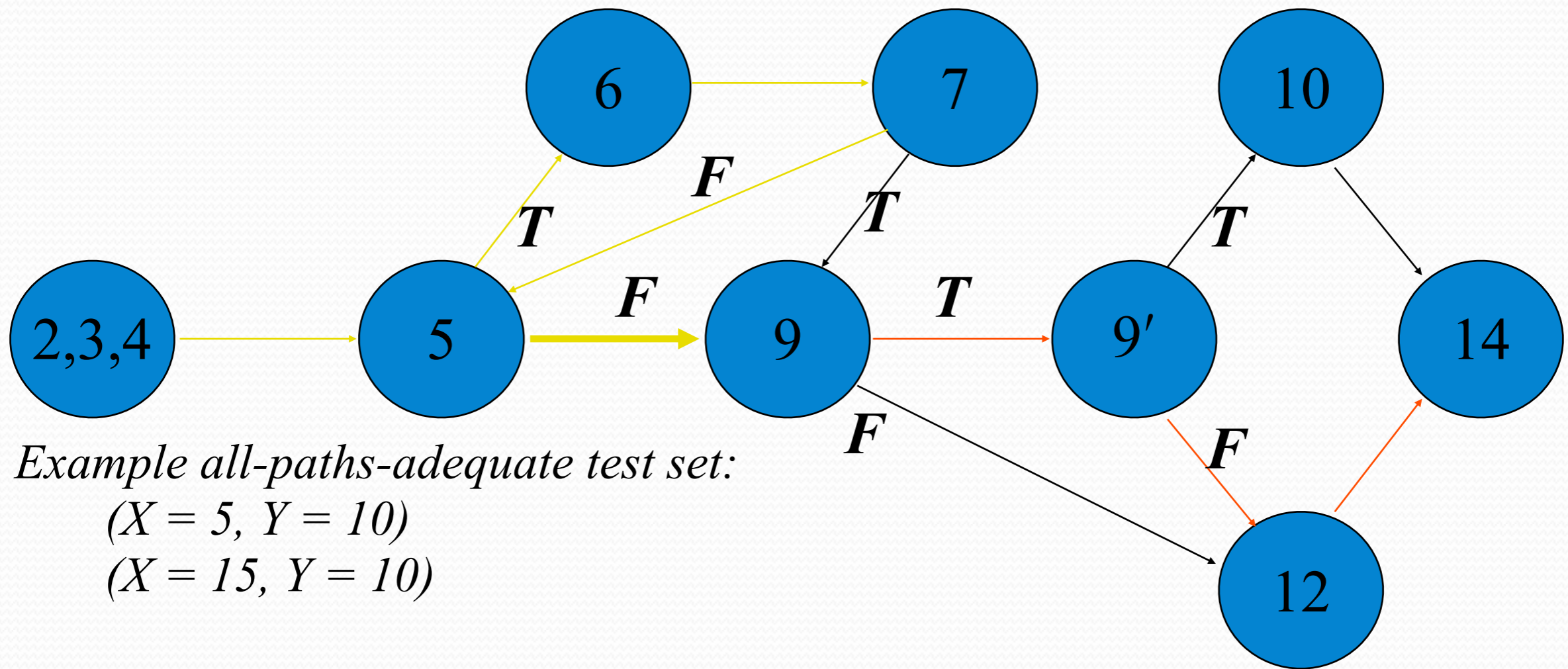
# All-Paths Coverage of P



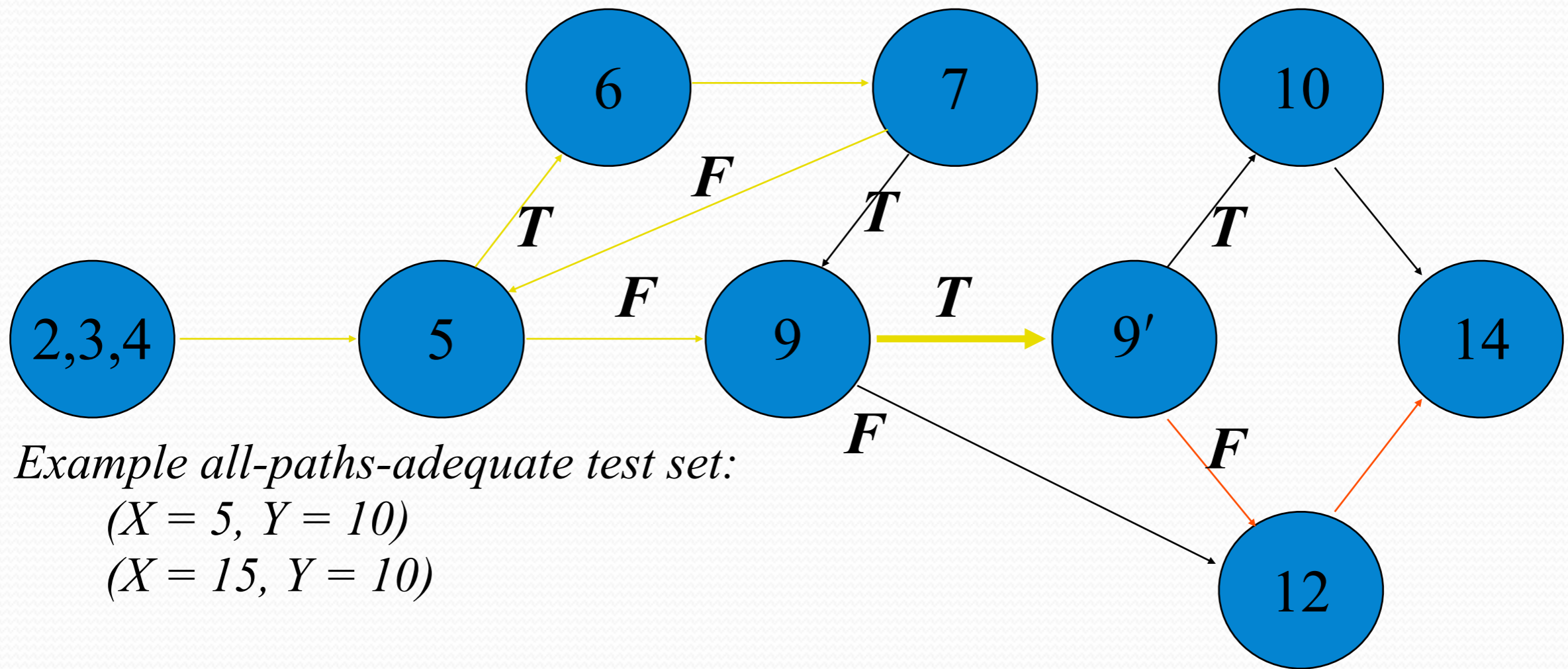
# All-Paths Coverage of P



# All-Paths Coverage of P

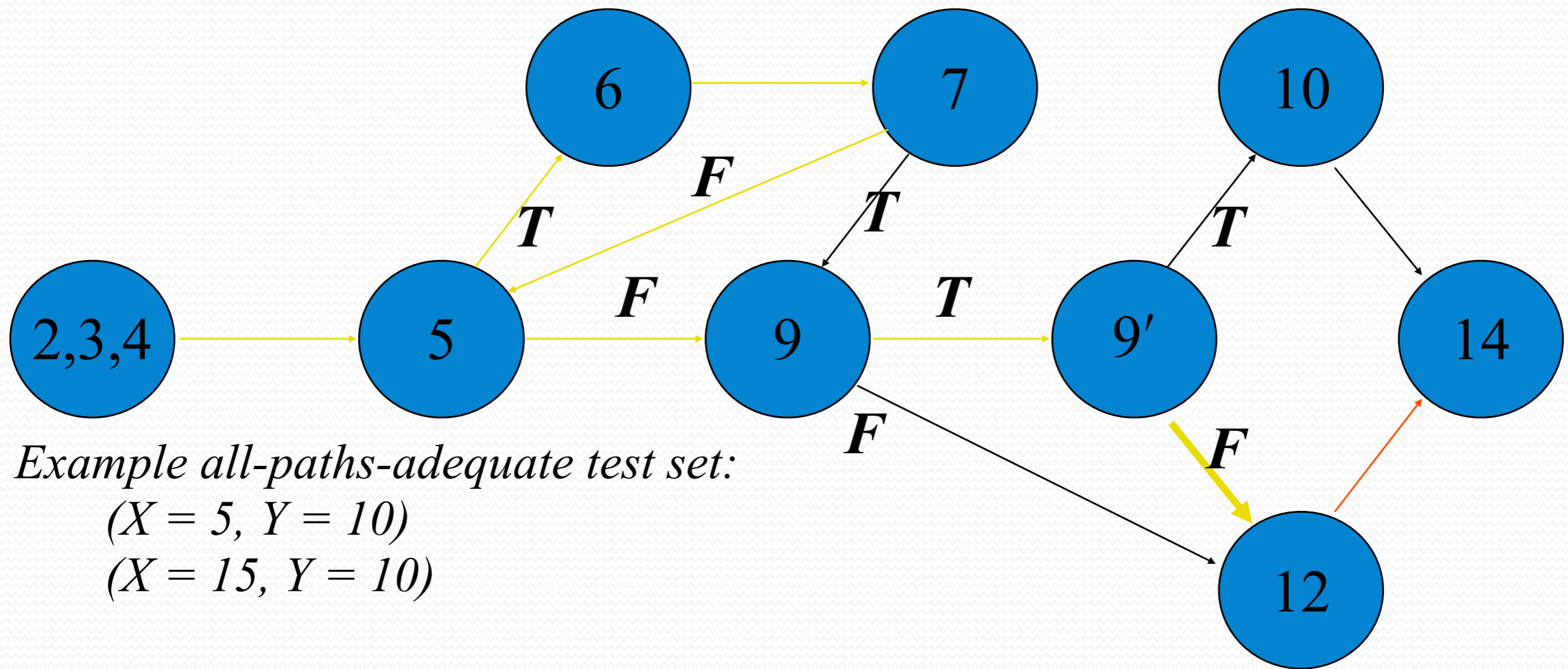


# All-Paths Coverage of P

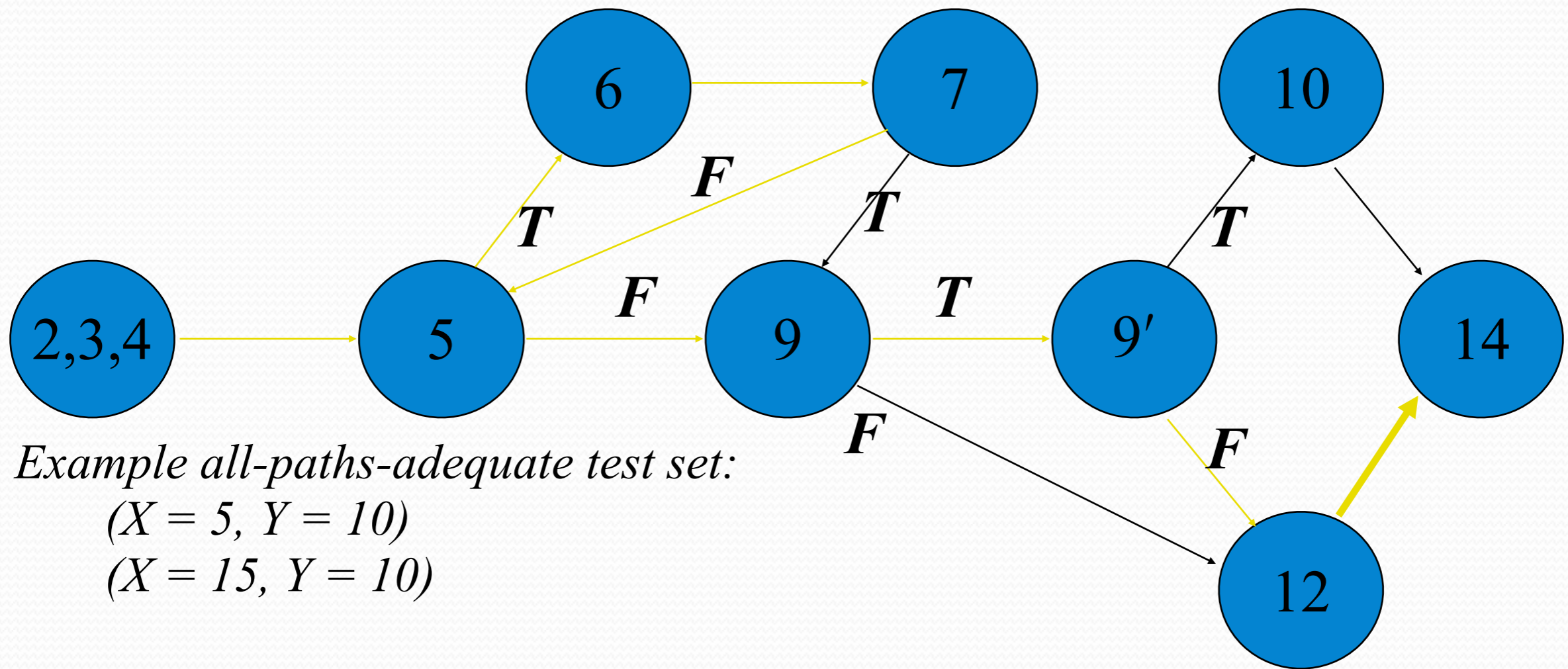




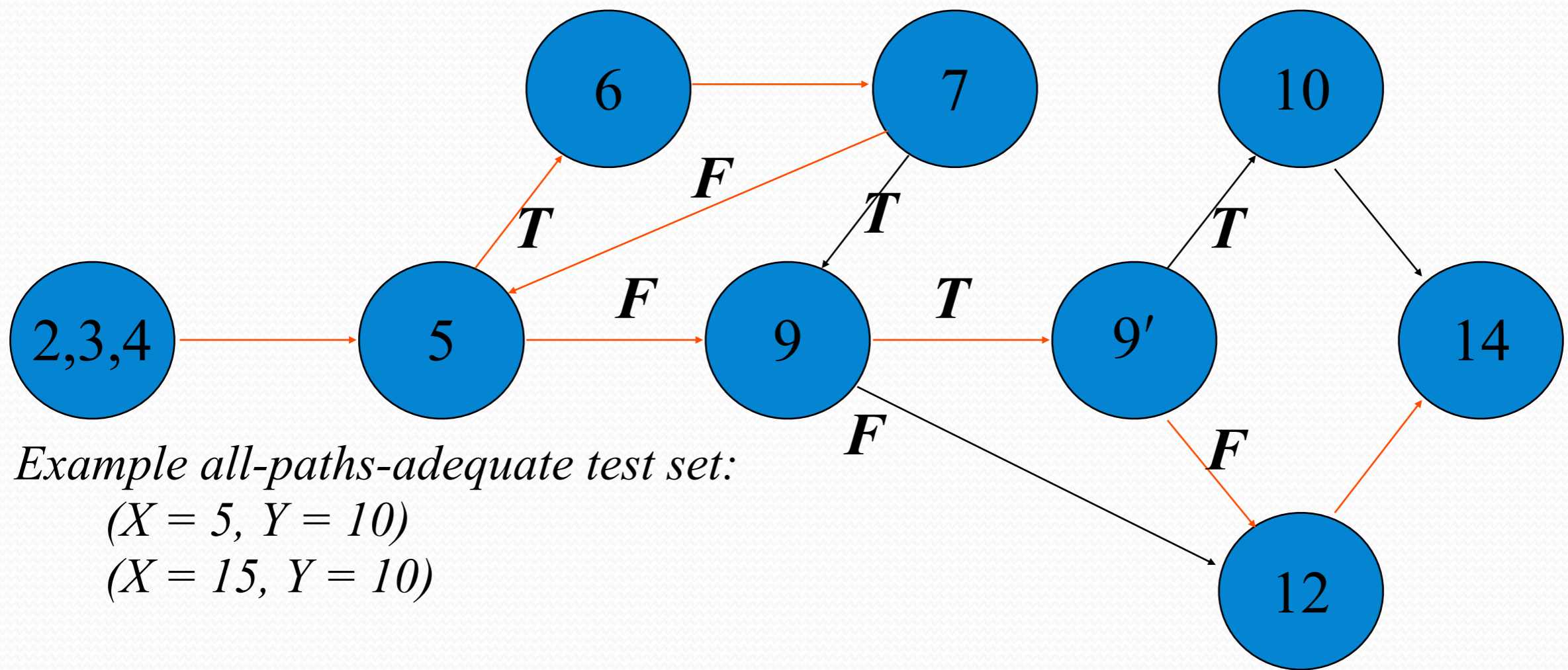
# All-Paths Coverage of P



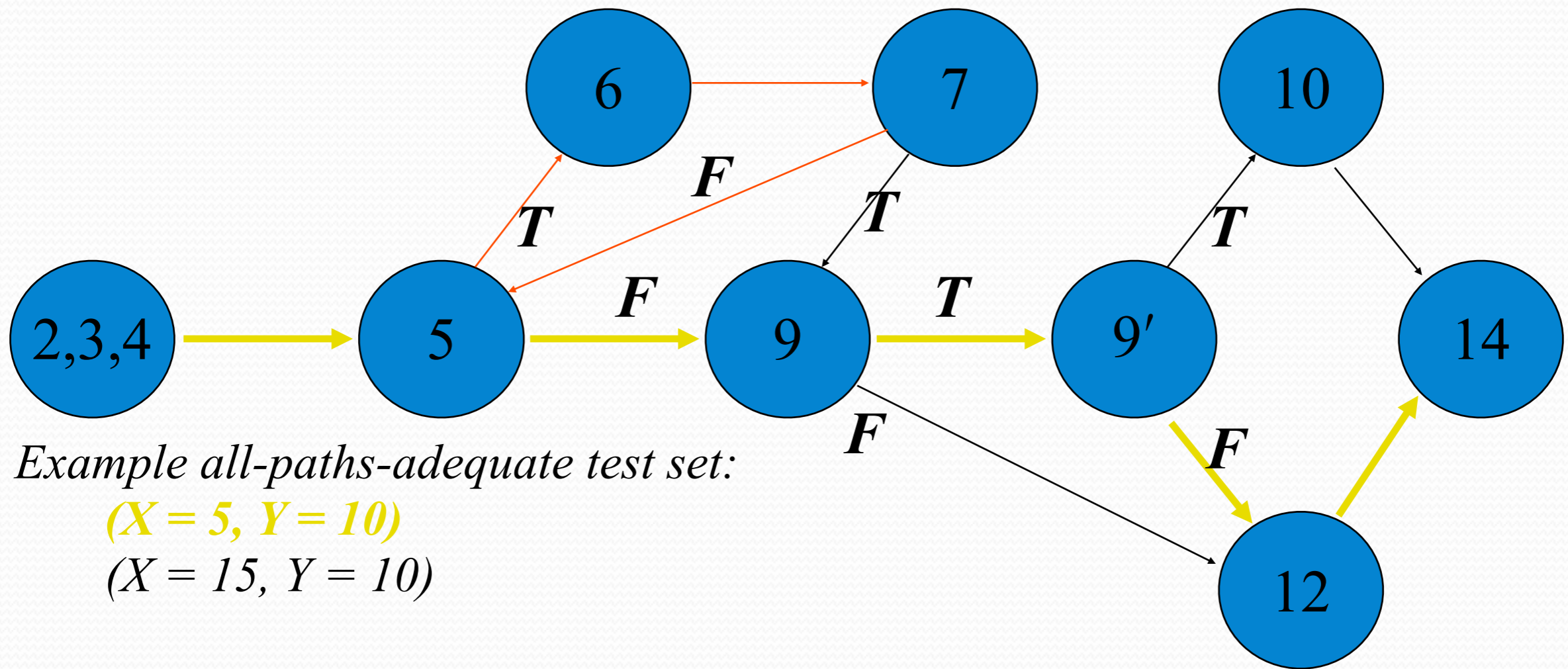
# All-Paths Coverage of P



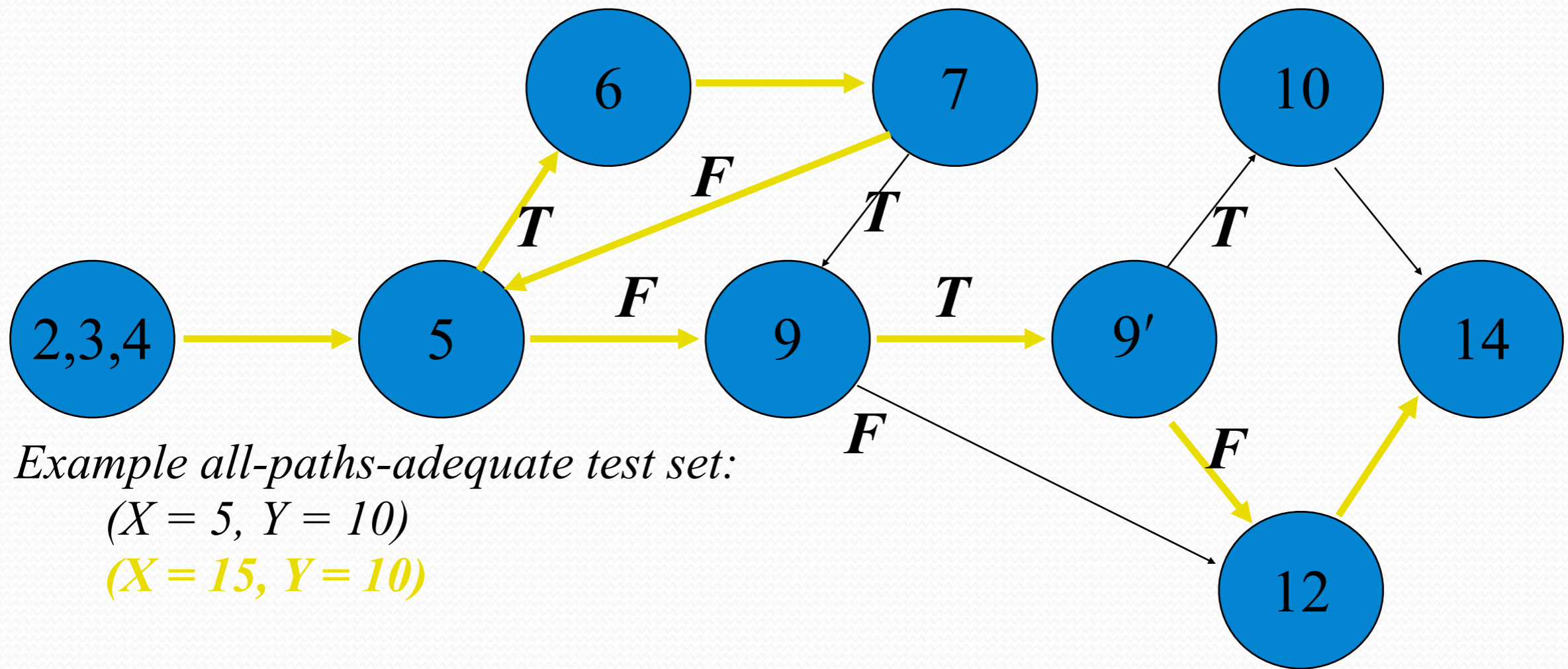
# All-Paths Coverage of P



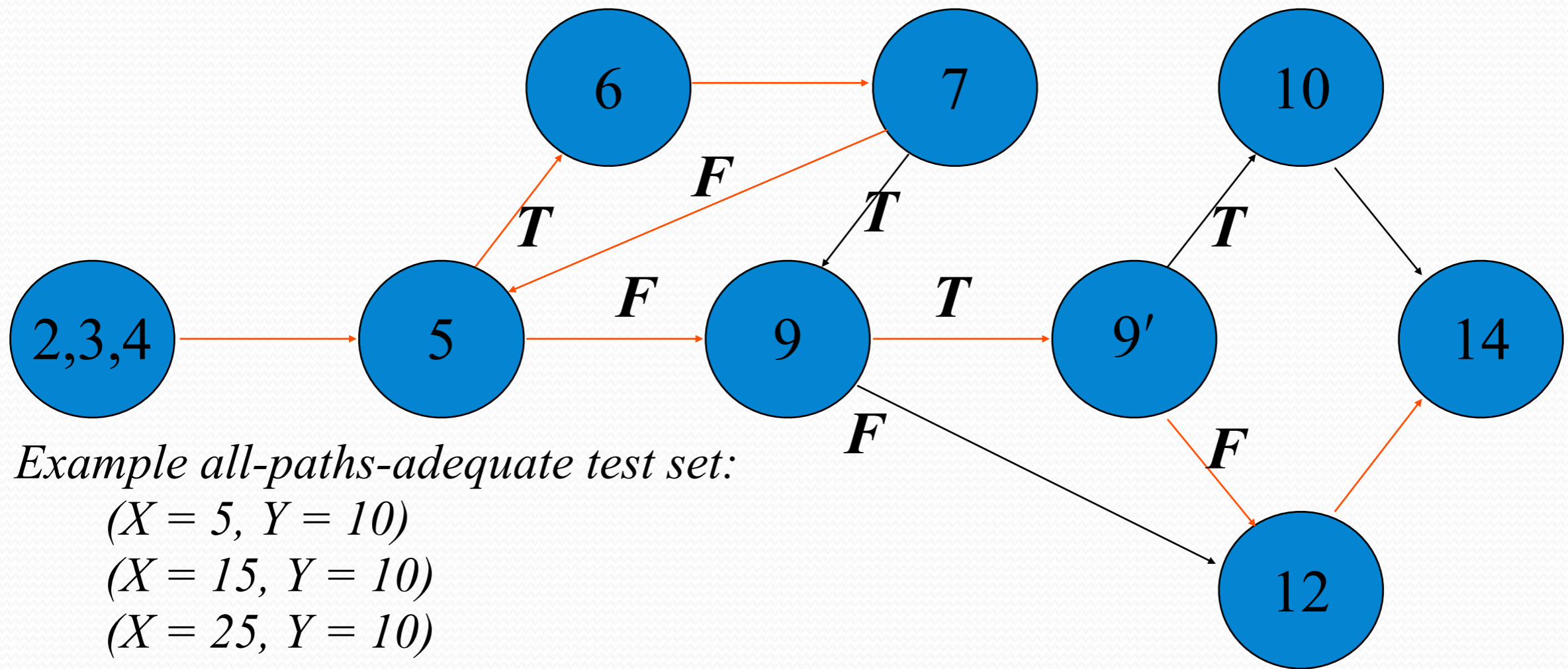
# All-Paths Coverage of P



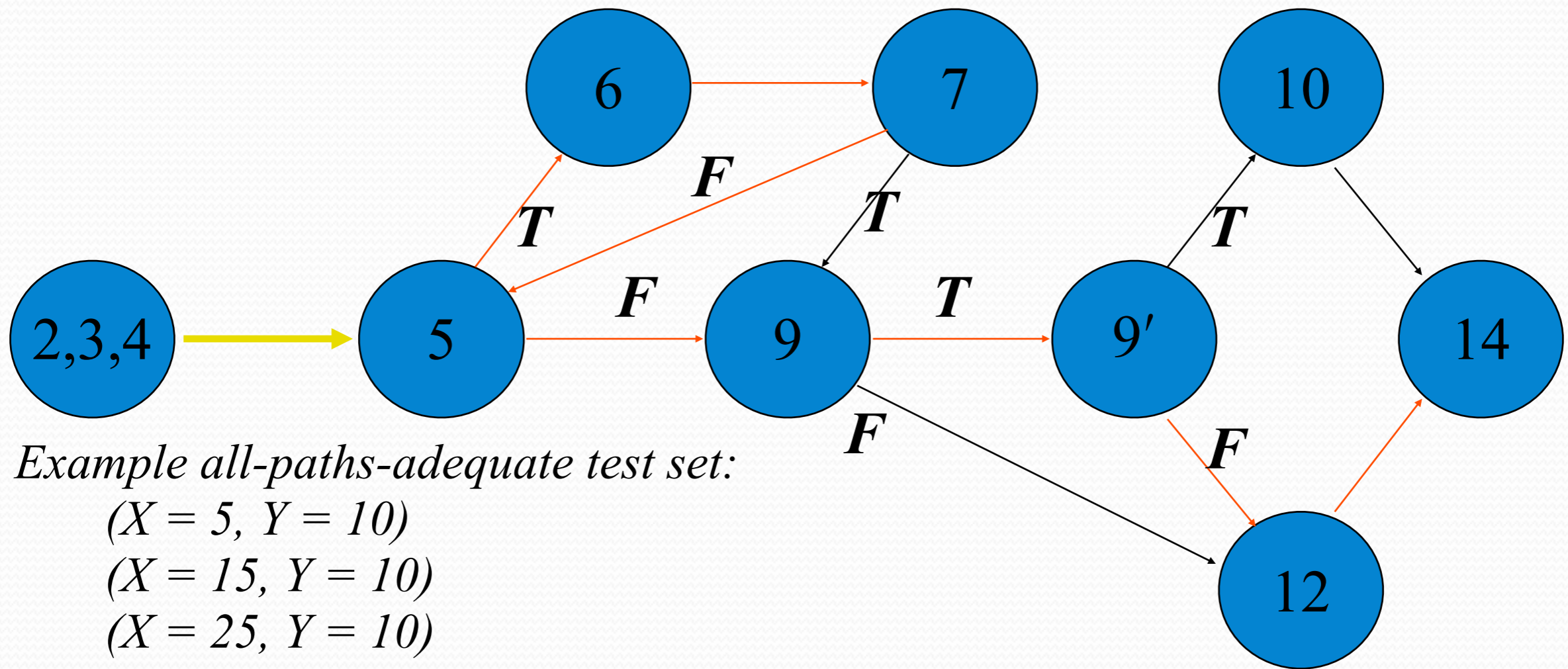
# All-Paths Coverage of P



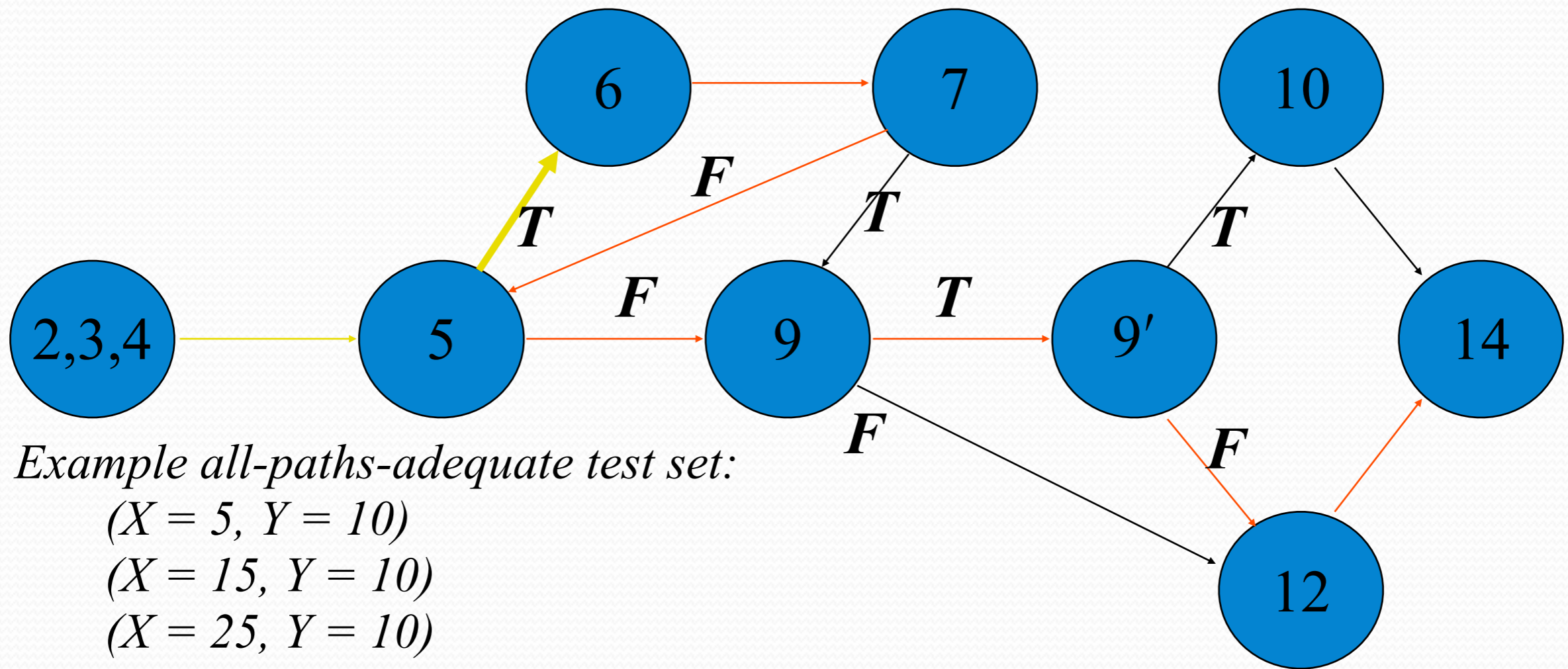
# All-Paths Coverage of P



# All-Paths Coverage of P

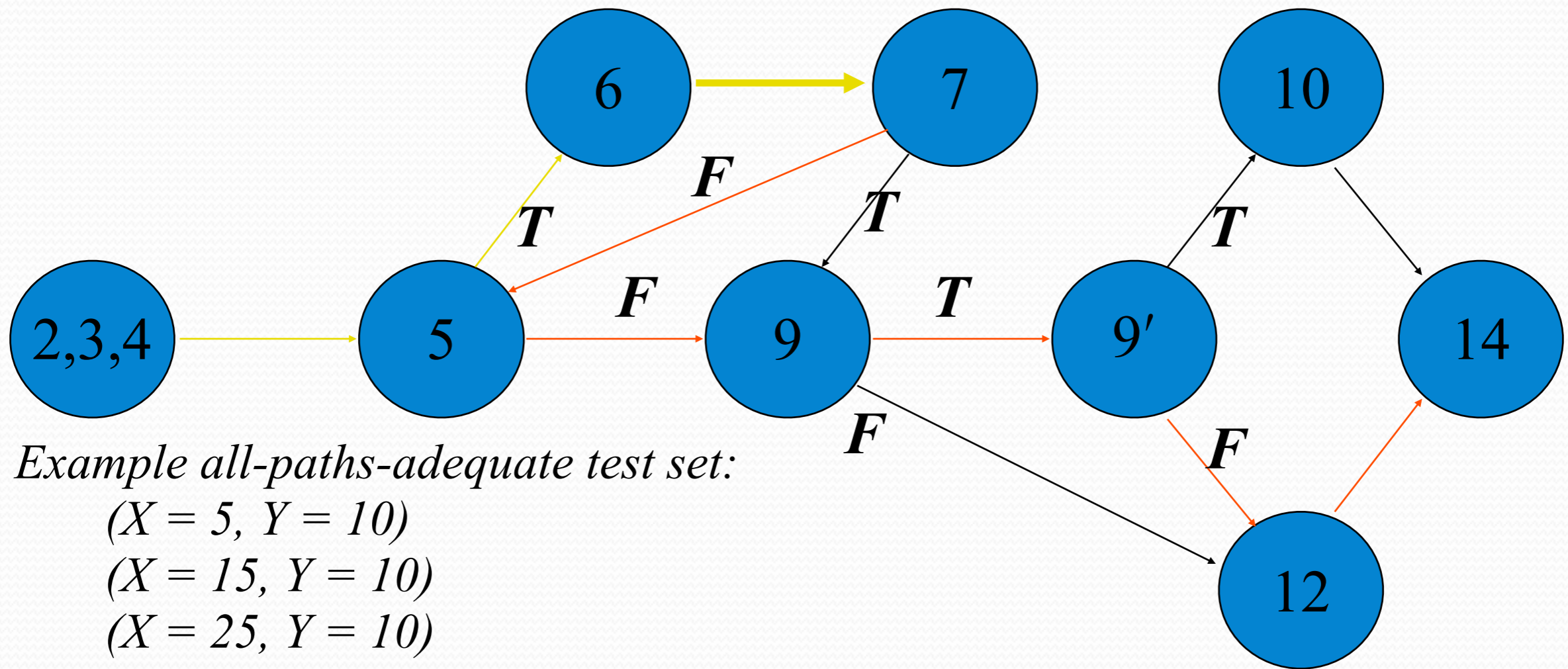


# All-Paths Coverage of P

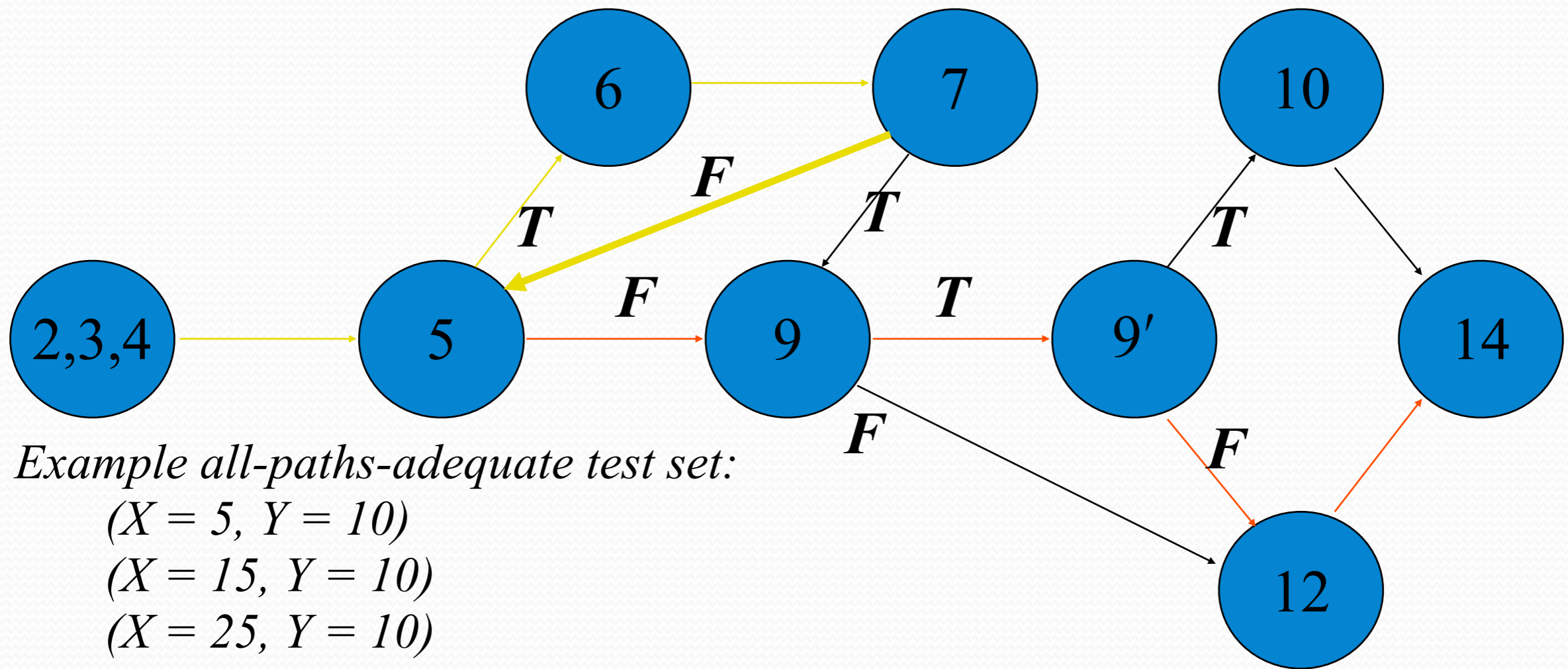




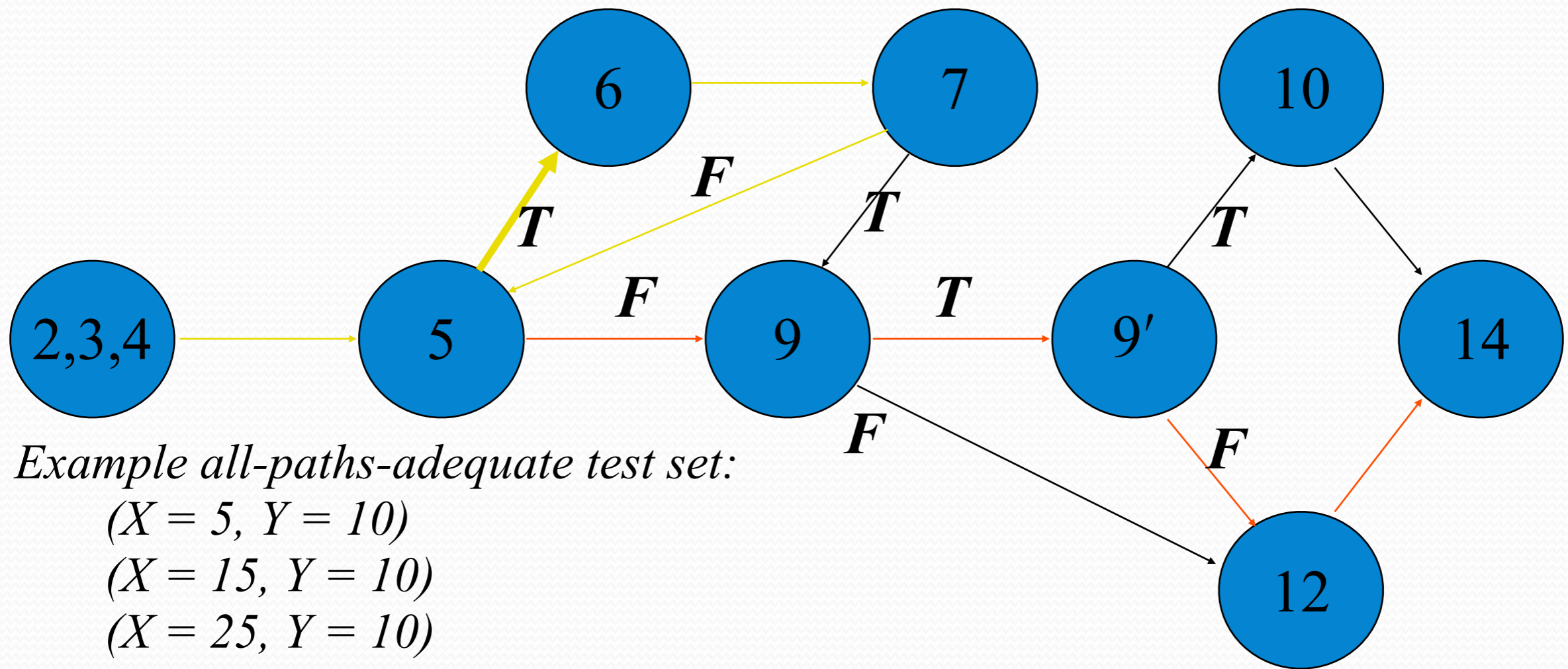
# All-Paths Coverage of P



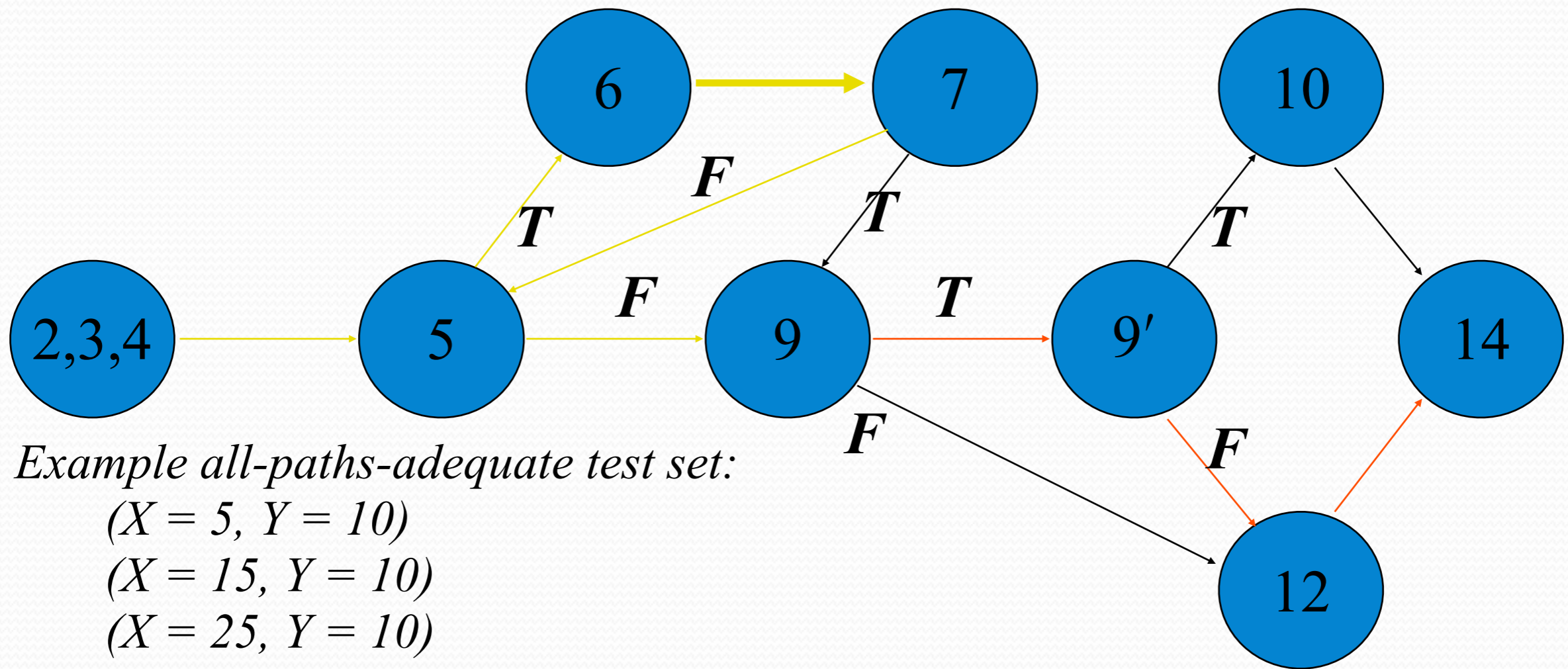
# All-Paths Coverage of P



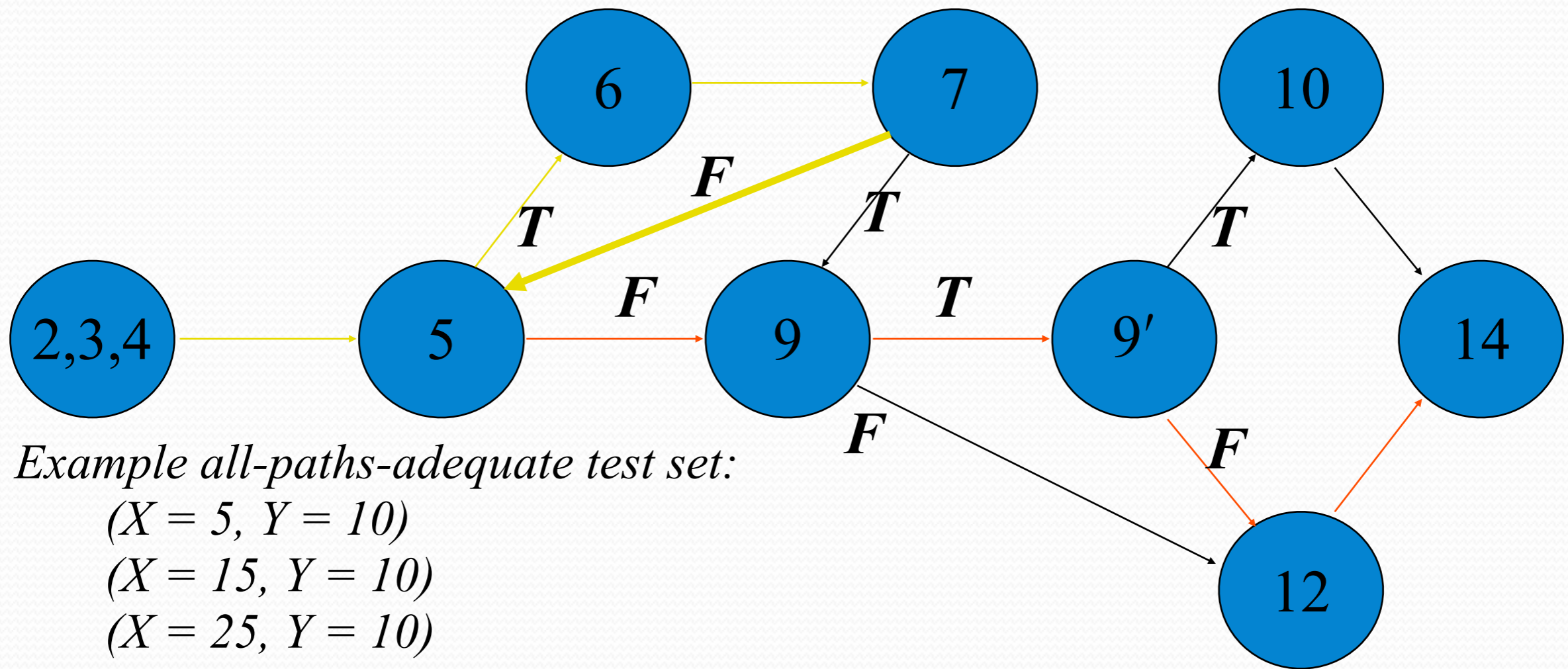
# All-Paths Coverage of P



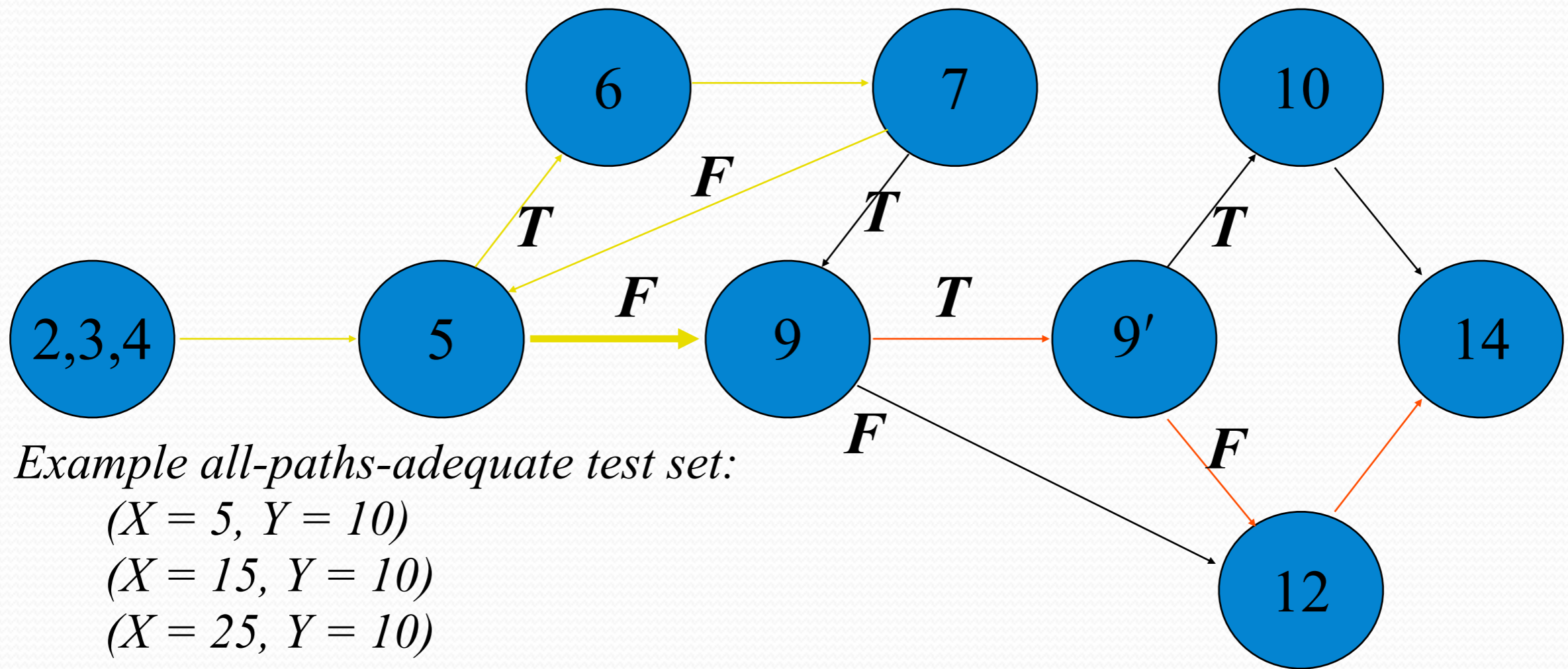
# All-Paths Coverage of P



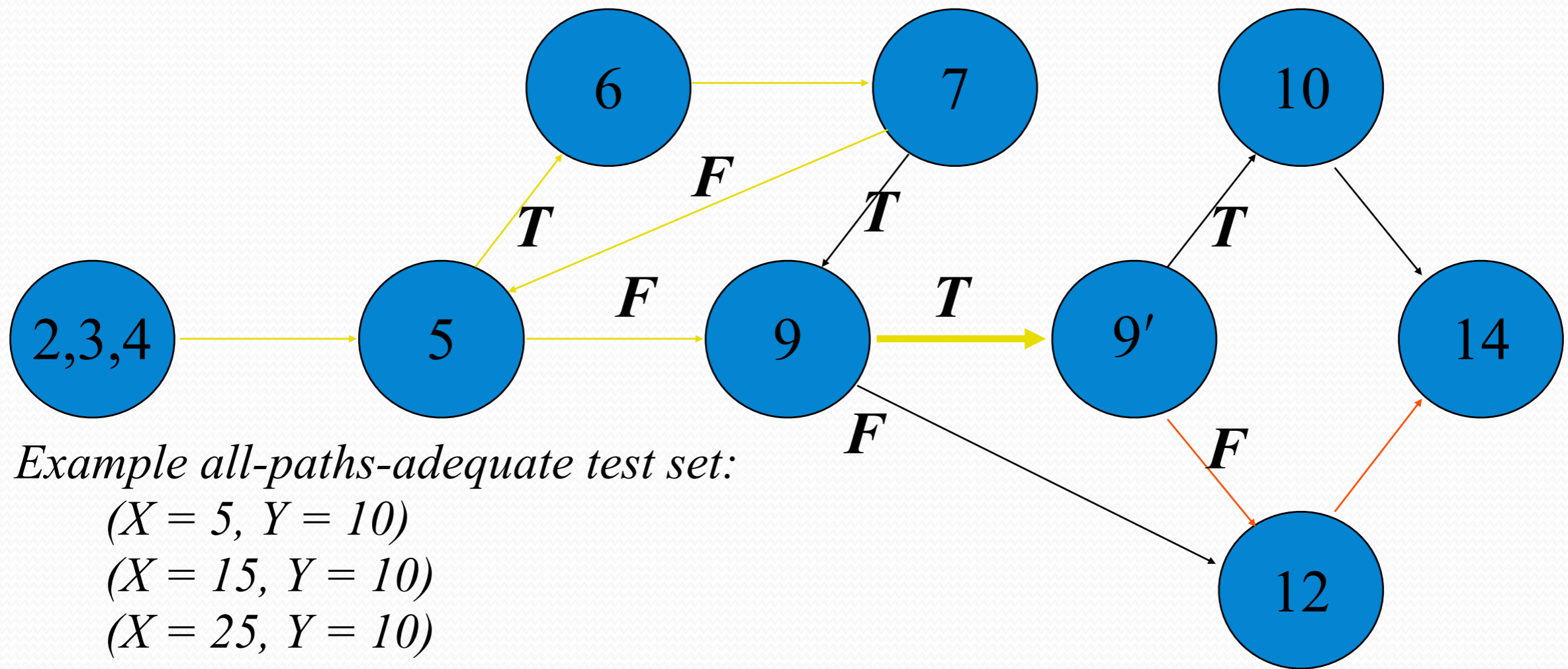
# All-Paths Coverage of P



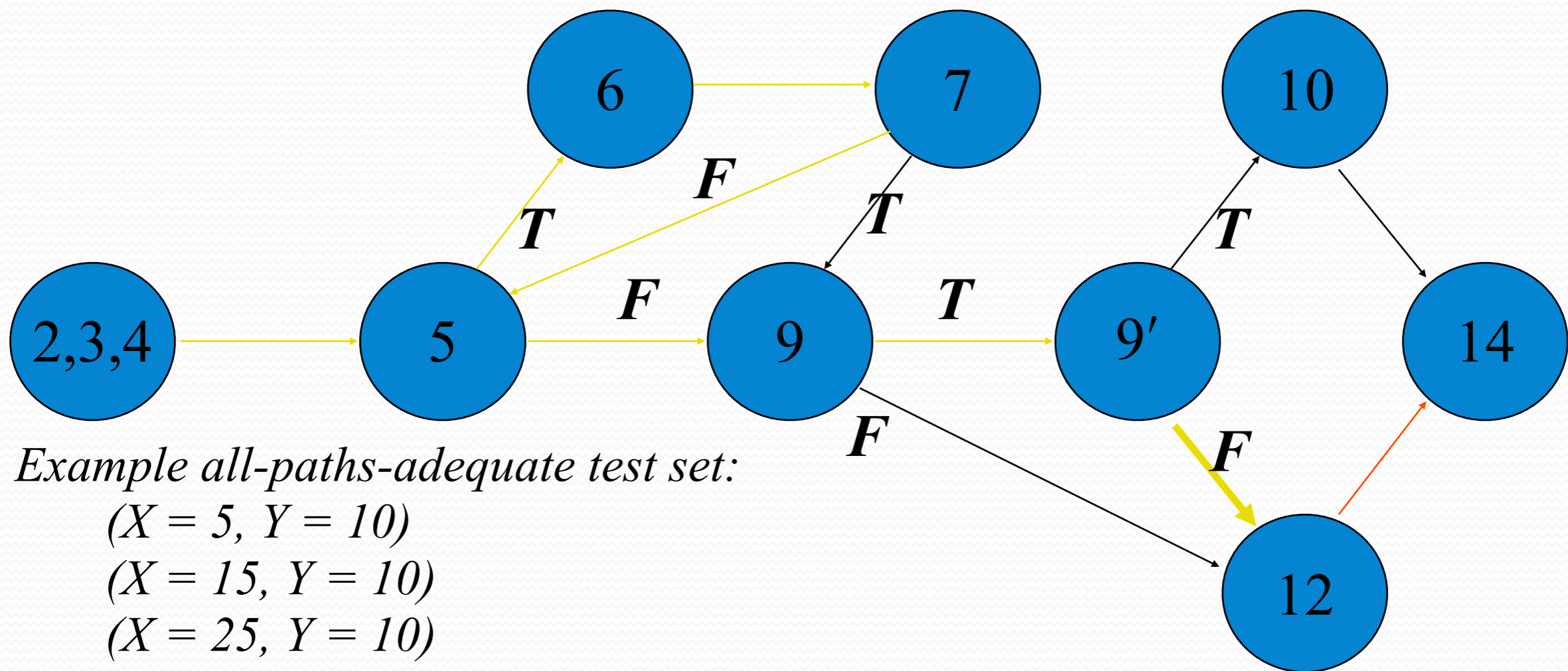
# All-Paths Coverage of P



# All-Paths Coverage of P

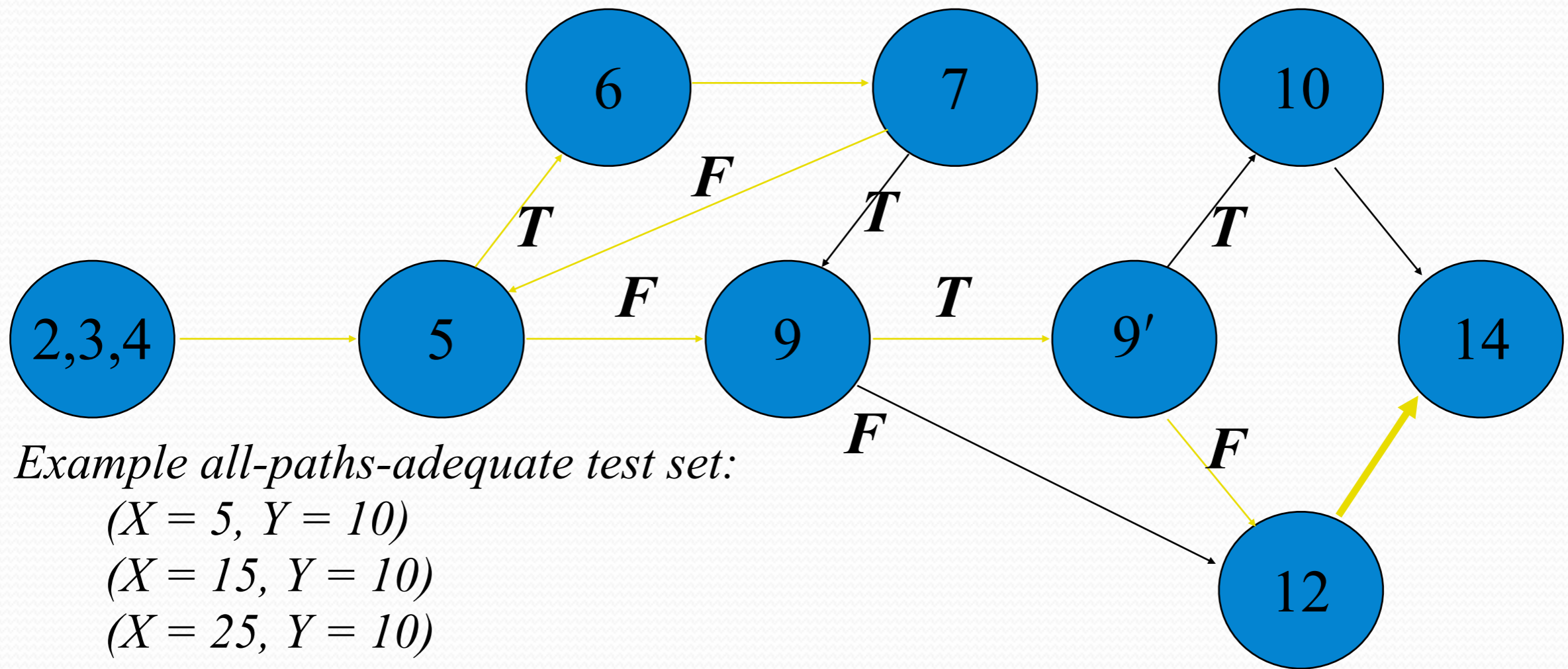


# All-Paths Coverage of P

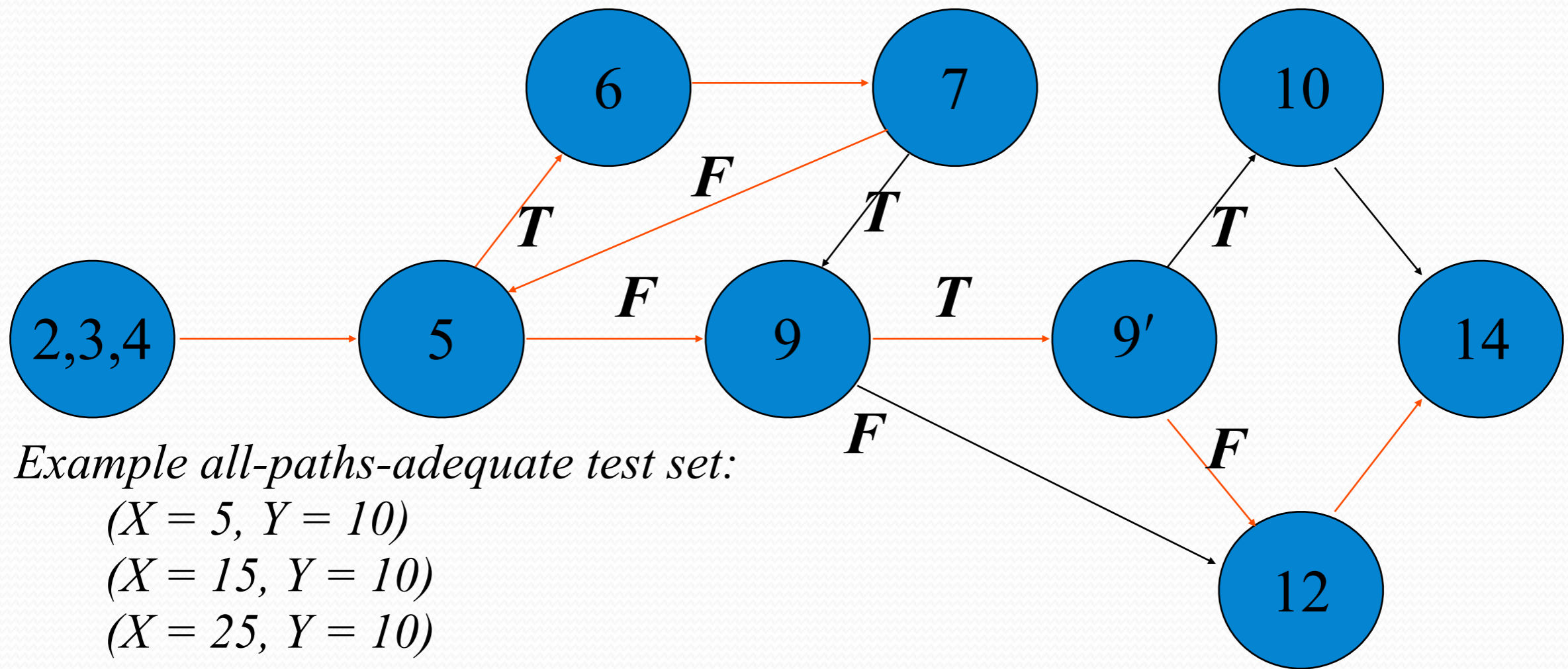




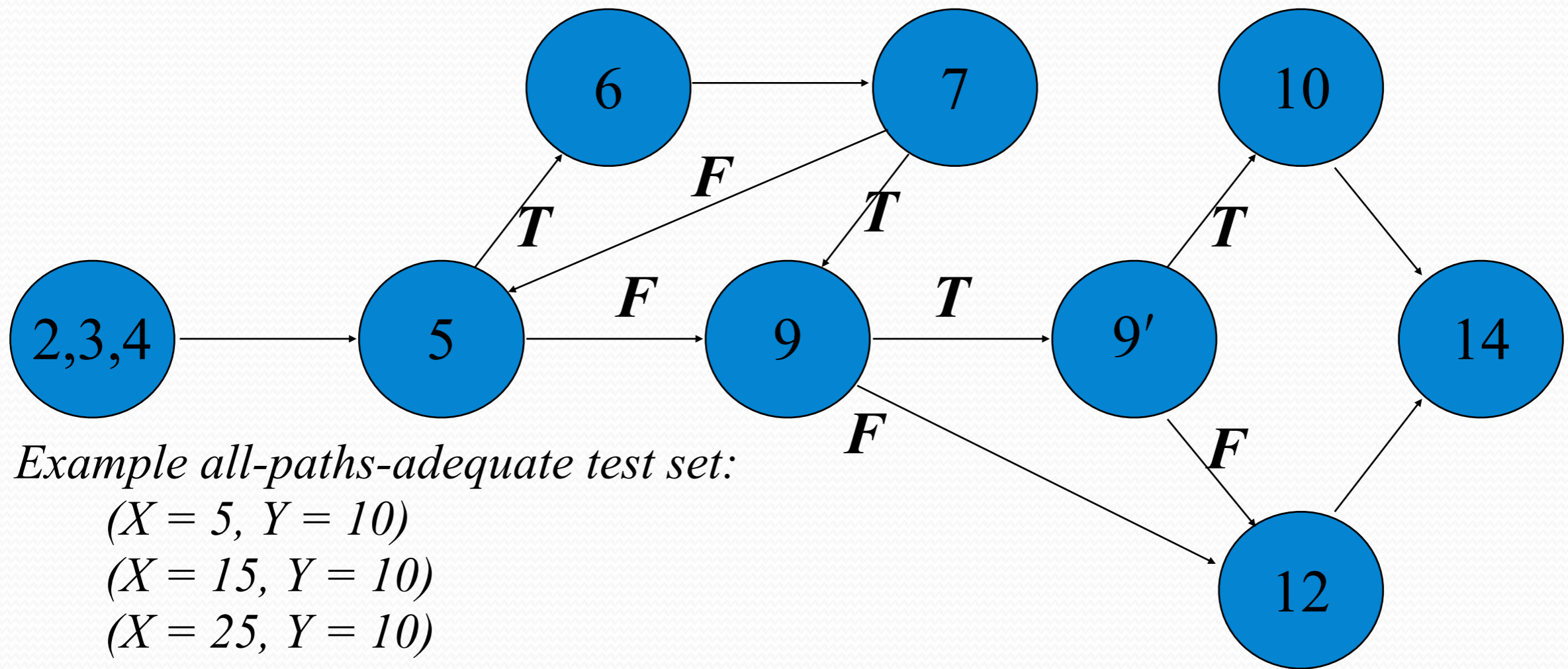
# All-Paths Coverage of P



# All-Paths Coverage of P



# All-Paths Coverage of P



*Example all-paths-adequate test set:*

$(X = 5, Y = 10)$

$(X = 15, Y = 10)$

$(X = 25, Y = 10)$

$(X = 35, Y = 10)$

...

# Code Coverage Tools

76

- ▶ Doing this by hand would be hard!
  - ▶ Fortunately, there are tools that can track code coverage metrics for you
    - ▶ typically just statement and branch coverage
  - ▶ The book covers one tool that is part of a larger system called Cruise Control
    - ▶ These systems typically generate reports that show the percentage of the metric being achieved
      - ▶ they will also typically provide a view of the source code annotated to show which statements and conditions were “hit” by your test suite

# Testing Automation (I)

77

- ▶ It is important that your tests be automated
  - ▶ More likely to be run
  - ▶ More likely to catch problems as changes are made
- ▶ As the number of tests grow, it can take a long time to run the tests, so it is important that the running time of each individual test is as small as possible
  - ▶ If that's not possible to achieve then segregate long running tests from short running tests
    - ▶ execute the latter multiple times per day, execute the former at least once per day (they still need to be run!!)

# Testing Automation (II)

78

- ▶ It is important that running tests be easy
  - ▶ testing frameworks allow tests to be run with a single command
    - ▶ often as part of the build management process (as shown in last lecture)
  - ▶ The book presents details on JUnit (but there are lots of testing frameworks out there)

# Continuous Integration

79

- ▶ Since test automation is so critical, systems known as continuous integration frameworks have emerged
  - ▶ The book covers one called CruiseControl
    - ▶ <http://cruisecontrol.sourceforge.net/>
- ▶ Continuous Integration (CI) systems wrap version control, compilation, and testing into a single repeatable process
  - ▶ You create/debug code as usual;
    - ▶ You then check your code and the CI system builds your code, tests it, and reports back to you

# Wrapping Up

80

- ▶ Testing is one element of software quality assurance
  - ▶ Verification and Validation can occur in any phase
- ▶ Testing of Code involves
  - ▶ Black Box, Grey Box, and White Box tests
  - ▶ All require: input, expected output (via spec), actual output
  - ▶ White box additionally looks for code coverage
- ▶ Testing of systems involves
  - ▶ unit tests, integration tests, system tests and acceptance tests
- ▶ Testing should be automated and various tools exist to integrate testing into the version control and build management processes of a development organization



# Coming Up

81

- ▶ Lecture 21: Deadlock
  - ▶ Read Chapter 6 of the Concurrency textbook
- ▶ Lecture 22: Test-Driven Design / Development
  - ▶ Read Chapter 7 of Head First Software Development