Wong Li Qi B200263C BoSE20-C

Question 1 – 15 marks

A production system maintains a set of rules about the characteristics of fruits as follows:

Rule 1:	IF	Shape = long AND Colour = green OR yellow
	THEN	Fruit = banana
Rule 2:	IF	Shape = round OR oblong AND Diameter > 4 inches
	THEN	Fruitclass = vine
Rule 3:	IF	Shape = round AND Diameter < 4 inches
	THEN	Fruitclass = tree
Rule 4:	IF	Seedcount = 1
	THEN	Seedclass = stonefruit
Rule 5:	IF	Seedcount > 1
	THEN	Seedclass = multiple
Rule 6:	IF	Fruitclass = vine AND Colour = green
	THEN	Fruit = watermelon
Rule 7:	IF	Fruitclass = vine AND Surface = smooth AND Colour = yellow
	THEN	Fruit = honeydew
Rule 8:	IF	Fruitclass = vine AND Surface = rough AND Colour = tan
	THEN	Fruit = cantaloupe
Rule 9:	IF	Fruitclass = tree AND Colour = orange AND Seedclass = stonefruit
	THEN	Fruit = apricot
Rule 10:	IF	Fruitclass = tree AND Colour = orange AND Seedclass = multiple
	THEN	Fruit = orange
Rule 11:	IF	Fruitclass = tree AND Colour = red AND Seedclass = stonefruit
	THEN	Fruit = cherry

Rule 12: IF Fruitclass = tree AND
Colour = orange AND
Seedclass = stonefruit
THEN Fruit = peach

Rule 13: IF Fruitclass = tree AND
Colour = red OR yellow OR green AND
Seedclass = multiple
THEN Fruit = apple

Rule 14: IF Fruitclass = tree AND
Colour = purple AND
Seedclass = stonefruit
THEN Fruit = plum

i) Use ***FORWARD CHAINING*** to describe the production system table including its working memory, conflict set and rule fired to establish a fruit. Initial data given is :

Shape = round
Diameter > 4 inches
Surface = smooth
Colour = yellow

Terminate when the final value for Fruit in the working memory. [6 marks]

Iteration #	Working memory	Conflict set	Rule fired
0	Shape = round	2,3	Halt
1	Diameter>4 inches	2	2
2	Fruitclass = vine	6,7,8	Halt
3	Surface = smooth	7	Halt
4	Color = yellow	7	7
5	Fruit = honeydew		Halt

ii) Given the fruit to search is apple, use ***BACKWARD CHAINING*** to describe the production system table including its working memory, conflict set and rule fired toestablish the initial data for this fruit.

State the initial facts required to establish that the fruit searched is an apple. [9 marks]

Iteration #	Working memory	Conflict set	Rule fired
0	Fruit = apple		Halt
	Seedclass = multiple	13	13
	Colour = red OR yellow OR green	11,13	Halt
	Fruitclass = tree	9,10,11,12,13,14	Halt
	Seedcount > 1	5	5
	Diameter < 4 inches	3	3
	Shape = round	2,3	Halt

The initial facts required to establish fruit to search is apple are:

Shape = round
Diameter < 4 inches
Seedcount > 1
Colour = Red or Yellow or Green

Introduction to Software Testing

CSCI 5828: Foundations of Software Engineering
Lecture 05 — 01/31/2012

© Kenneth M. Anderson, 2012

Goals

- Provide introduction to fundamental concepts of software testing
    - Terminology
    - Testing of Systems
        - unit tests, integration tests, system tests, acceptance tests
    - Testing of Code
        - Black Box
        - Gray Box
        - White Box
        - Code Coverage

© Kenneth M. Anderson, 2012

Testing

- Testing is a **critical element** of software development life cycles
  - called **software quality control** or **software quality assurance**
  - basic goals: **validation** and **verification**
    - validation: **are we building the right product?**
      - verification: **does "X" meet its specification?**
        - where "X" 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

© Kenneth M. Anderson, 2012

Terminology

- An **error** is a mistake made by an engineer
  - often a misunderstanding of a requirement or design specification
- A **fault** is a manifestation of that error in the code
  - what we often call “a bug”
- A **failure** is an incorrect output/behavior that is caused by executing a fault
  - The failure may occur immediately (crash!) or much, much later in the execution
- **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?

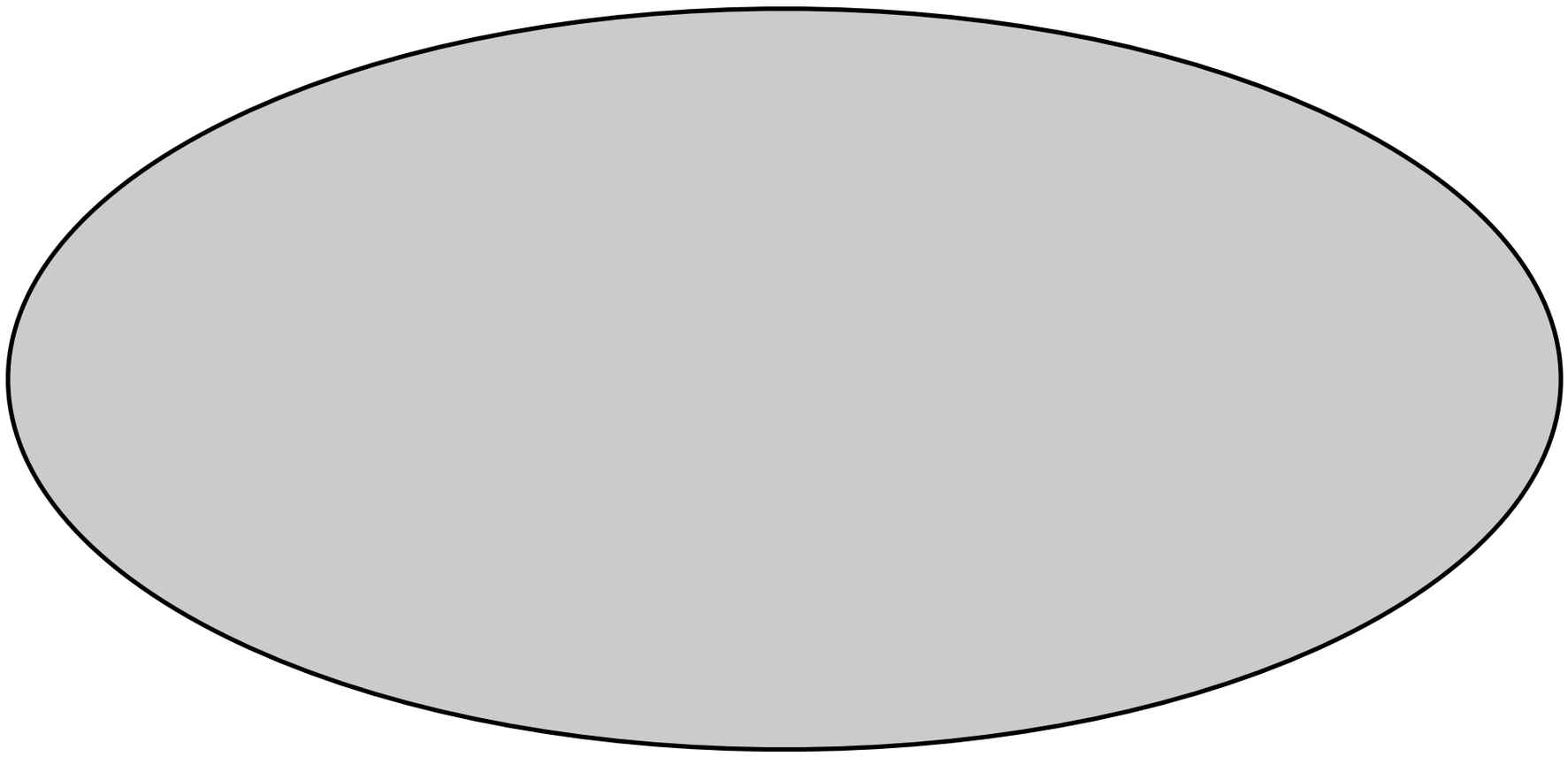
© Kenneth M. Anderson, 2012

No!

- 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 resulting in no visible failures!
      - this is particularly insidious
- However, if we do a good job in creating a test set that
  - covers all functional capabilities of a system
  - and 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

© Kenneth M. Anderson, 2012

Looking for Faults

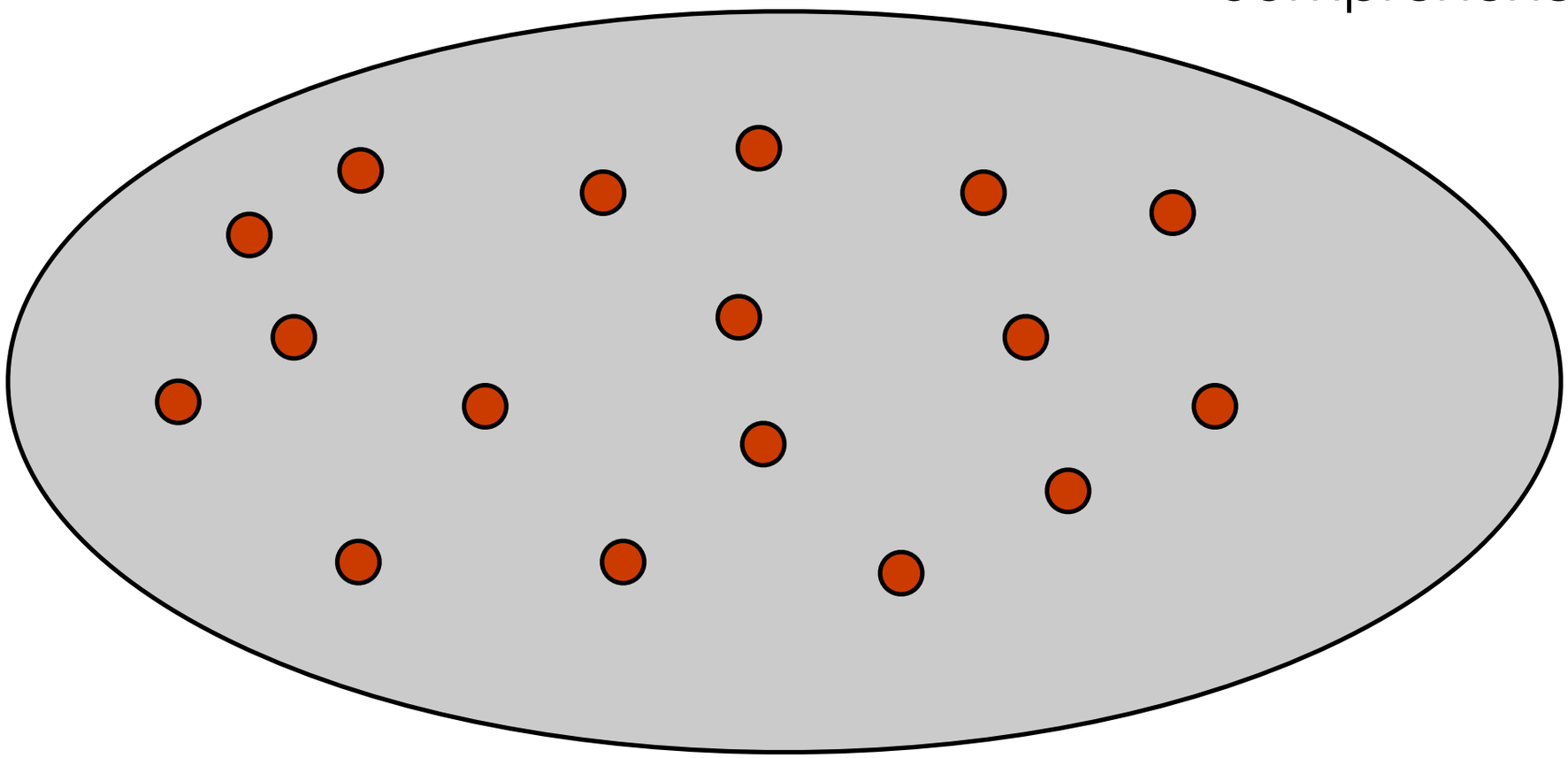


All possible states/behaviors of a system

© Kenneth M. Anderson, 2012

Looking for Faults

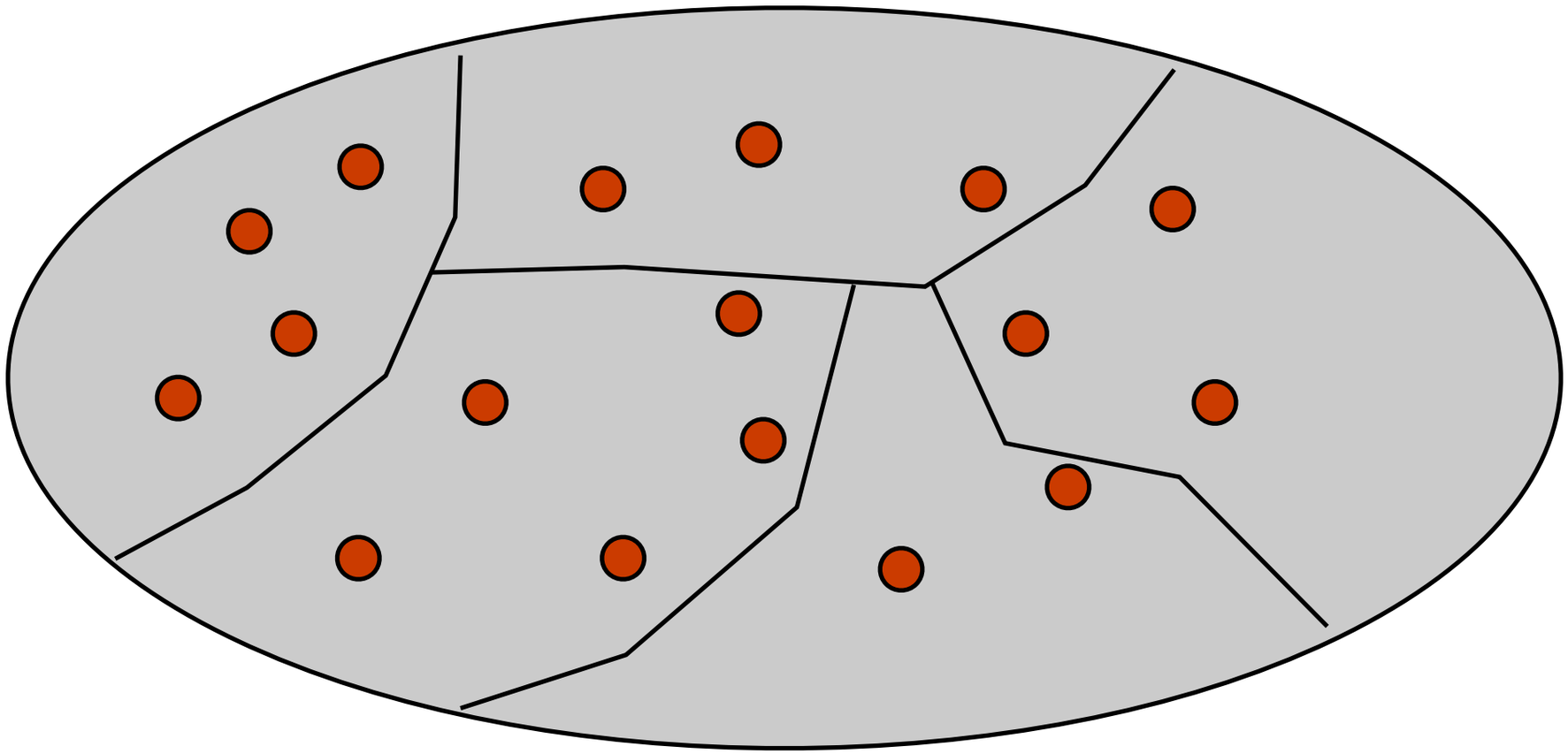
As you can see, its not very comprehensive



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

© Kenneth M. Anderson, 2012

One way forward? Fold



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

© Kenneth M. Anderson, 2012

What does that mean?

- 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

© Kenneth M. Anderson, 2012

Completeness

- 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?

© Kenneth M. Anderson, 2012

Continuous Testing

- Testing is a continuous process that should be performed at every stage of a software development process
    - During requirements gathering, for instance, we must continually query the user, “Did we get this right?”
    - Facilitated by an emphasis on iteration throughout a life cycle
- at the end of each iteration
    - we check our results to see if what we built is meeting our requirements (specification)

© Kenneth M. Anderson, 2012

Testing the System (I)

- **Unit Tests**
  - Tests that cover low-level aspects of a system
    - For each module, does each operation perform as expected
    - For method **foo()**, we'd like to see another method **testFoo()**
- **Integration Tests**
  - Tests that check that modules work together in combination
  - Most projects on schedule until they hit this point (MMM, Brooks)
    - 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 (Mars Polar Lander)

© Kenneth M. Anderson, 2012

Testing the System (II)

- **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

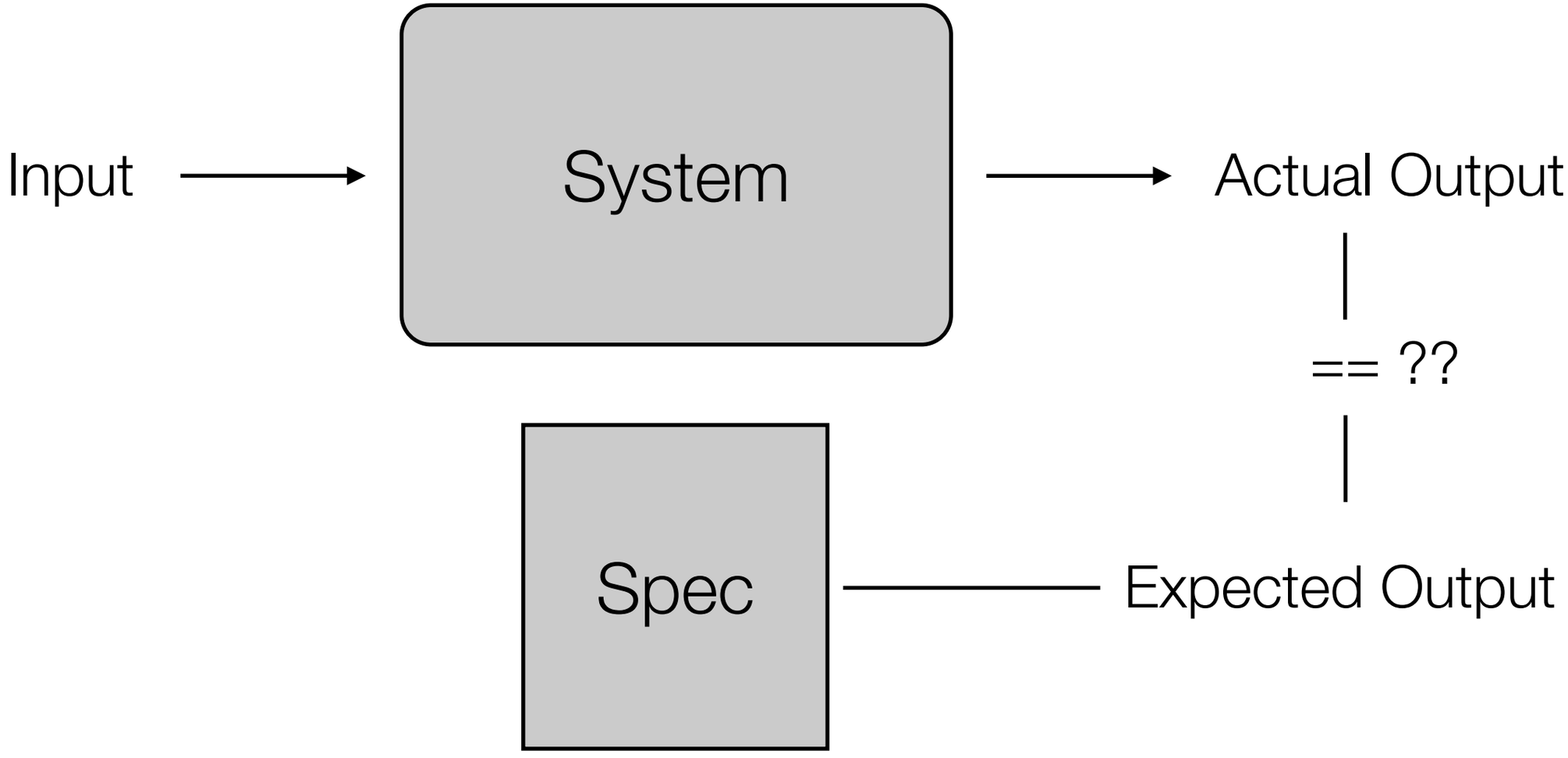
© Kenneth M. Anderson, 2012

Multi-Level Testing

- 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, …

© Kenneth M. Anderson, 2012

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!

© Kenneth M. Anderson, 2012

Results

- 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

© Kenneth M. Anderson, 2012

Black Box Categories

- 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

© Kenneth M. Anderson, 2012

Grey Box Testing

- 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

© Kenneth M. Anderson, 2012

White Box Testing

- 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?

© Kenneth M. Anderson, 2012

Code Coverage (I)

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

© Kenneth M. Anderson, 2012

Code Coverage (II)

- 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)

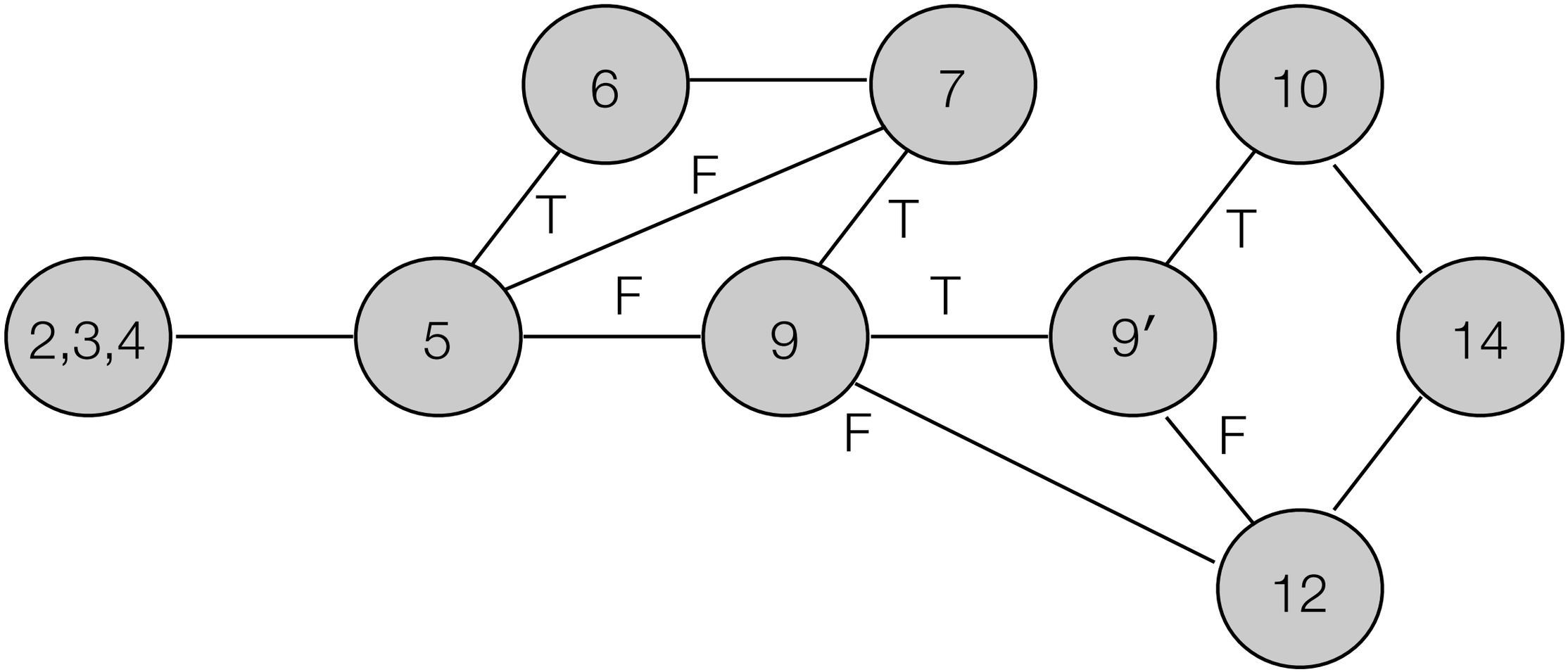
© Kenneth M. Anderson, 2012

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;
```

© Kenneth M. Anderson, 2012

P’s Control Flow Graph (CFG)



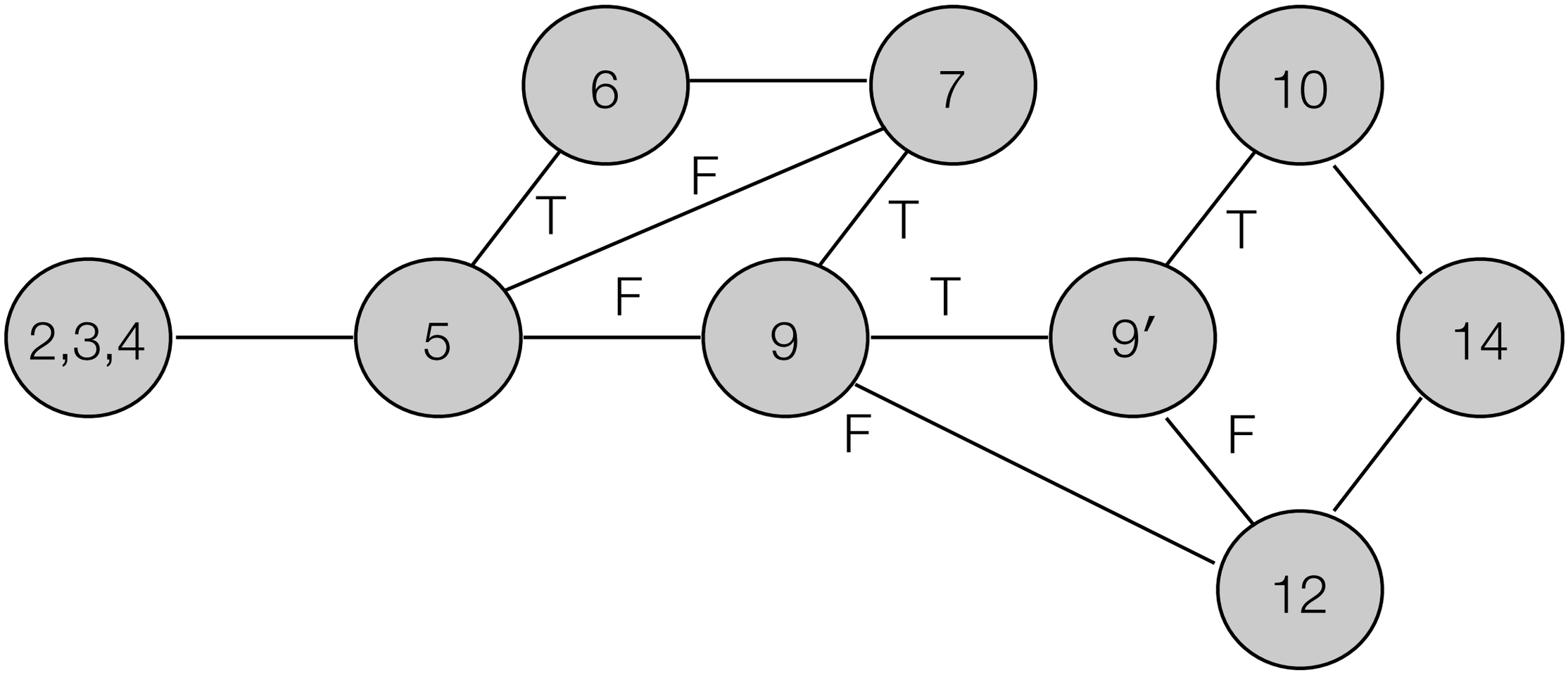
© Kenneth M. Anderson, 2012

White-box Testing Criteria

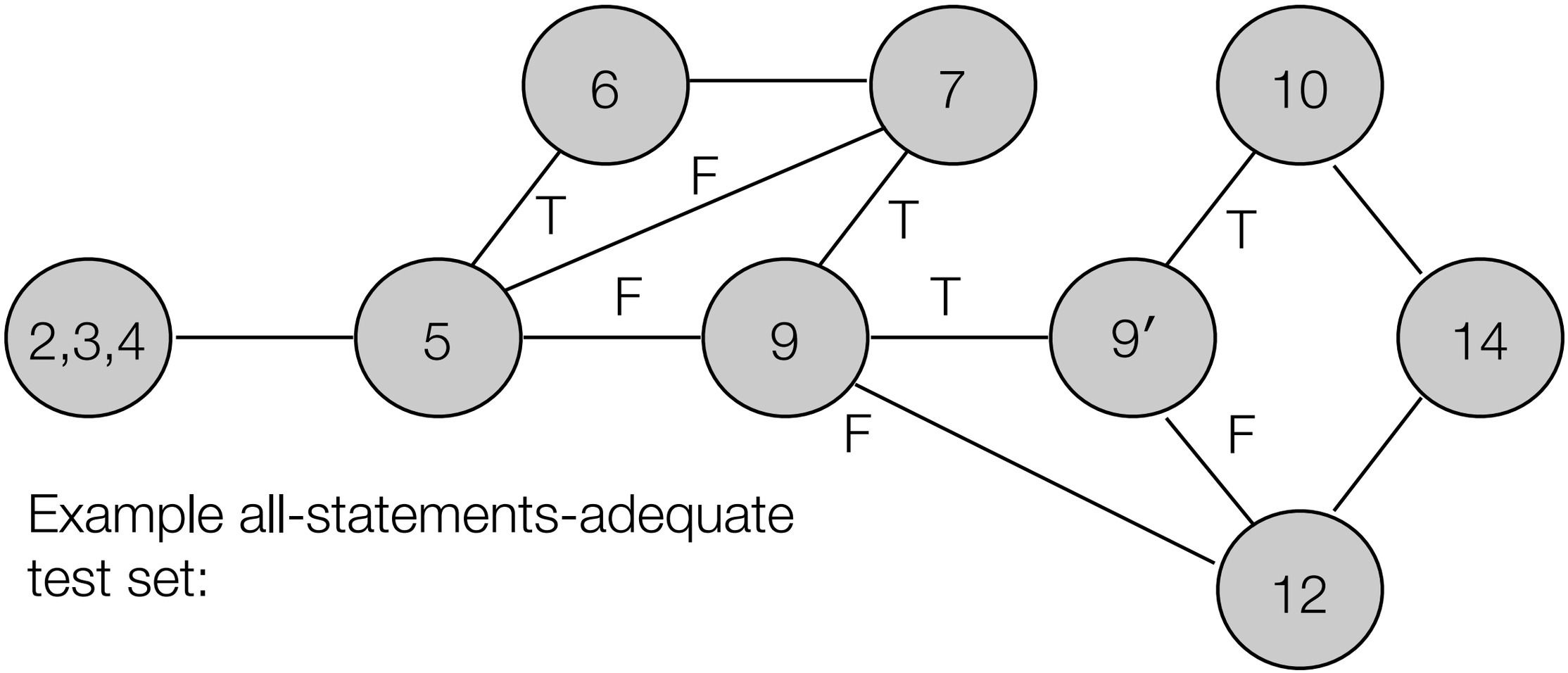
- Statement Coverage
  - Create a test set T such that
    - by executing P for each t in T
      - each elementary statement of P is executed at least once

© Kenneth M. Anderson, 2012

All-Statements Coverage of P

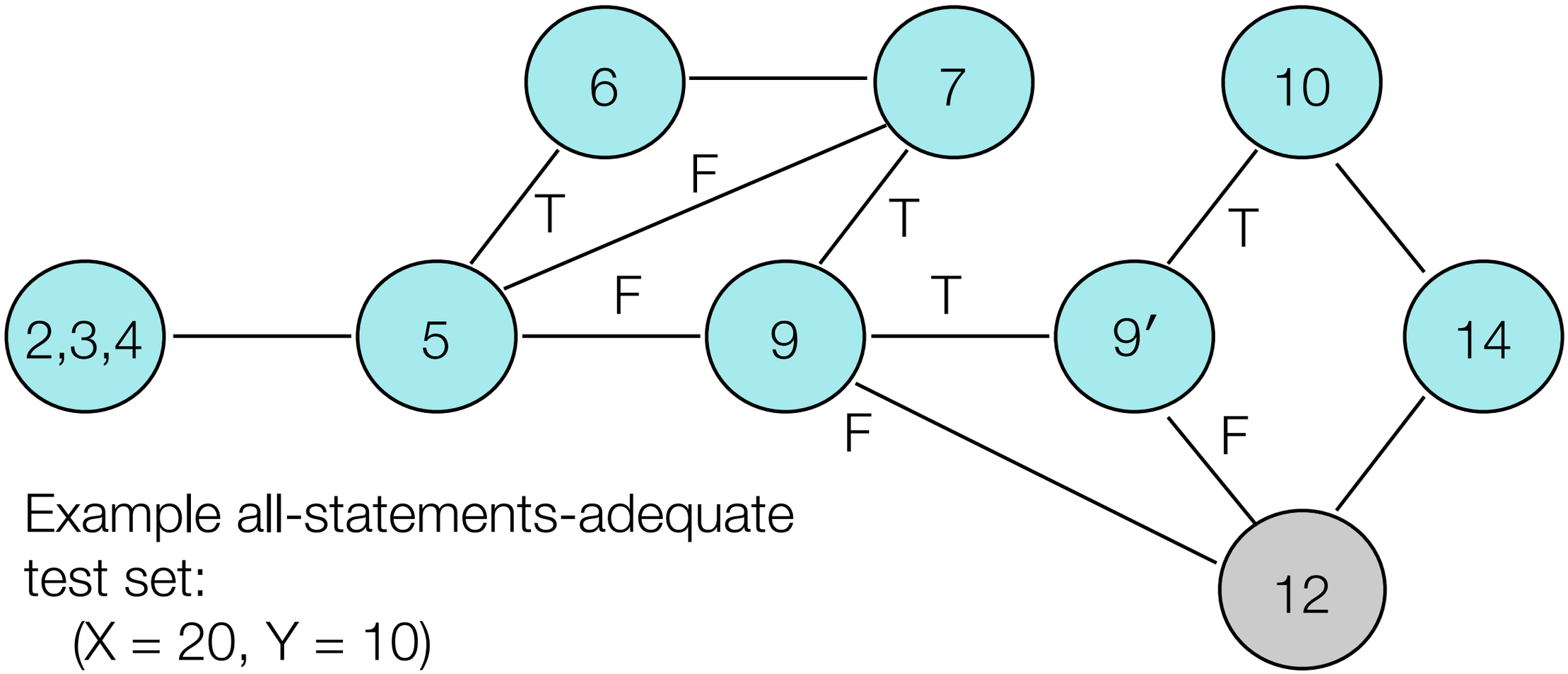


All-Statements Coverage of P



Example all-statements-adequate test set:

All-Statements Coverage of P

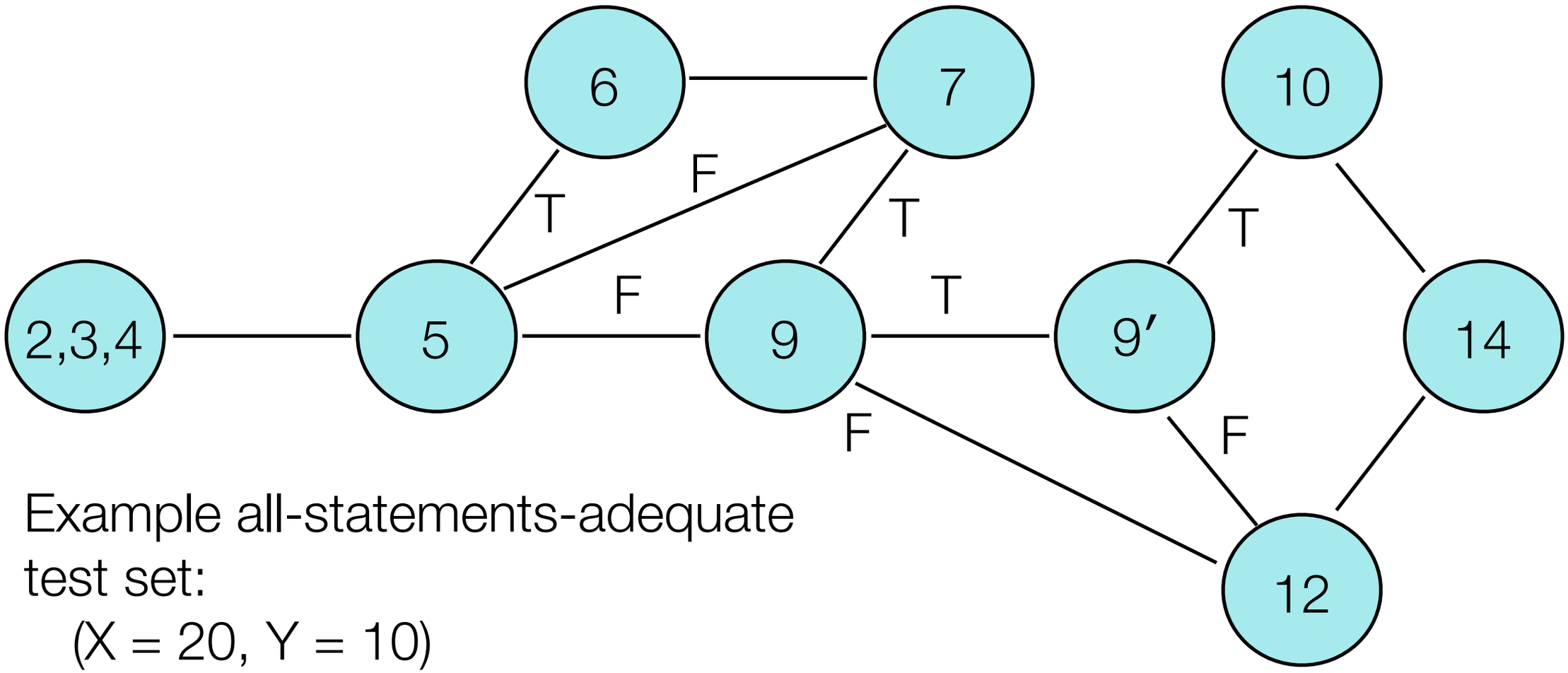


Example all-statements-adequate test set:

(X = 20, Y = 10)

© Kenneth M. Anderson, 2012

All-Statements Coverage of P



Example all-statements-adequate test set:

(X = 20, Y = 10)

(X = 20, Y = 30)

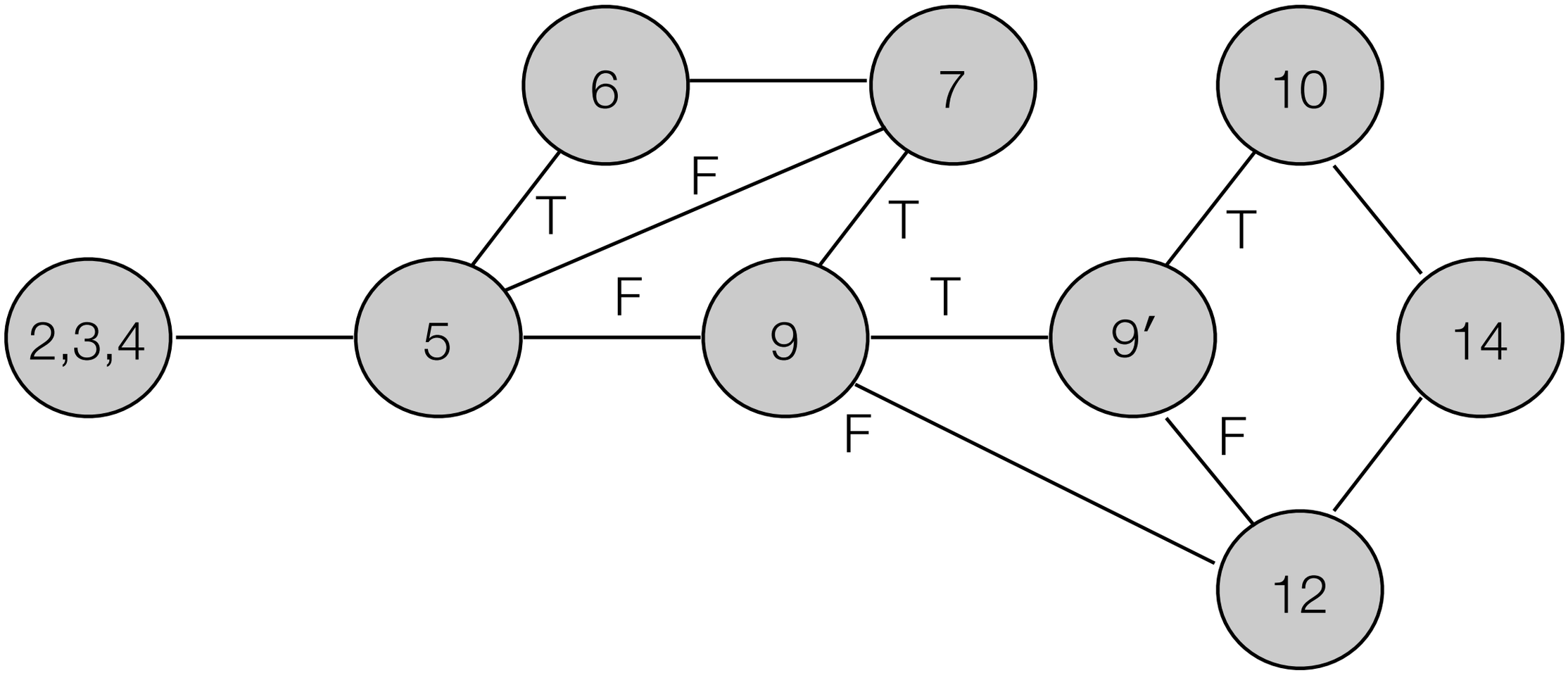
© Kenneth M. Anderson, 2012

White-box Testing Criteria

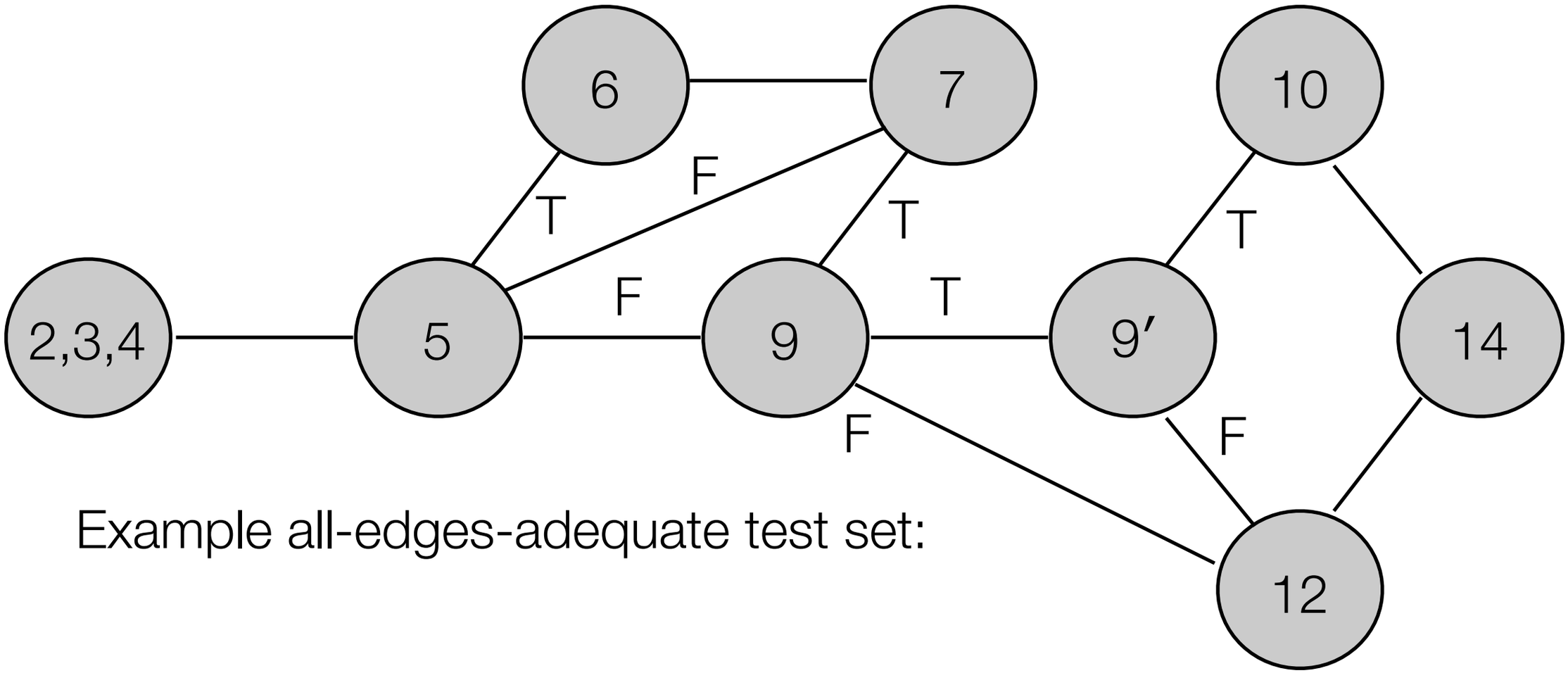
- Edge Coverage
  - Select a test set T such that
    - by executing P for each t in T
      - each edge of P's control flow graph is traversed at least once

© Kenneth M. Anderson, 2012

All-Edges Coverage of P



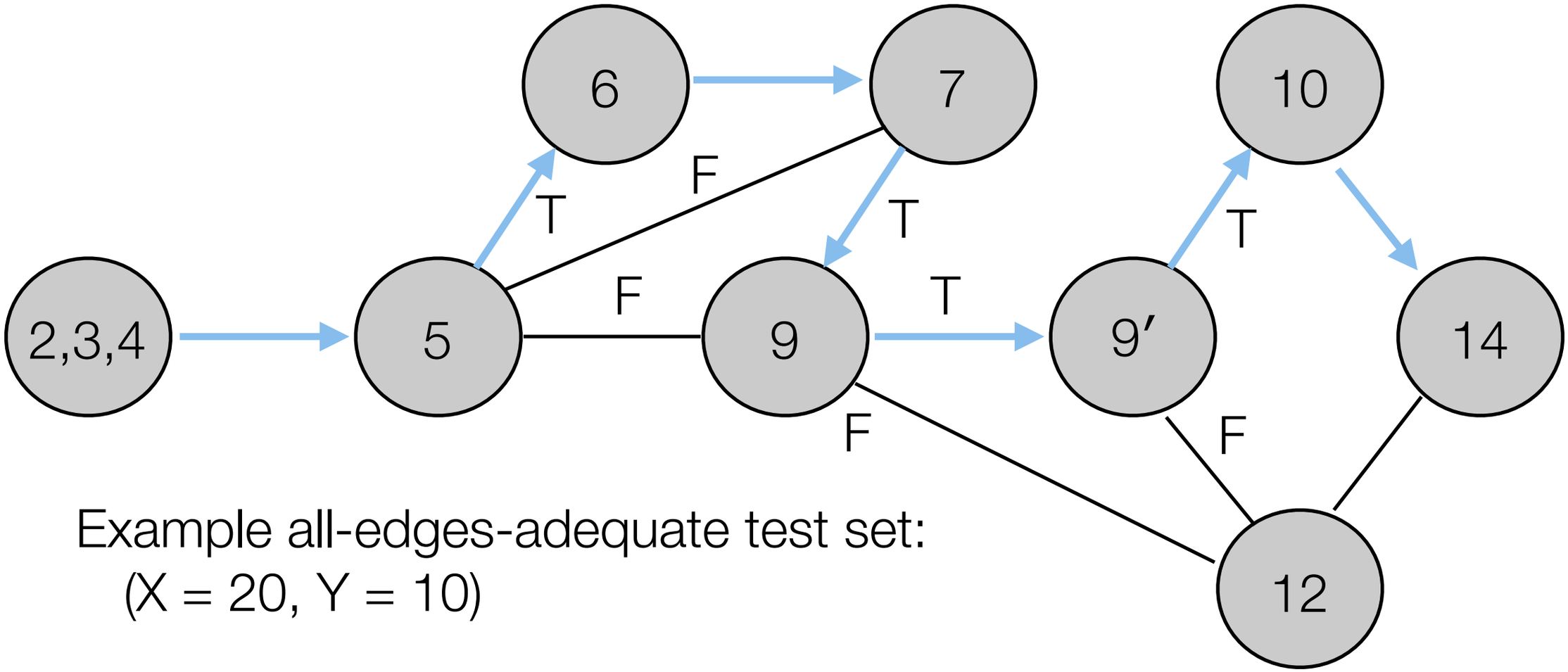
All-Edges Coverage of P



Example all-edges-adequate test set:

© Kenneth M. Anderson, 2012

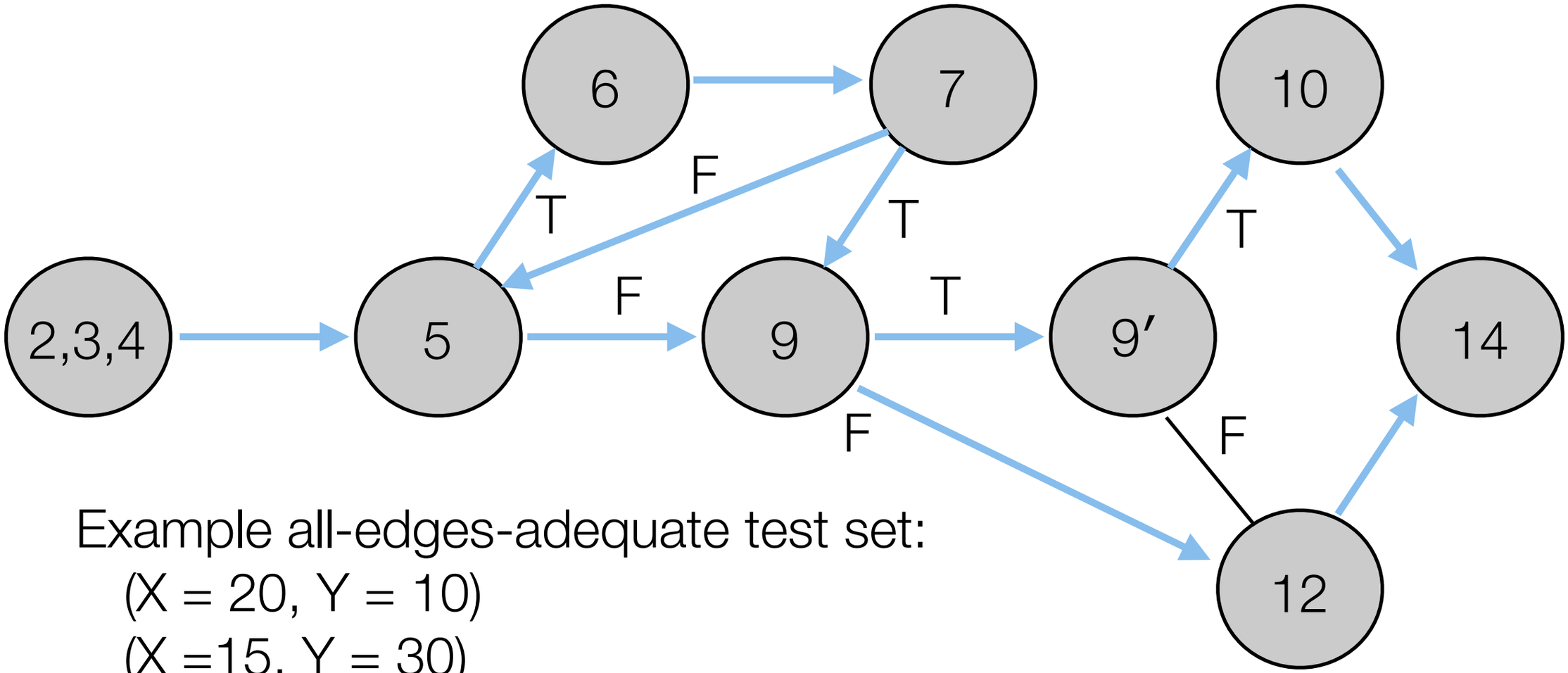
All-Edges Coverage of P



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

© Kenneth M. Anderson, 2012

All-Edges Coverage of P



Example all-edges-adequate test set:

(X = 20, Y = 10)

(X =15, Y = 30)

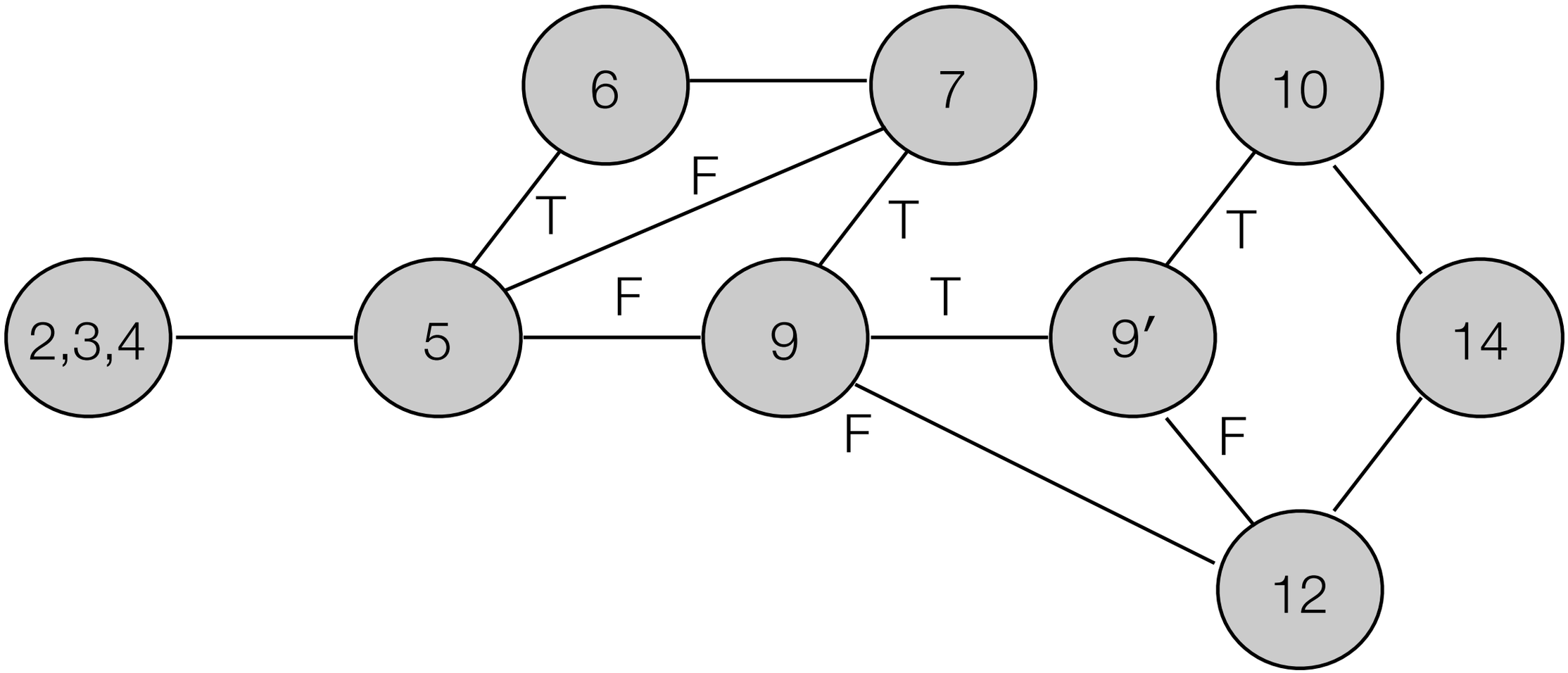
© Kenneth M. Anderson, 2012

White-box Testing Criteria

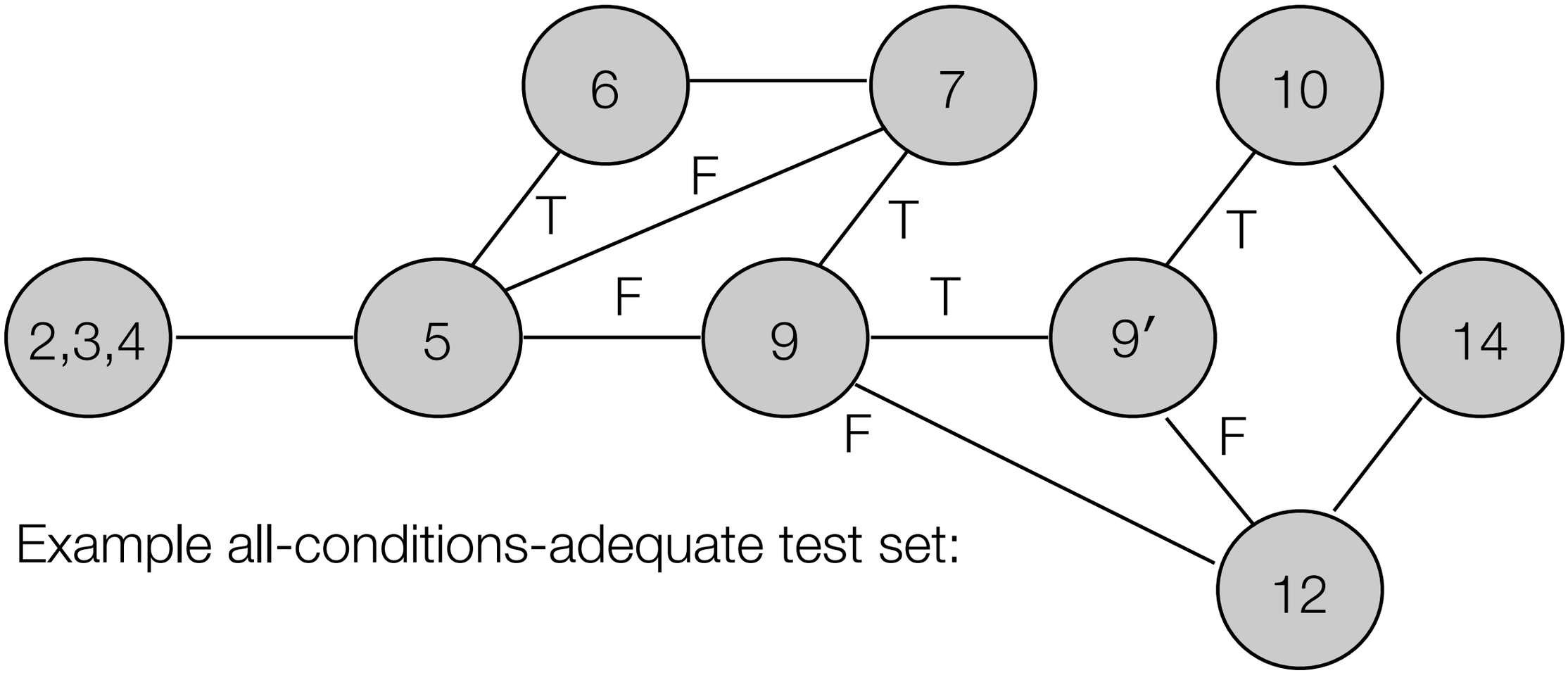
- Condition Coverage
  - Select a test set T such that
    - by executing P for each t in T
      - each edge of P's control flow graph is traversed at least once
      - and all possible values of the constituents of compound conditions are exercised at least once

© Kenneth M. Anderson, 2012

All-Conditions Coverage of P



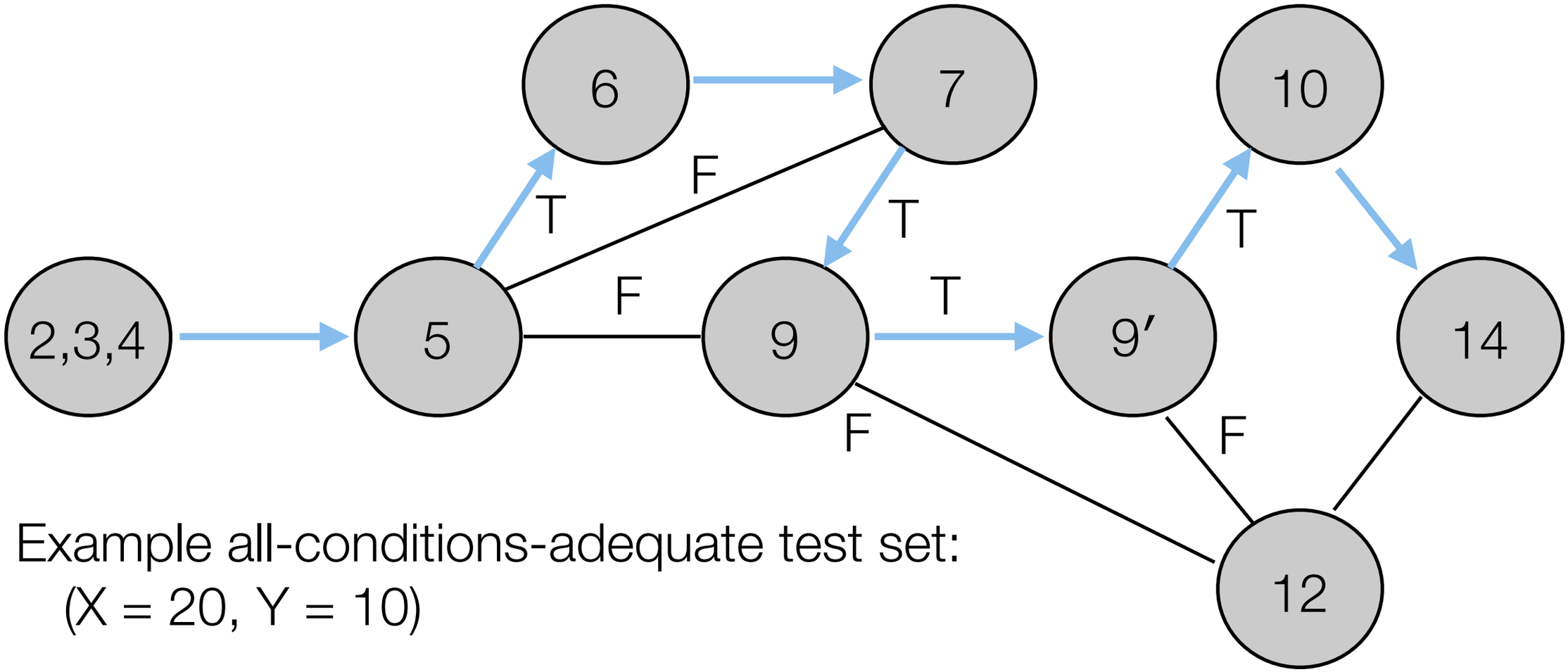
All-Conditions Coverage of P



Example all-conditions-adequate test set:

© Kenneth M. Anderson, 2012

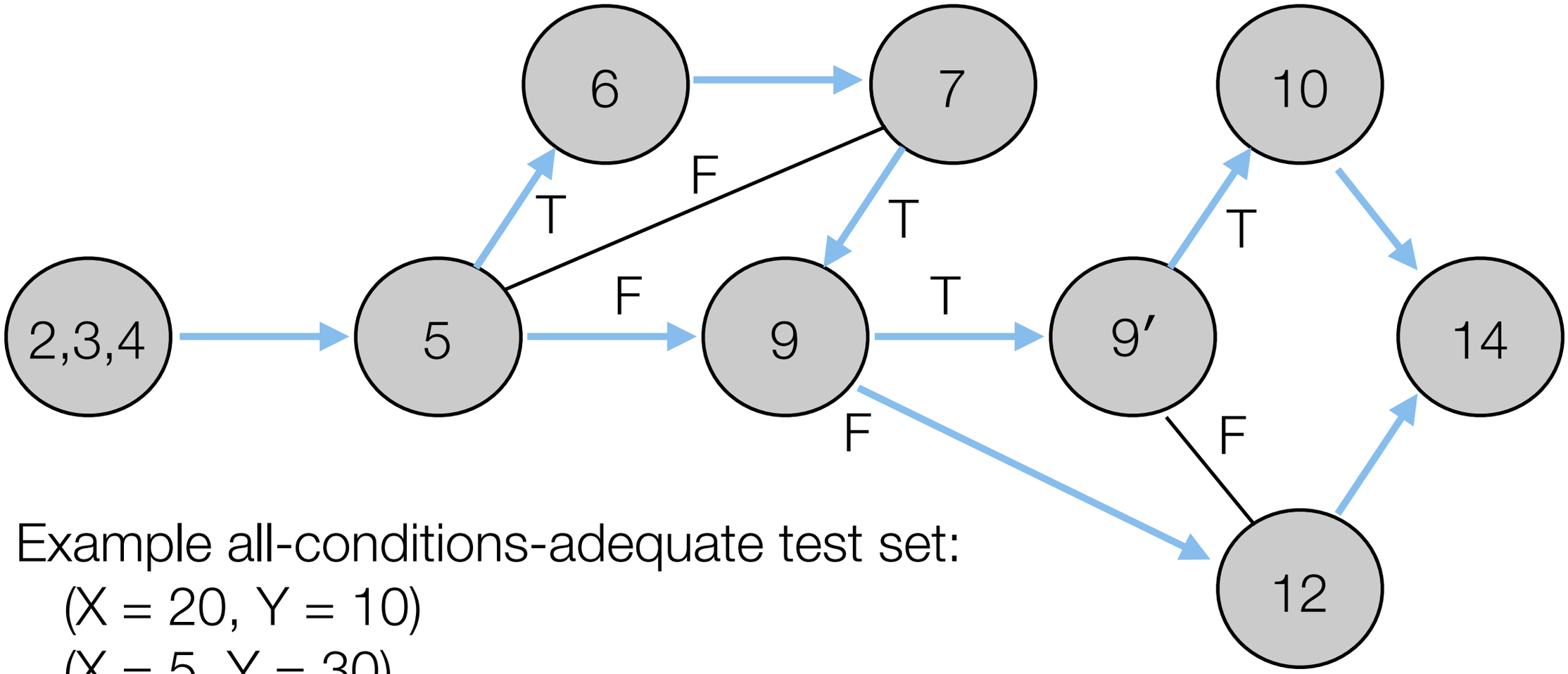
All-Conditions Coverage of P



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

© Kenneth M. Anderson, 2012

All-Conditions Coverage of P



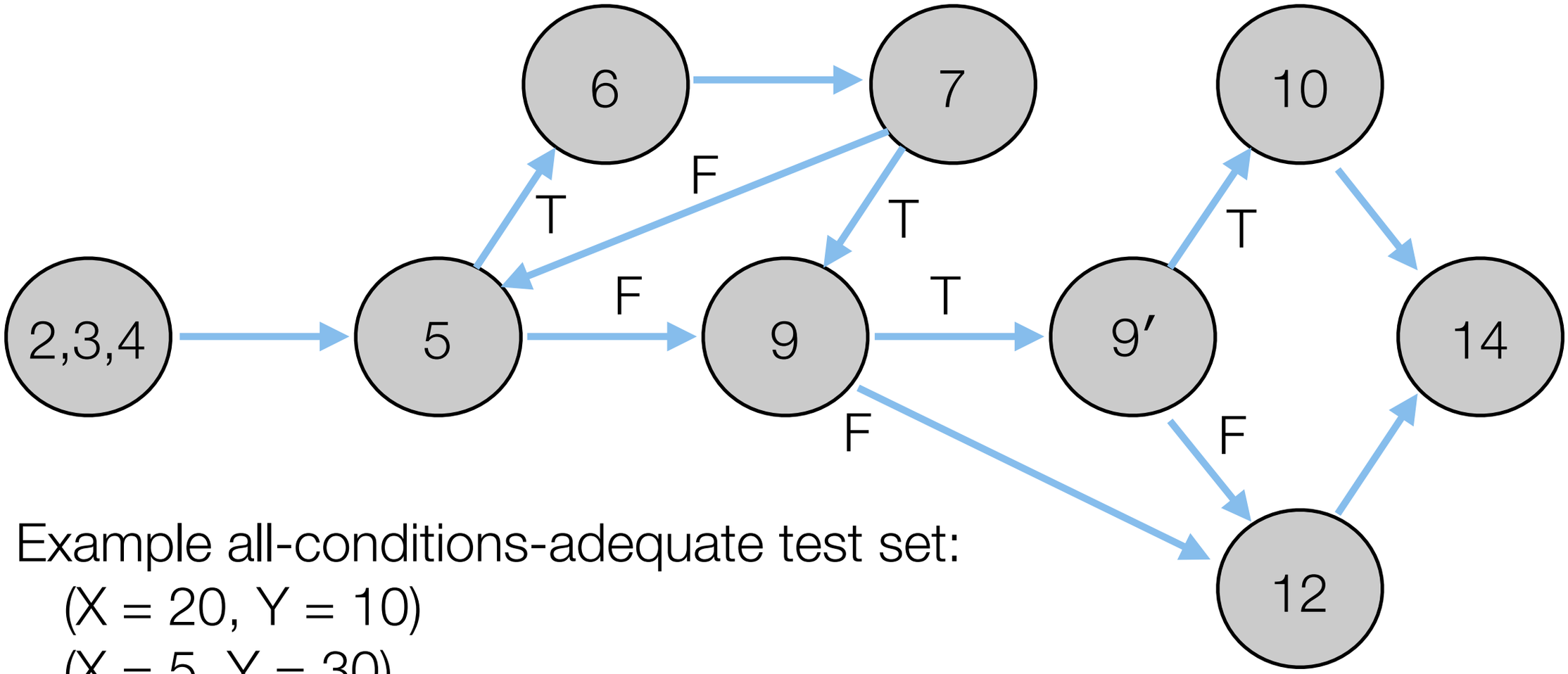
Example all-conditions-adequate test set:

(X = 20, Y = 10)

(X = 5, Y = 30)

© Kenneth M. Anderson, 2012

All-Conditions Coverage of P



Example all-conditions-adequate test set:

(X = 20, Y = 10)

(X = 5, Y = 30)

(X = 21, Y = 10)

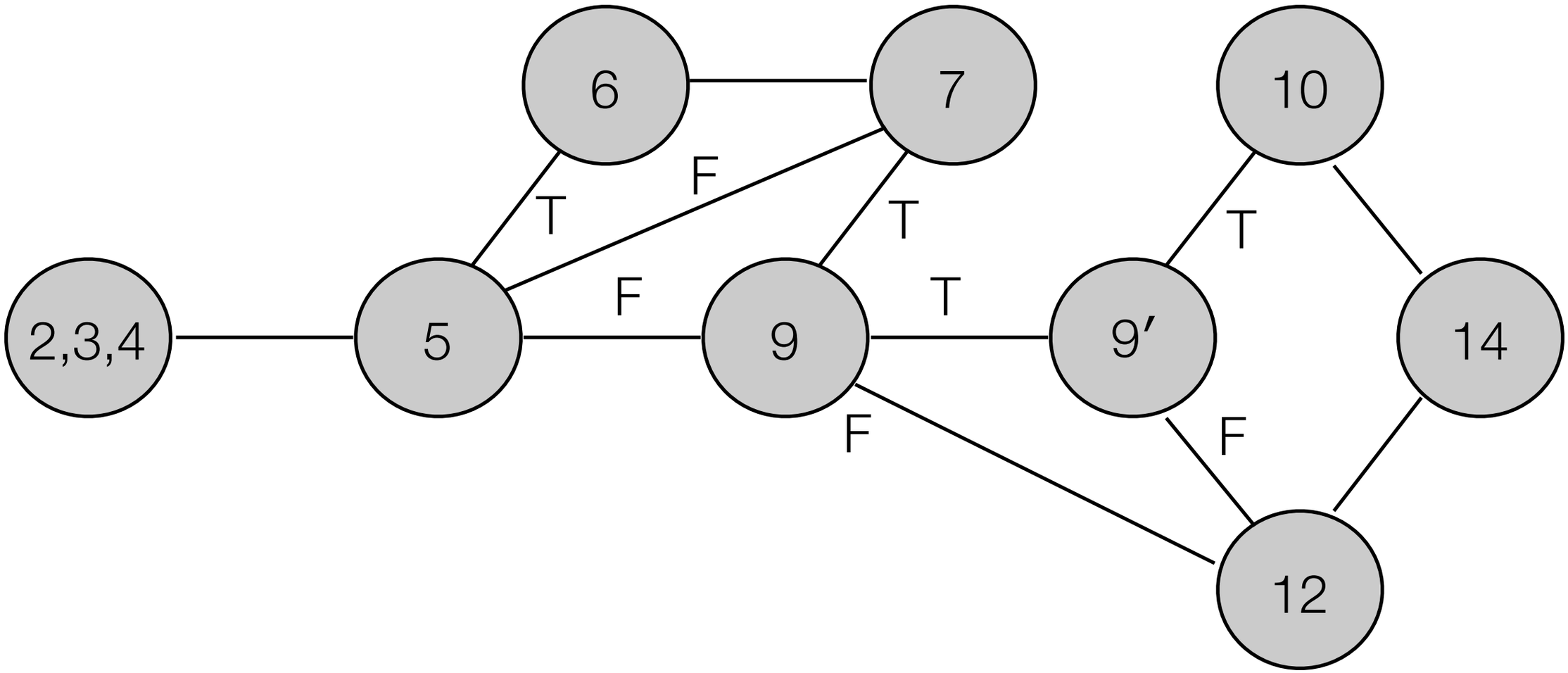
© Kenneth M. Anderson, 2012

White-box Testing Criteria

- 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

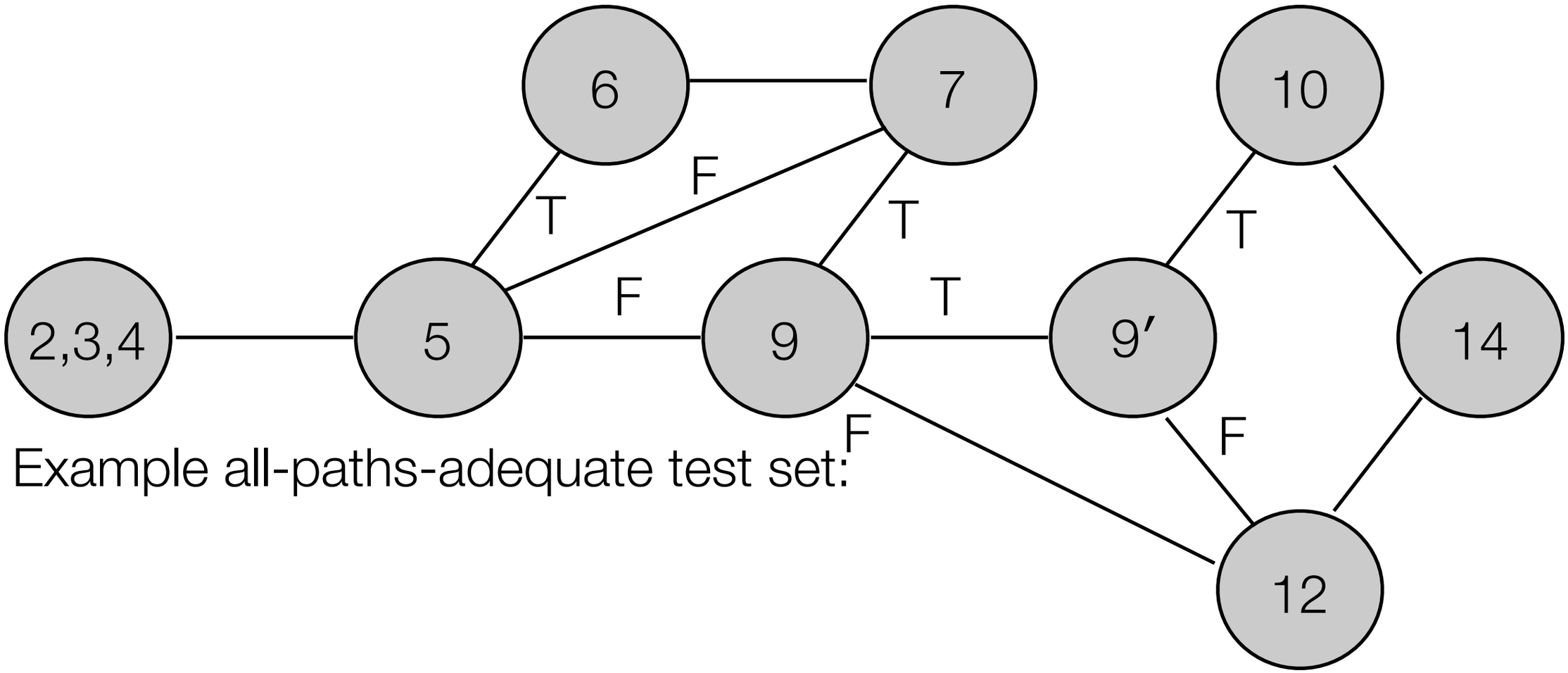
© Kenneth M. Anderson, 2012

All-Paths Coverage of P



© Kenneth M. Anderson, 2012

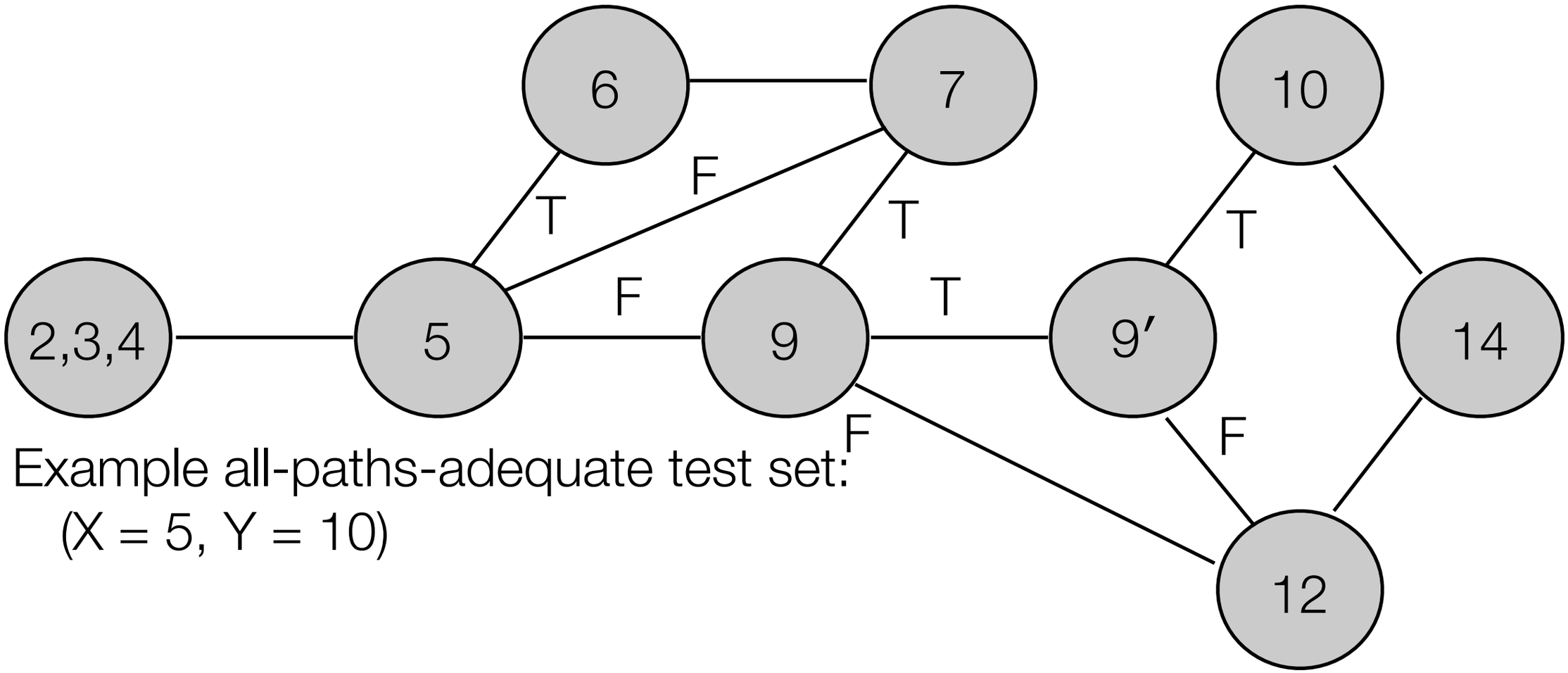
All-Paths Coverage of P



Example all-paths-adequate test set:

© Kenneth M. Anderson, 2012

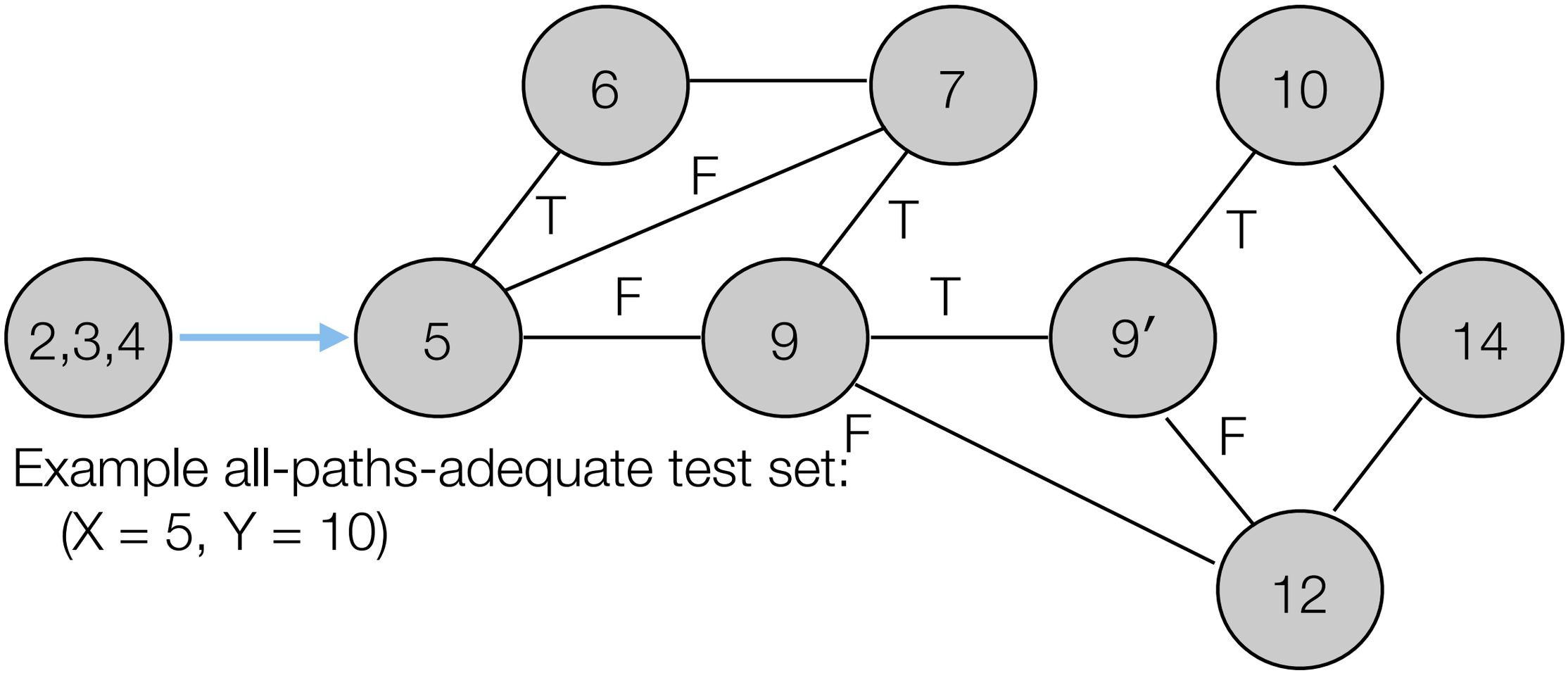
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

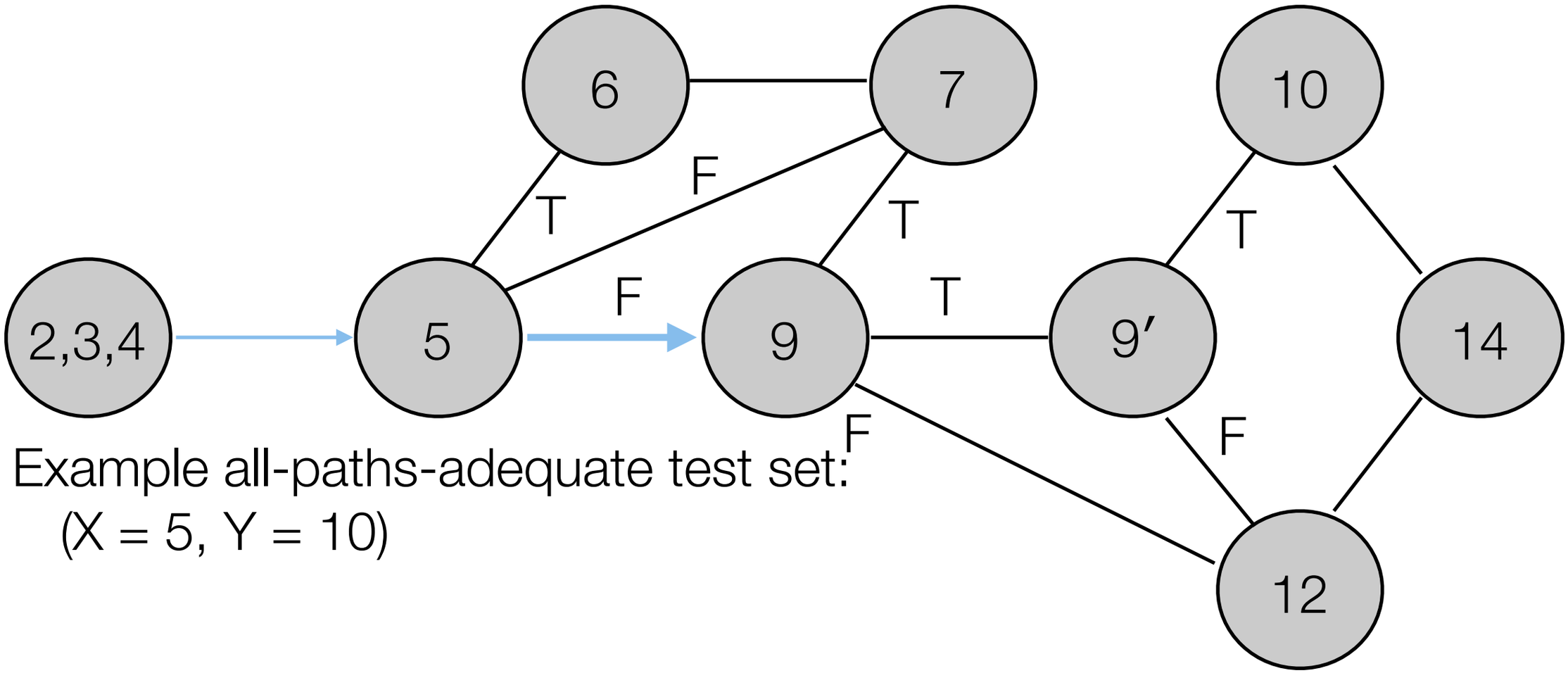
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

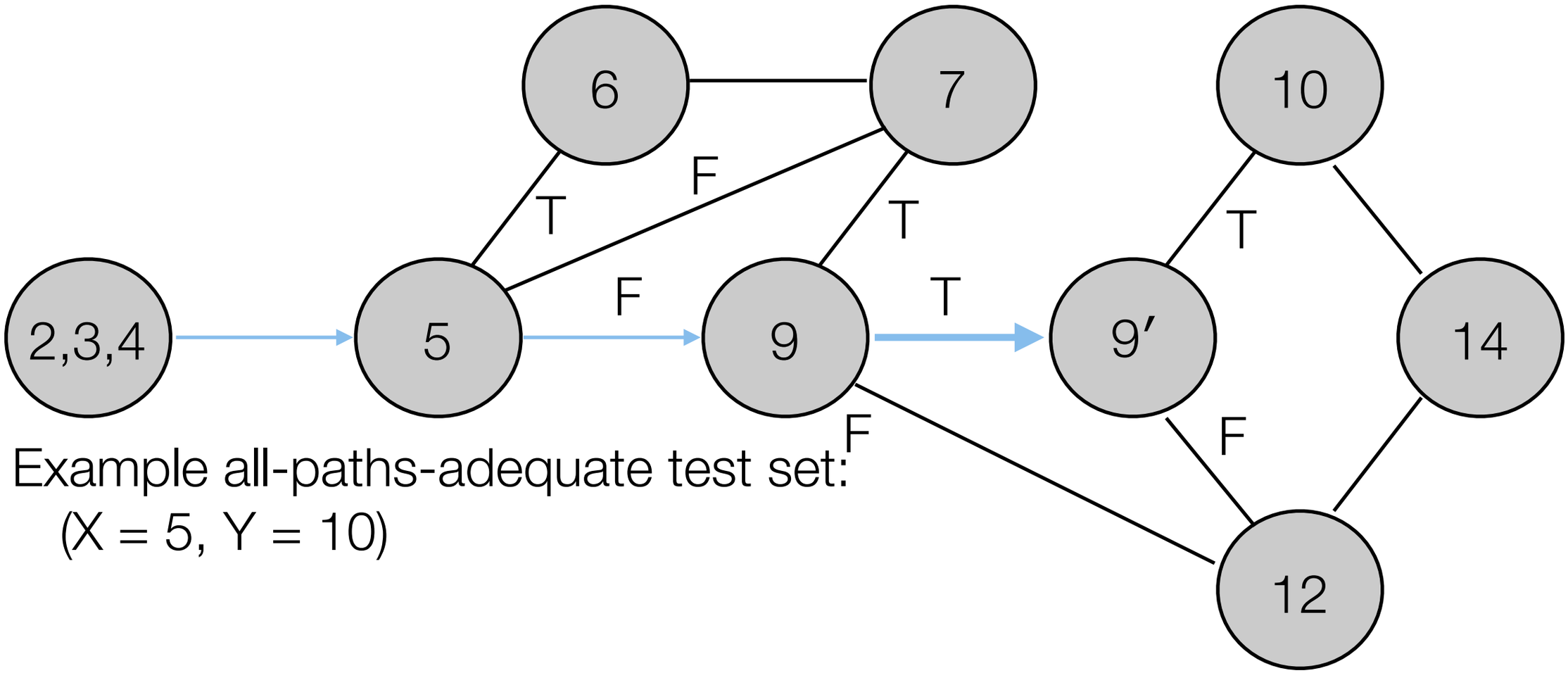
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

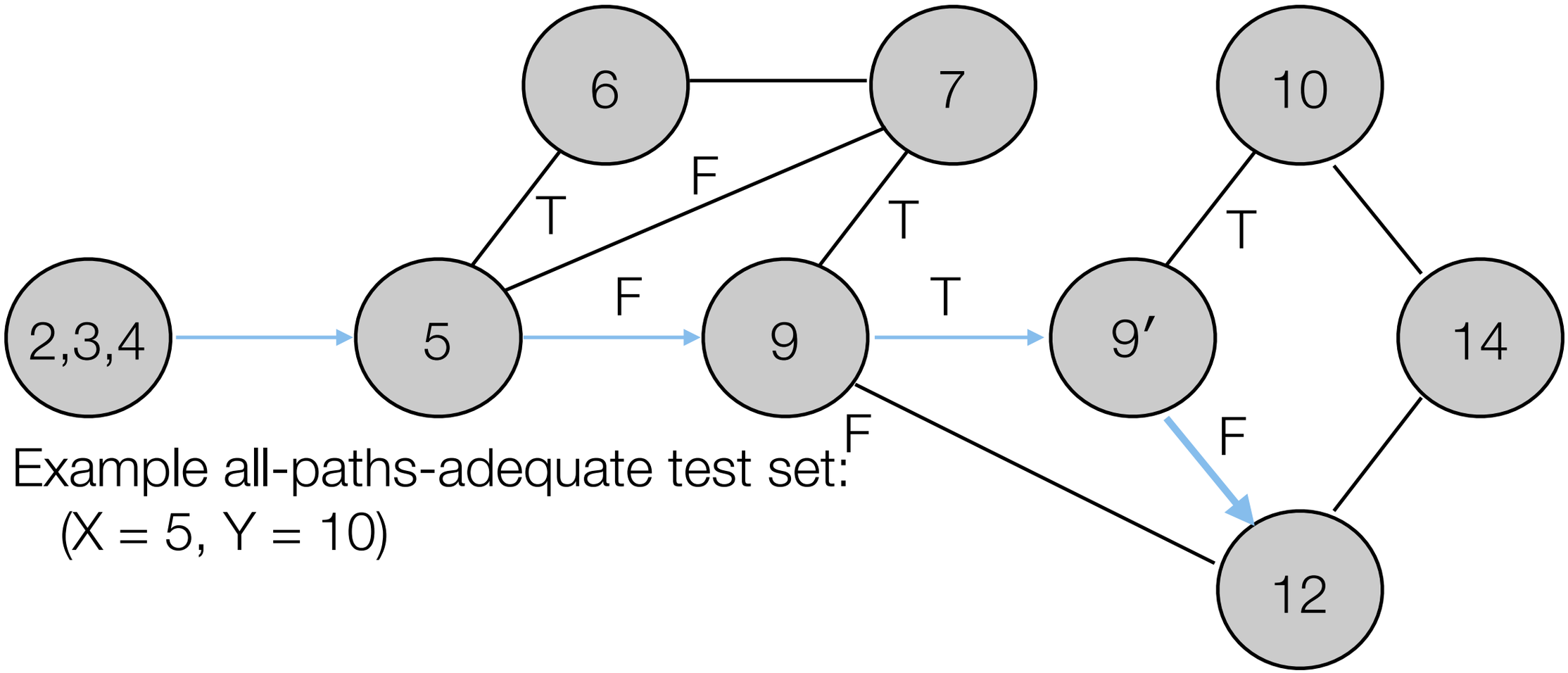
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

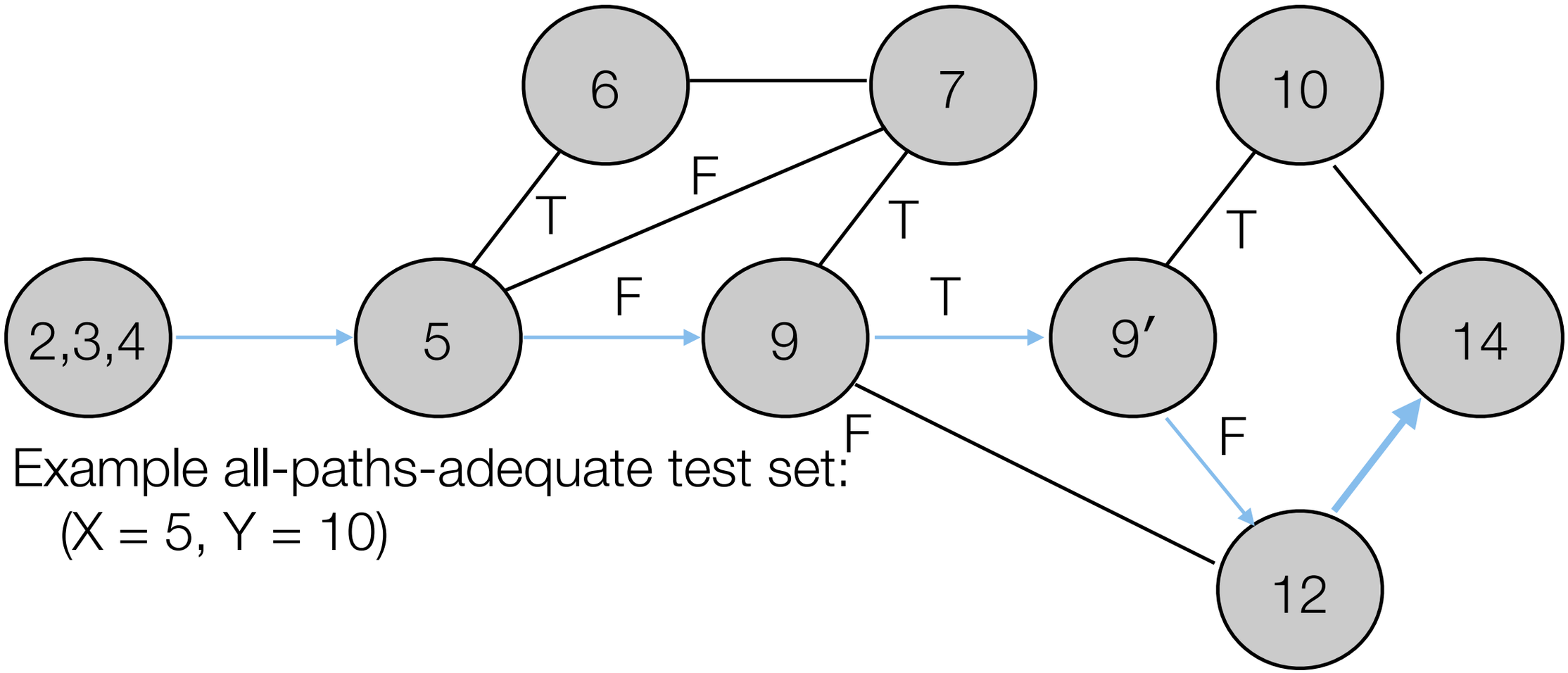
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

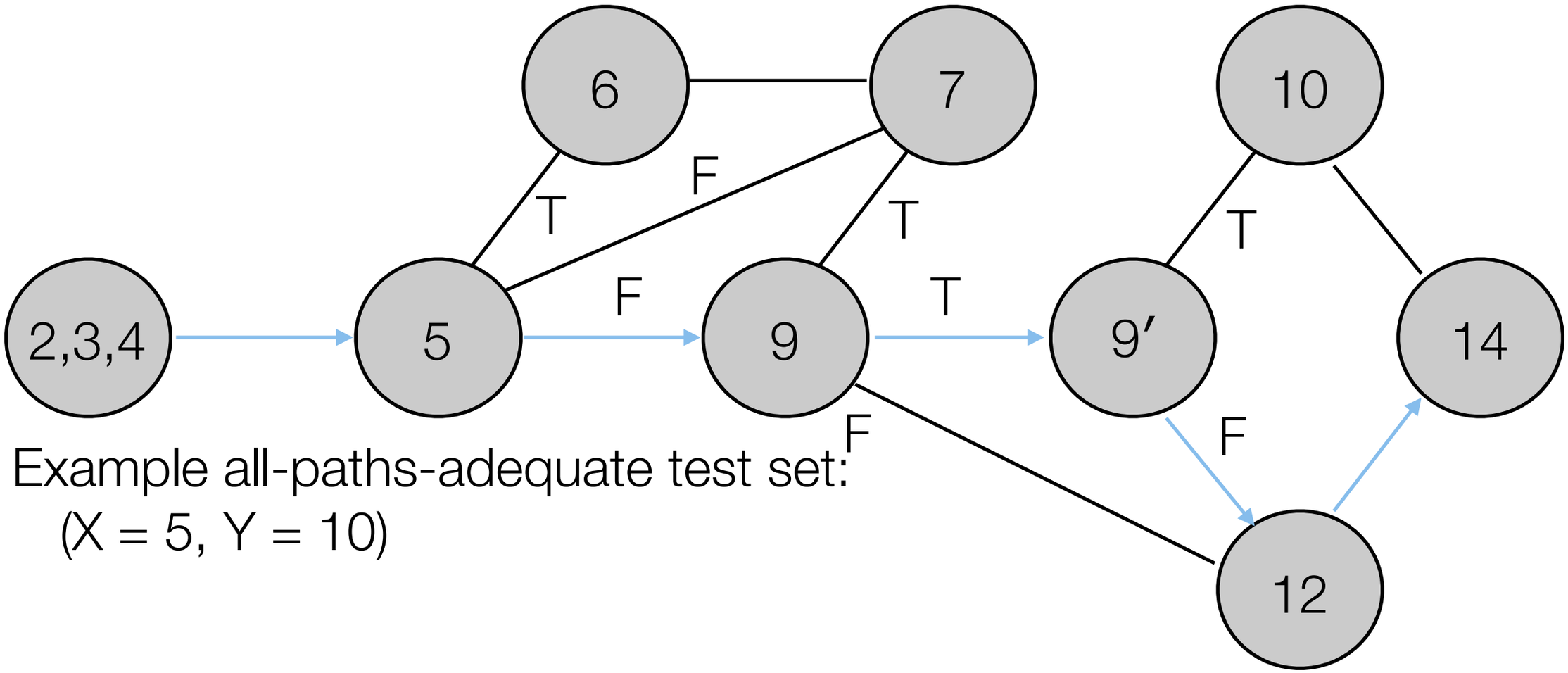
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

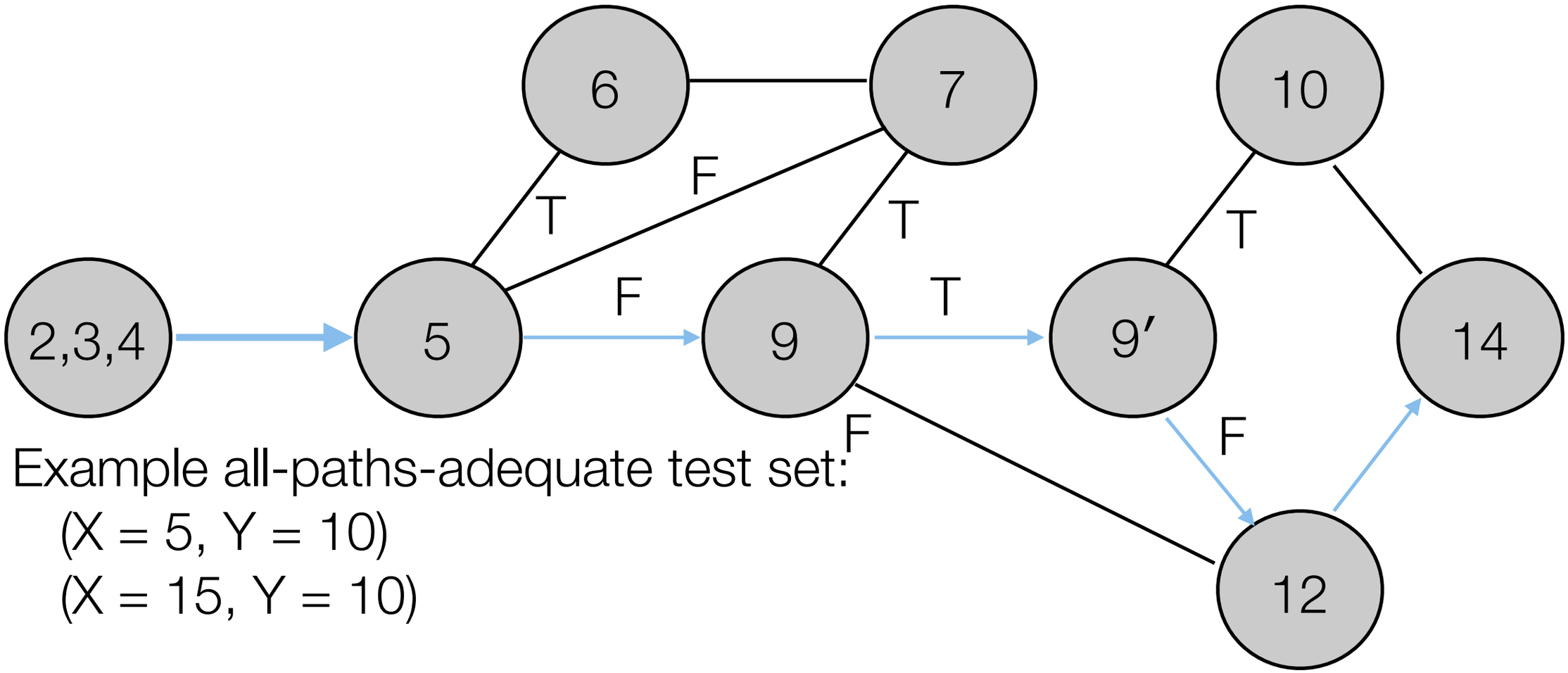
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



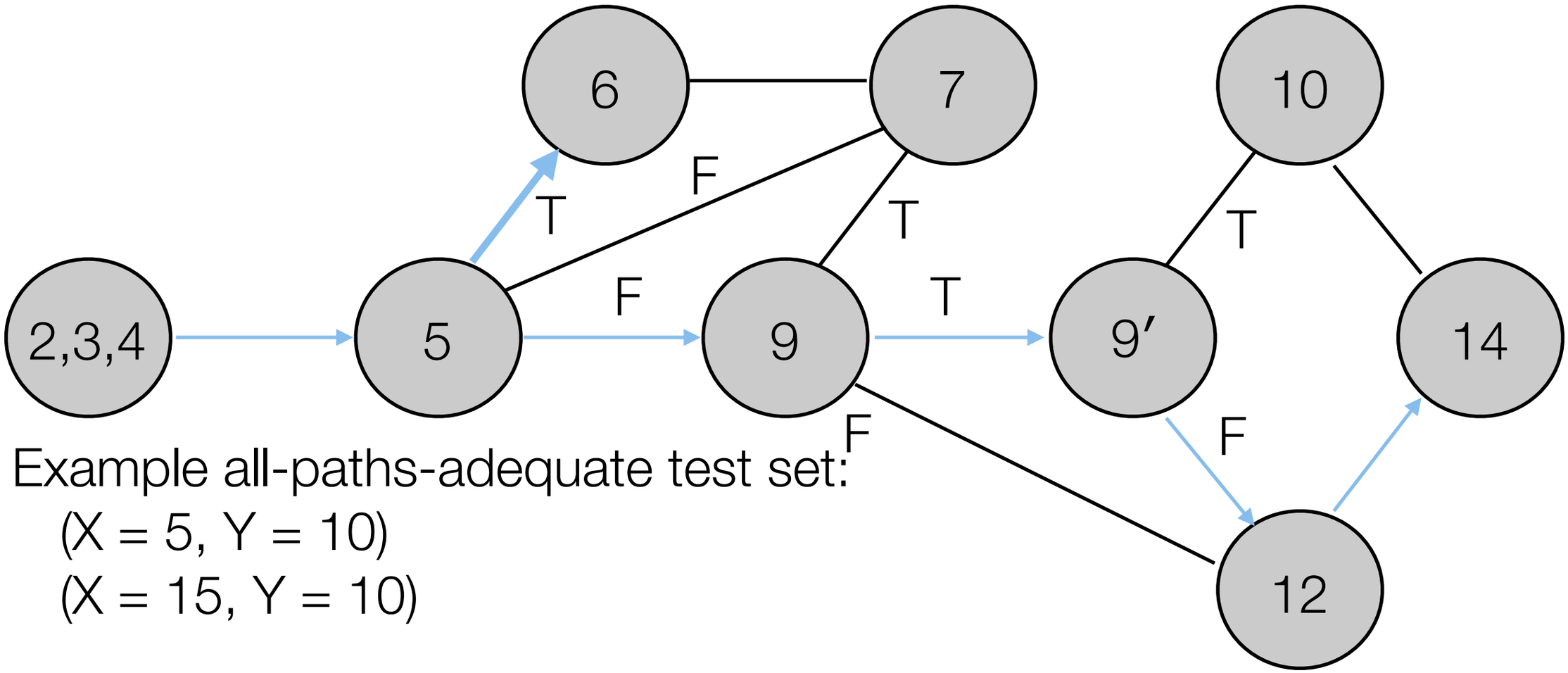
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



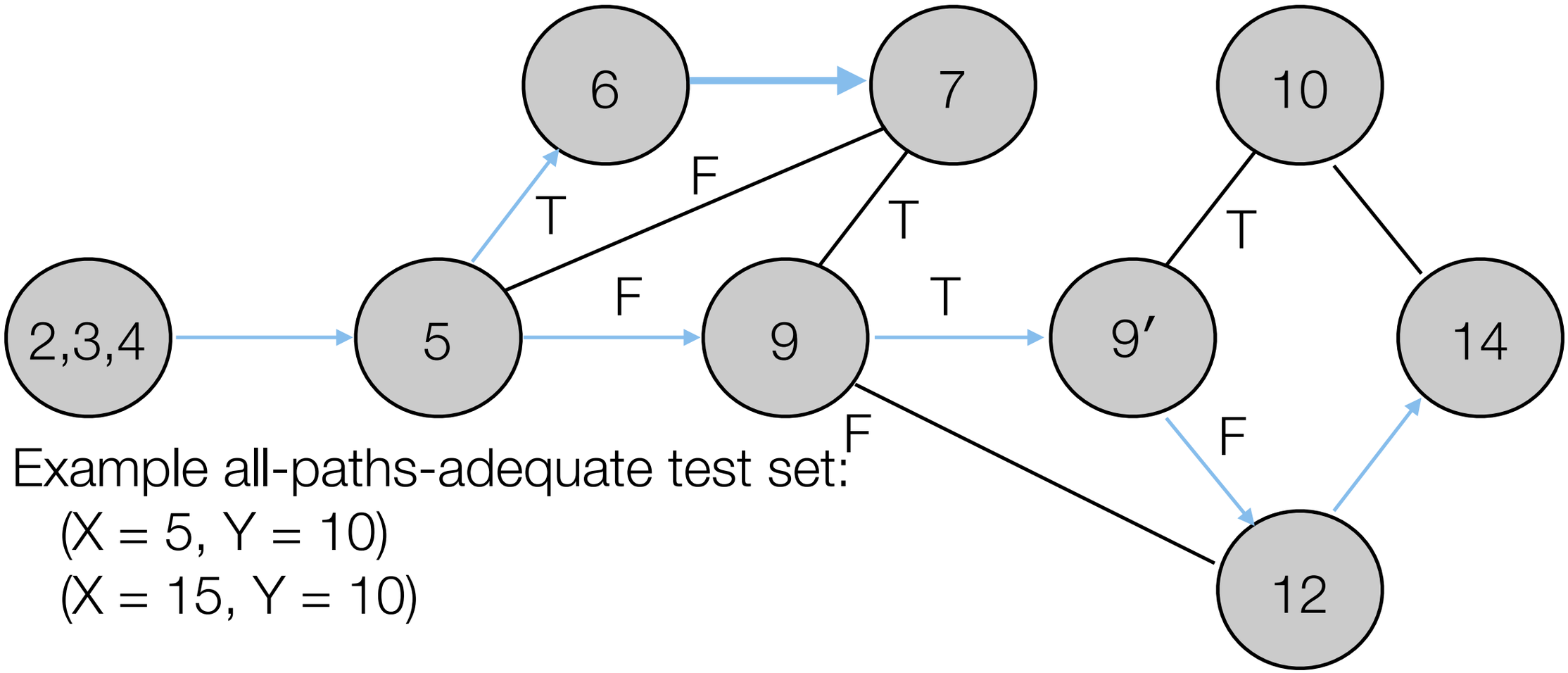
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



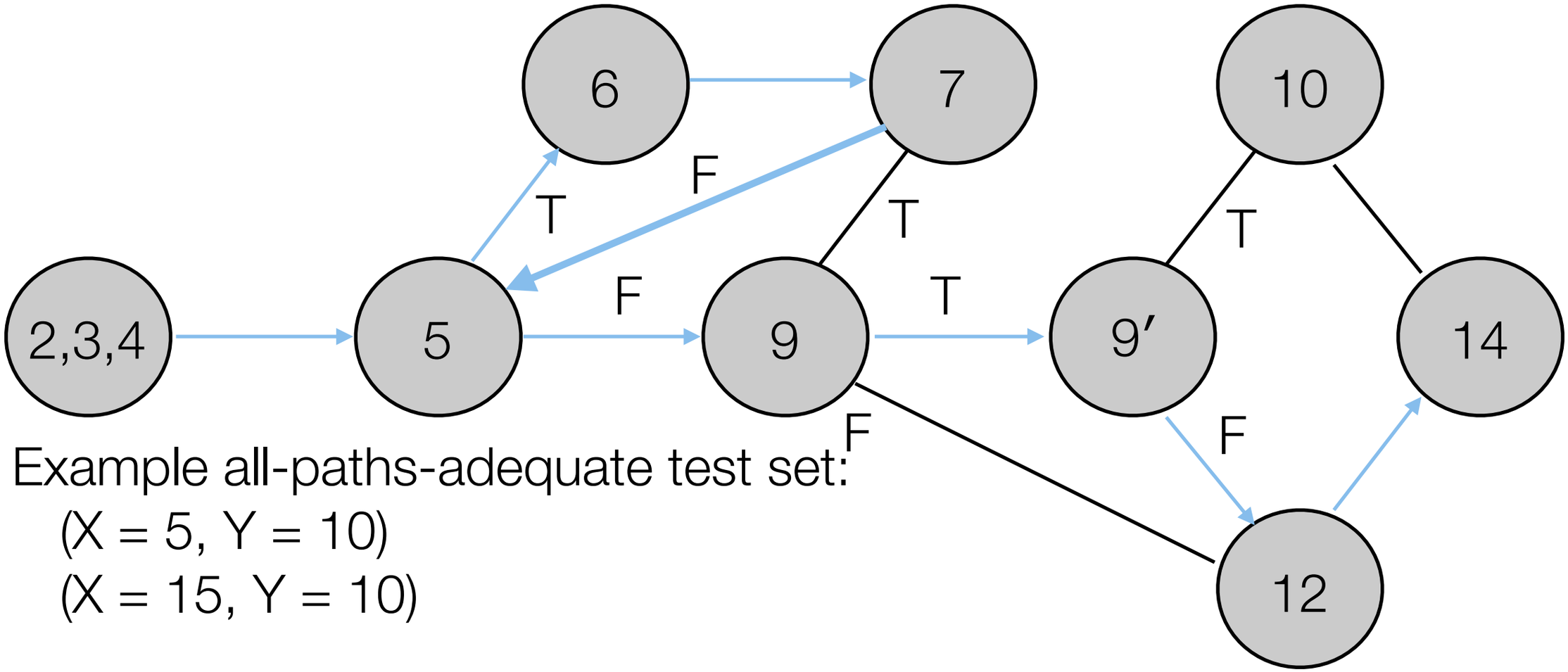
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



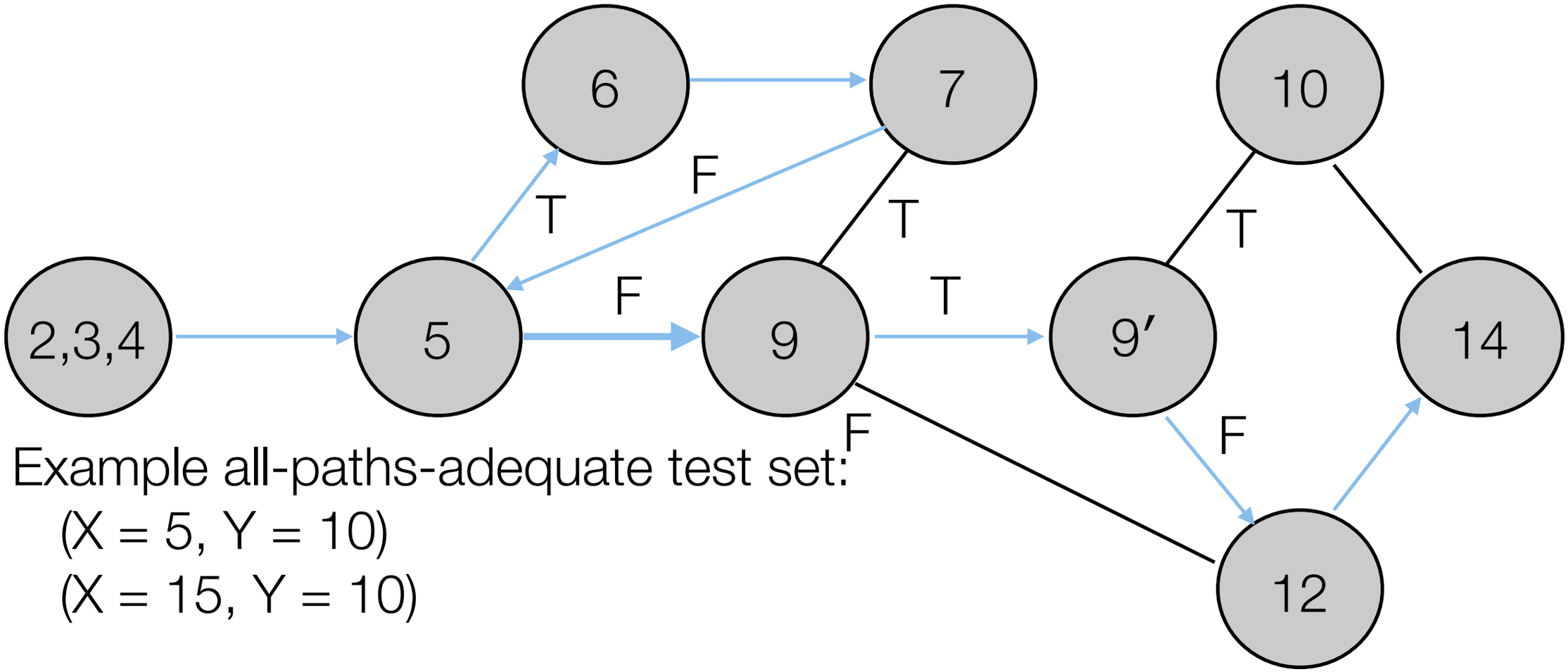
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



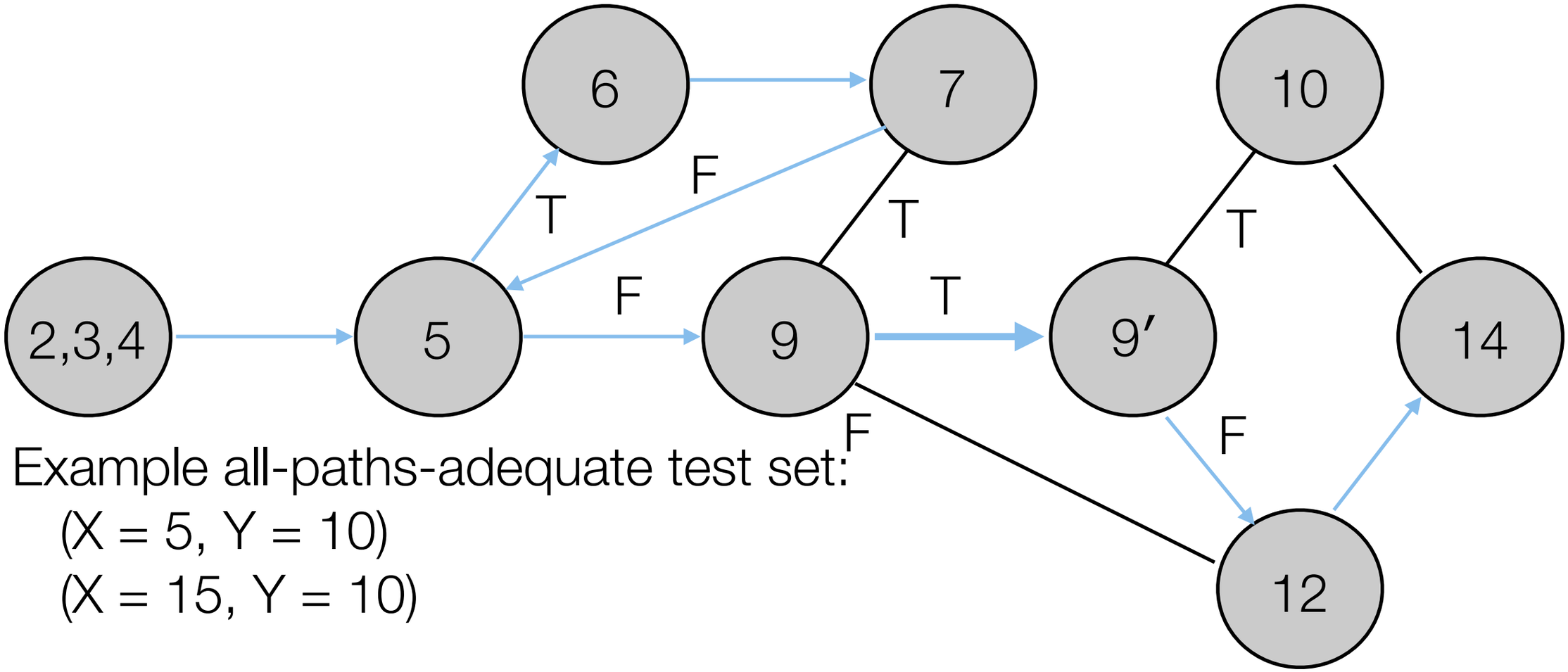
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



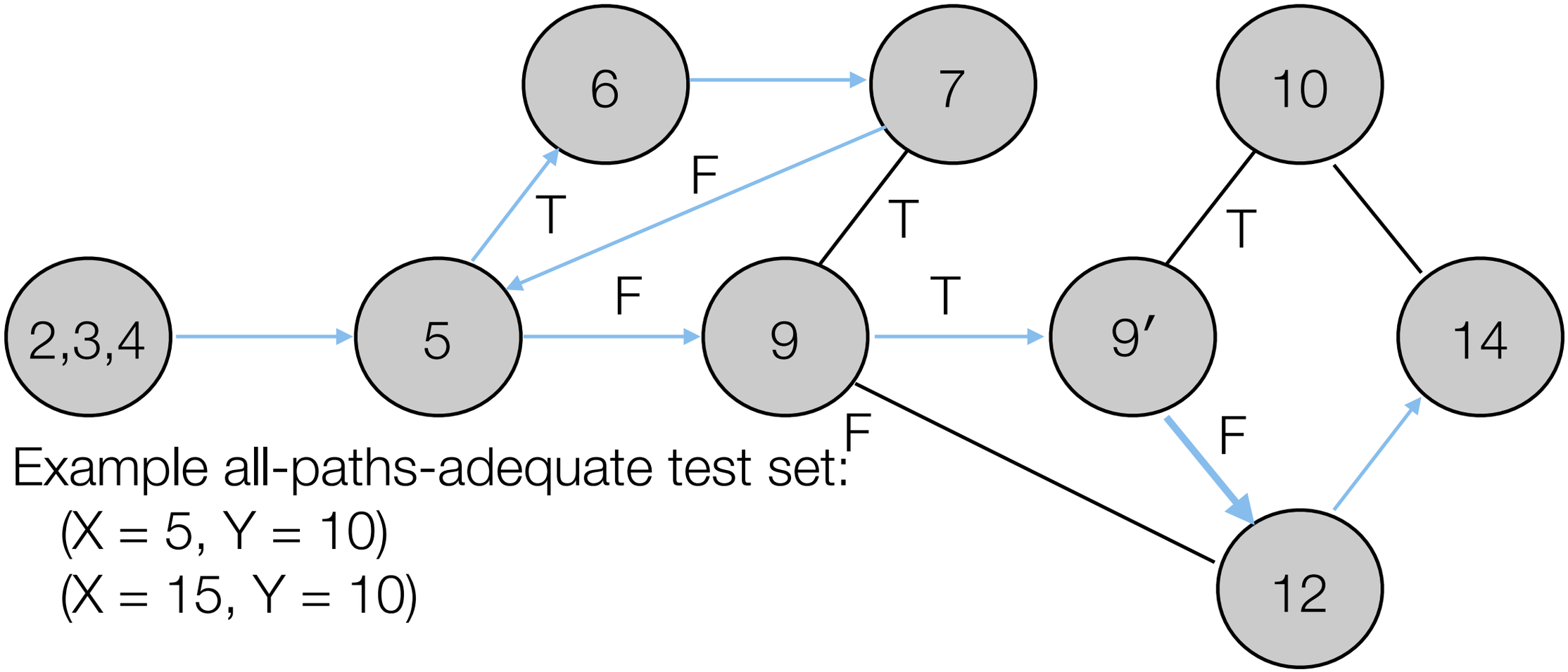
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



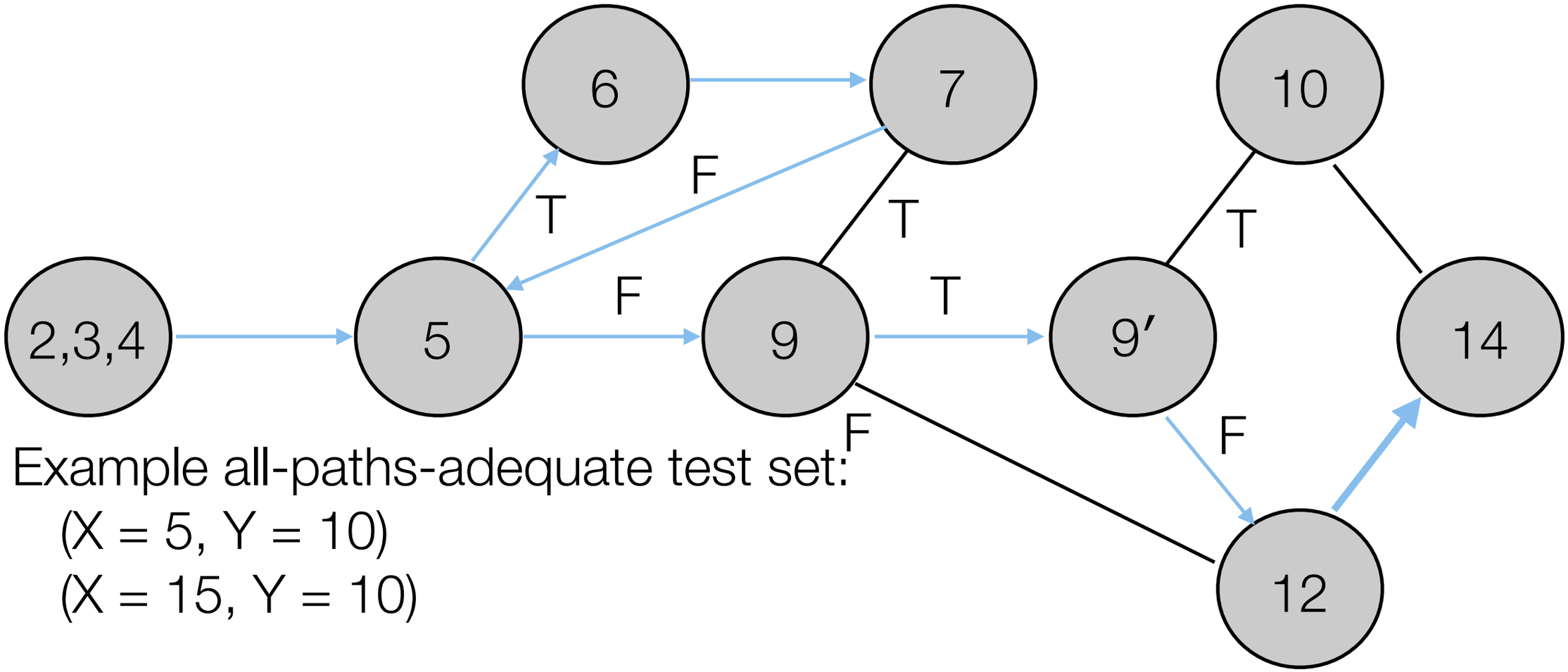
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



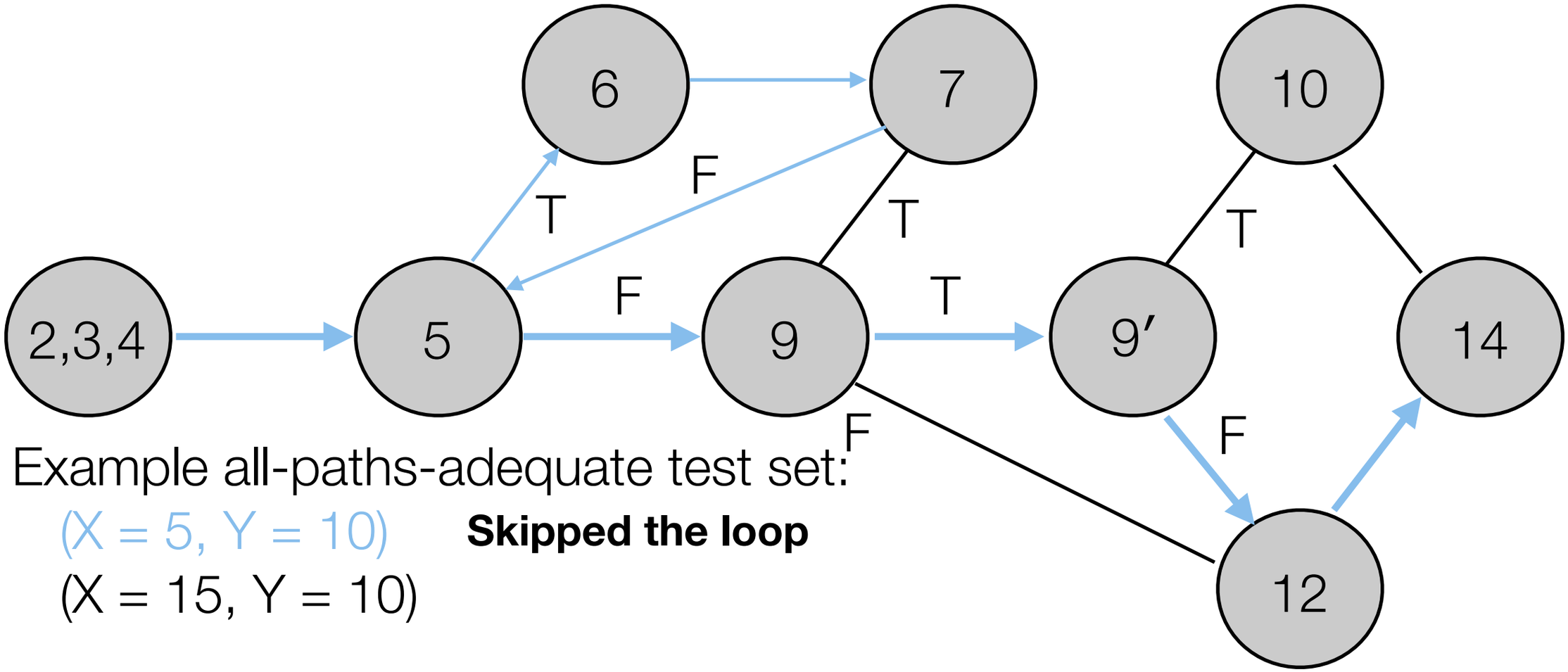
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



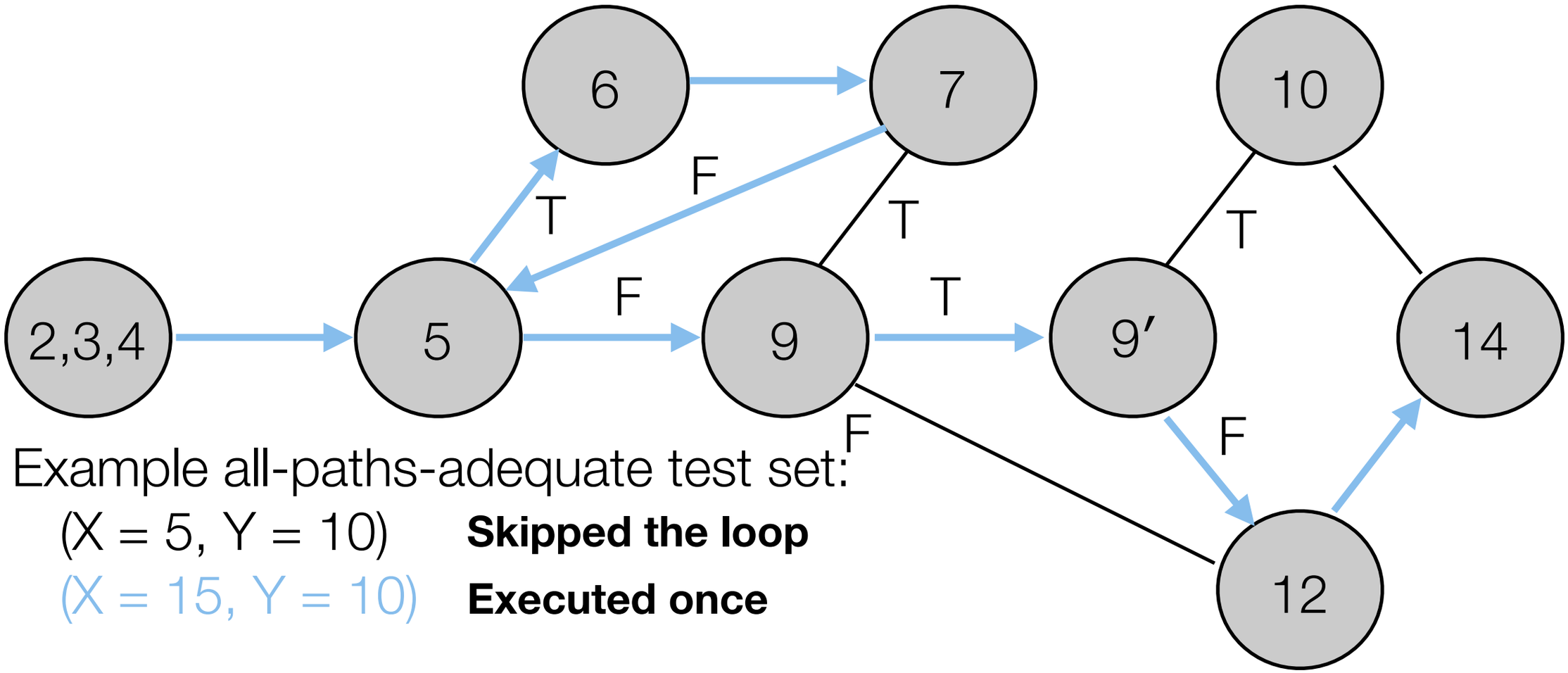
Example all-paths-adequate test set:

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

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



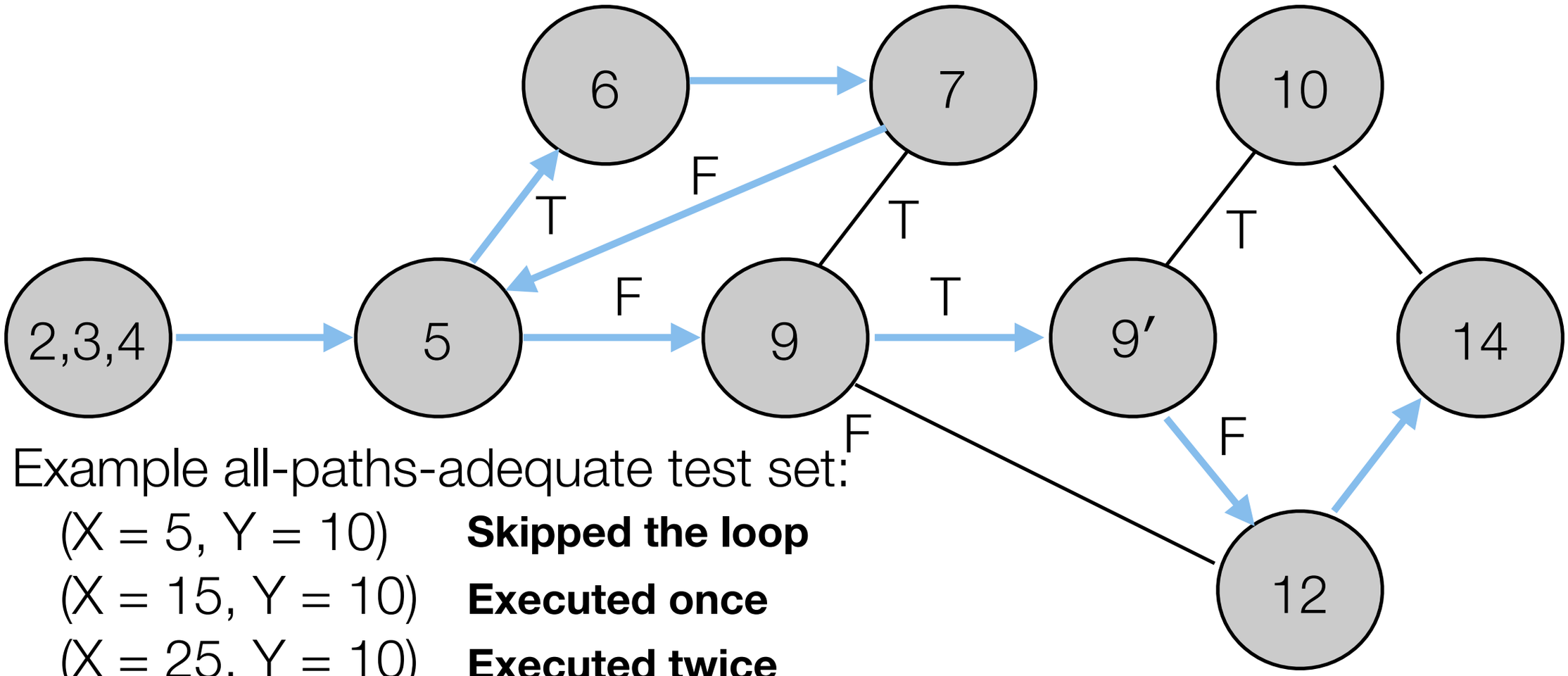
Example all-paths-adequate test set:

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

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

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



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**

And so on… you would also want permutations that exit the loop early

© Kenneth M. Anderson, 2012

Code Coverage Tools

- 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
    - These systems 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

© Kenneth M. Anderson, 2012

Testing Automation (I)

- 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!!)

© Kenneth M. Anderson, 2012

Testing Automation (II)

- 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
- We'll see examples of this later in the semester

© Kenneth M. Anderson, 2012

Continuous Integration

- Since test automation is so critical, systems known as continuous integration frameworks have emerged
- 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

© Kenneth M. Anderson, 2012

Summary

- 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 exists to integrate testing into the version control and build management processes of a development organization

© Kenneth M. Anderson, 2012

Coming Up Next:

- Lecture 6: Agile Methods and Agile Teams
- Lecture 7: Division of Labor and Design Approaches in Concurrency

© Kenneth M. Anderson, 2012

Meeting:title

Date:2021-08-27

Time:09:24:00

Venue:venue

Attendance
Present:
LIM G WEI2,small admin12
LIM G WEI,big one
Absent :

1.Introduction

2.first title

2.1.agenda 1

description of agenda 1

2.2.agenda 3

description agenda 4

3.second title

3.1.agenda 2

description agenda 2

3.2.agenda 4

description agenda 4

Introduction to Software Testing

CSCI 5828: Foundations of Software Engineering
Lecture 05 — 01/31/2012

© Kenneth M. Anderson, 2012

Goals

- Provide introduction to fundamental concepts of software testing
    - Terminology
    - Testing of Systems
        - unit tests, integration tests, system tests, acceptance tests
    - Testing of Code
        - Black Box
        - Gray Box
        - White Box
        - Code Coverage

© Kenneth M. Anderson, 2012

Testing

- Testing is a **critical element** of software development life cycles
  - called **software quality control** or **software quality assurance**
  - basic goals: **validation** and **verification**
    - validation: **are we building the right product?**
      - verification: **does "X" meet its specification?**
        - where "X" 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

© Kenneth M. Anderson, 2012

Terminology

- An **error** is a mistake made by an engineer
  - often a misunderstanding of a requirement or design specification
- A **fault** is a manifestation of that error in the code
  - what we often call “a bug”
- A **failure** is an incorrect output/behavior that is caused by executing a fault
  - The failure may occur immediately (crash!) or much, much later in the execution
- **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?

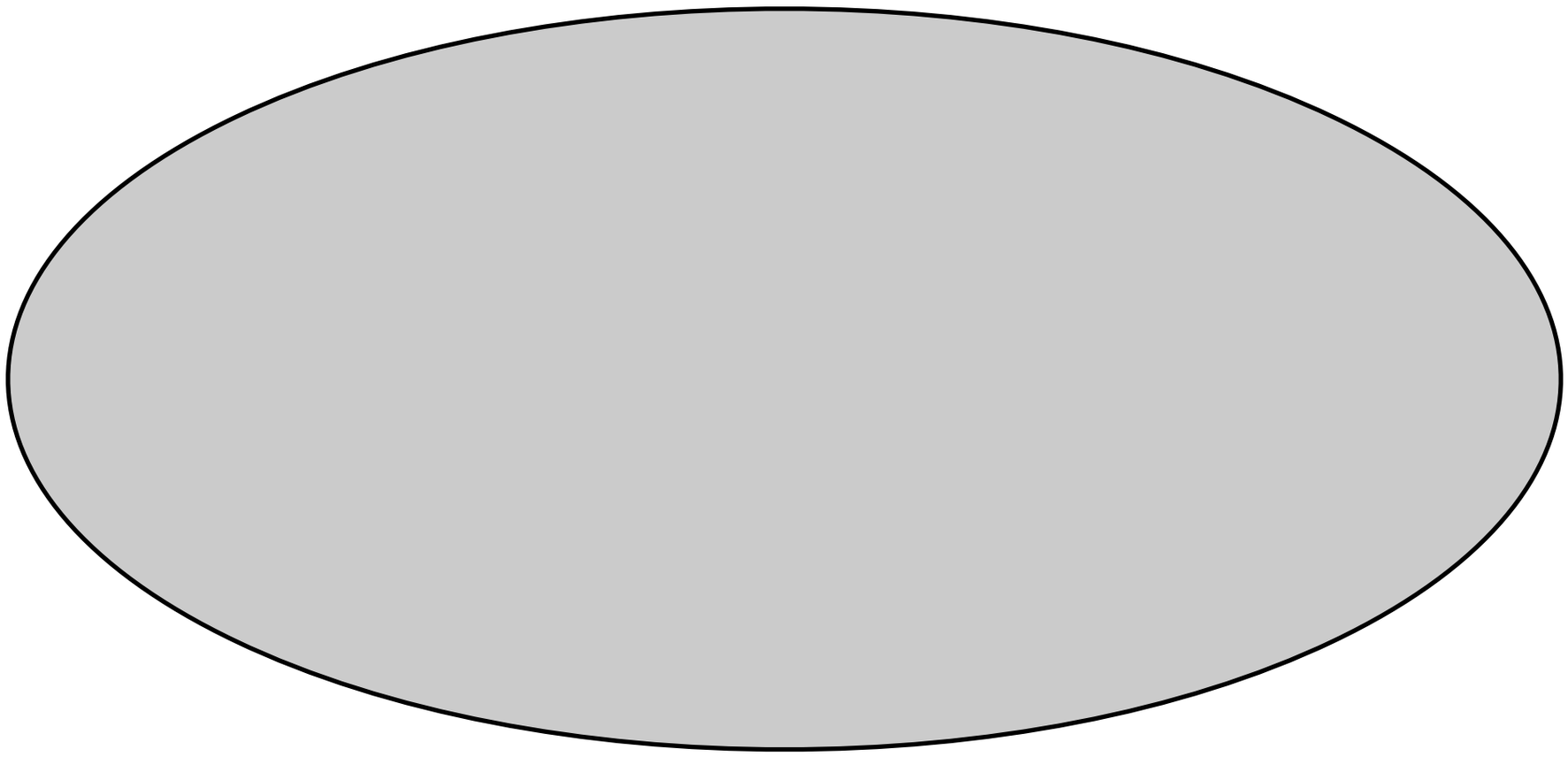
© Kenneth M. Anderson, 2012

No!

- 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 resulting in no visible failures!
      - this is particularly insidious
- However, if we do a good job in creating a test set that
  - covers all functional capabilities of a system
  - and 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

© Kenneth M. Anderson, 2012

Looking for Faults

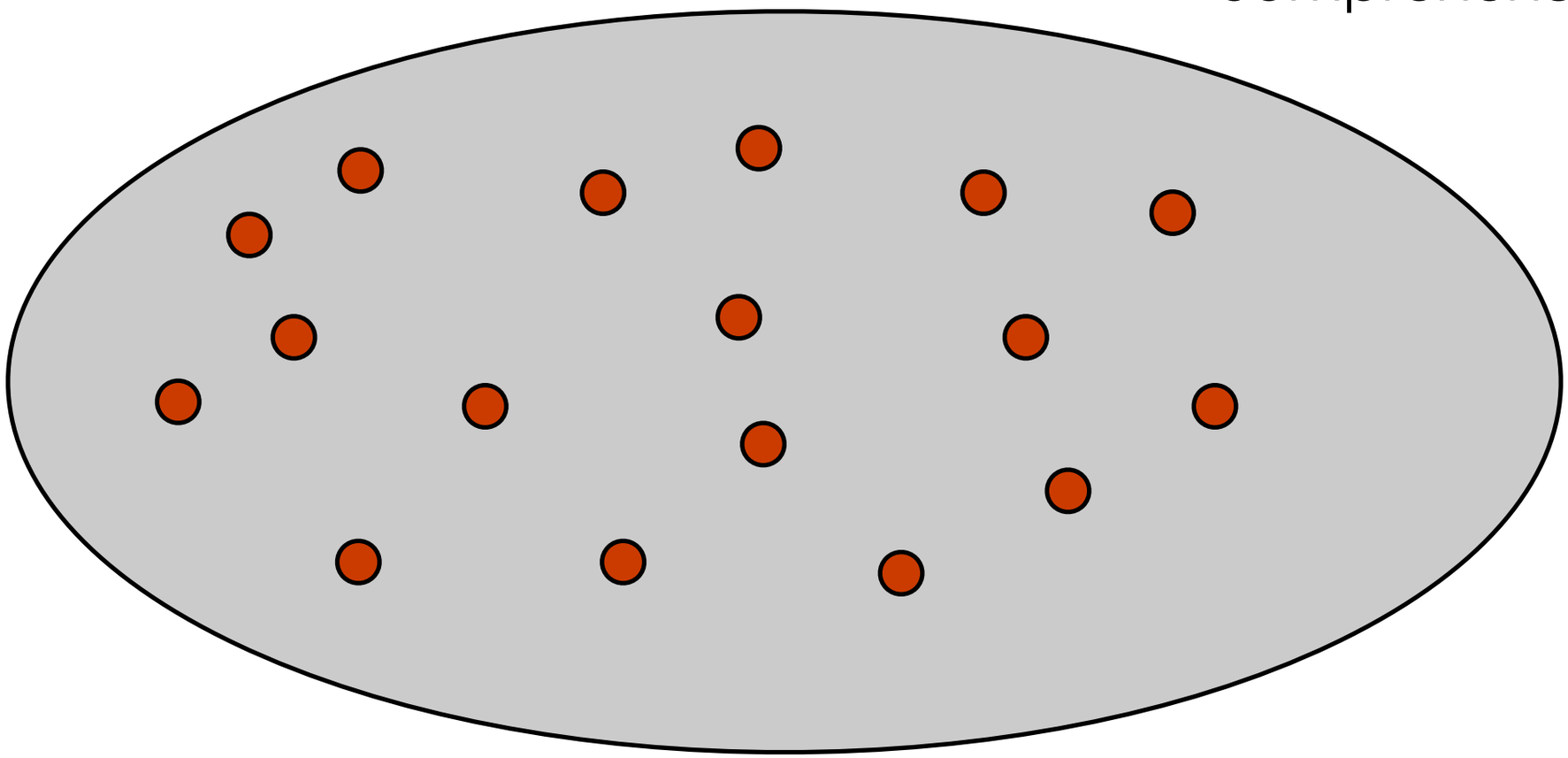


All possible states/behaviors of a system

© Kenneth M. Anderson, 2012

Looking for Faults

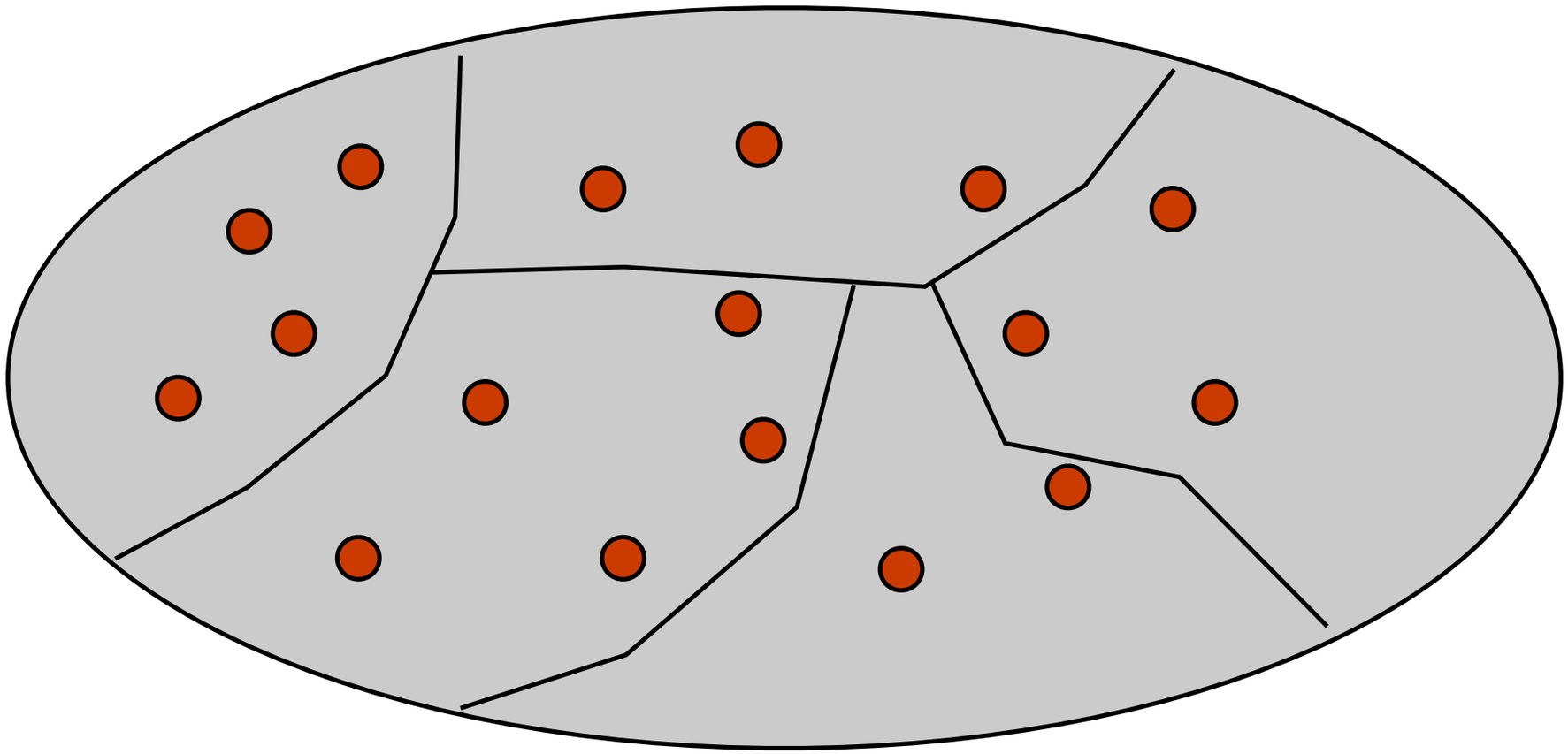
As you can see, its not very comprehensive



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

© Kenneth M. Anderson, 2012

One way forward? Fold



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

© Kenneth M. Anderson, 2012

What does that mean?

- 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

© Kenneth M. Anderson, 2012

Completeness

- 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?

© Kenneth M. Anderson, 2012

Continuous Testing

- Testing is a continuous process that should be performed at every stage of a software development process
    - During requirements gathering, for instance, we must continually query the user, “Did we get this right?”
    - Facilitated by an emphasis on iteration throughout a life cycle
- at the end of each iteration
    - we check our results to see if what we built is meeting our requirements (specification)

© Kenneth M. Anderson, 2012

Testing the System (I)

- **Unit Tests**
  - Tests that cover low-level aspects of a system
    - For each module, does each operation perform as expected
    - For method **foo()**, we'd like to see another method **testFoo()**
- **Integration Tests**
  - Tests that check that modules work together in combination
  - Most projects on schedule until they hit this point (MMM, Brooks)
    - 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 (Mars Polar Lander)

© Kenneth M. Anderson, 2012

Testing the System (II)

- **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

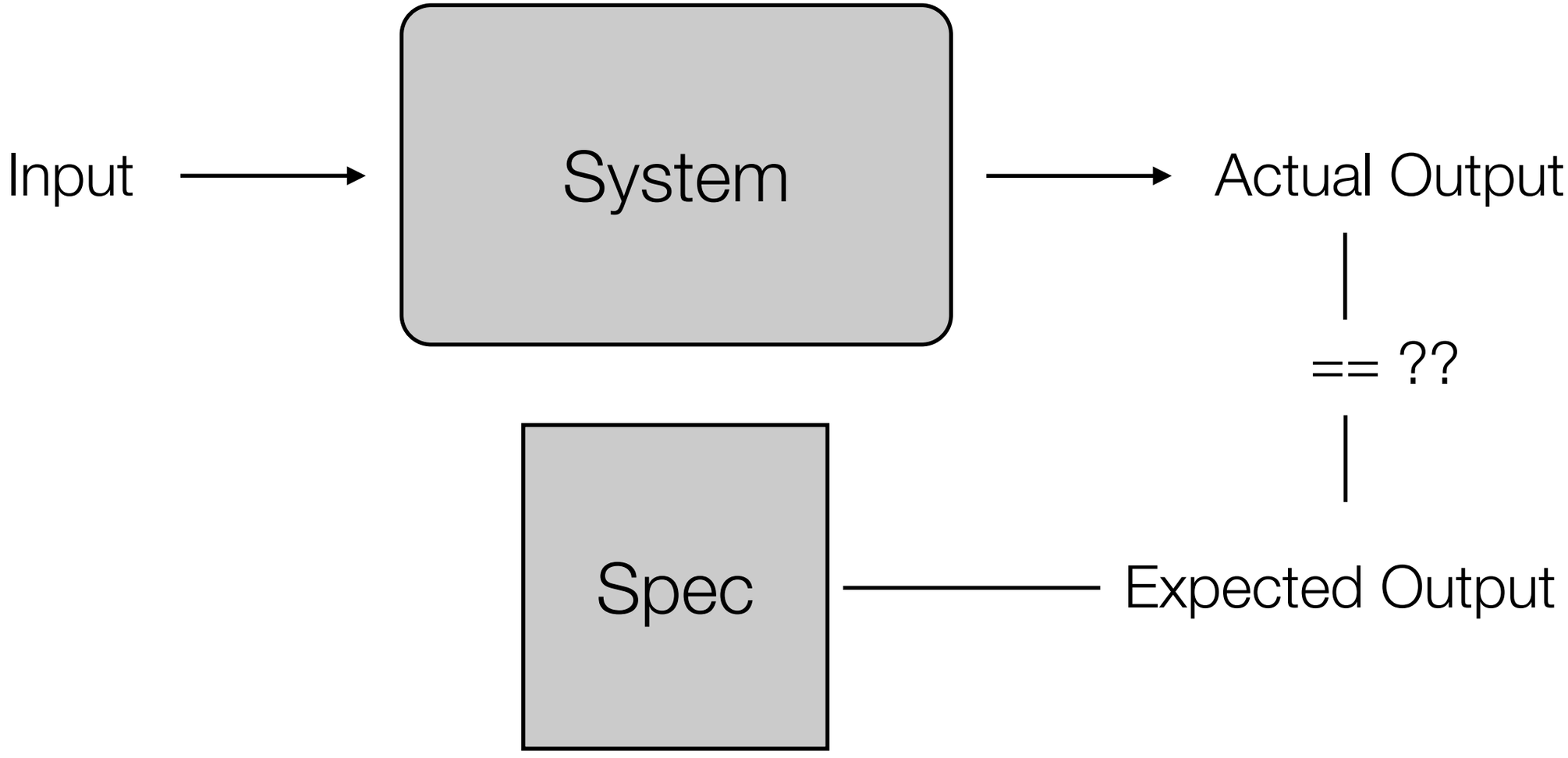
© Kenneth M. Anderson, 2012

Multi-Level Testing

- 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, …

© Kenneth M. Anderson, 2012

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!

© Kenneth M. Anderson, 2012

Results

- 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

© Kenneth M. Anderson, 2012

Black Box Categories

- 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

© Kenneth M. Anderson, 2012

Grey Box Testing

- 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

© Kenneth M. Anderson, 2012

White Box Testing

- 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?

© Kenneth M. Anderson, 2012

Code Coverage (I)

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

© Kenneth M. Anderson, 2012

Code Coverage (II)

- 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)

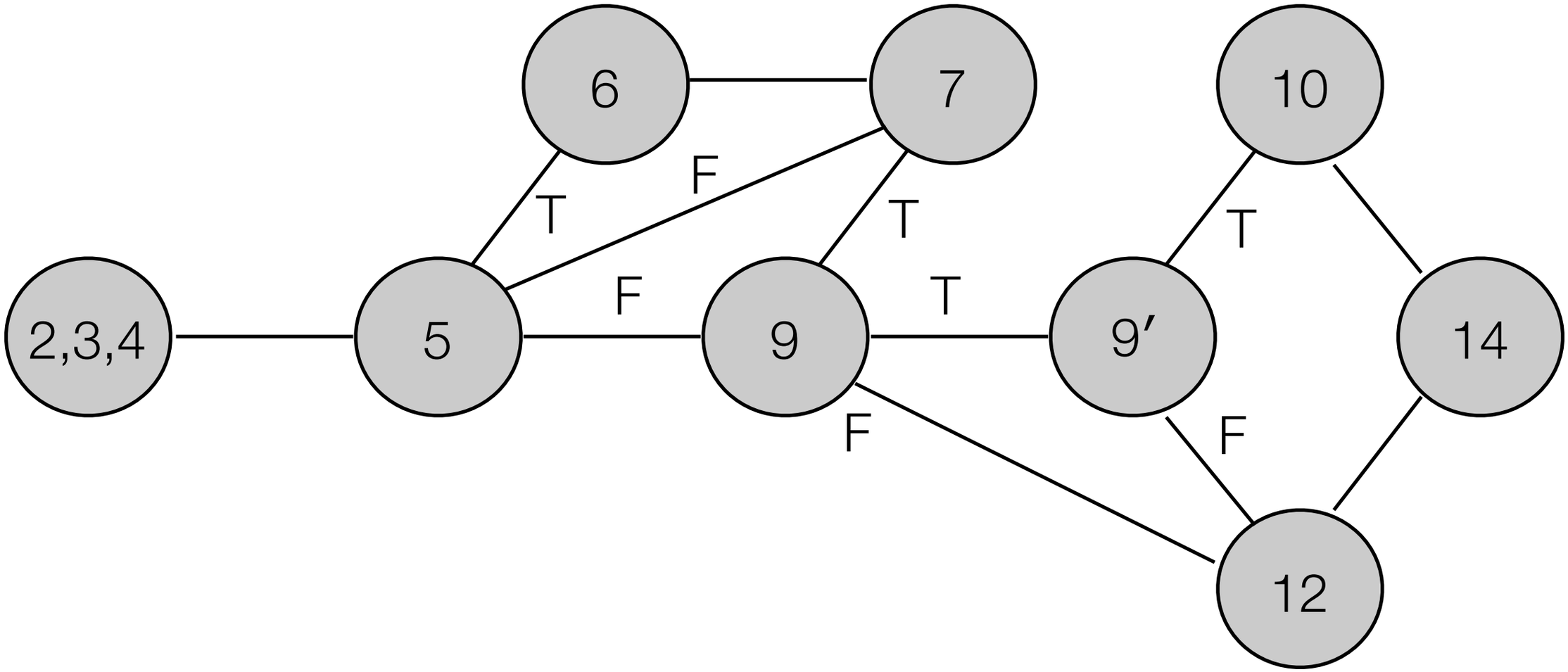
© Kenneth M. Anderson, 2012

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;
```

© Kenneth M. Anderson, 2012

P’s Control Flow Graph (CFG)



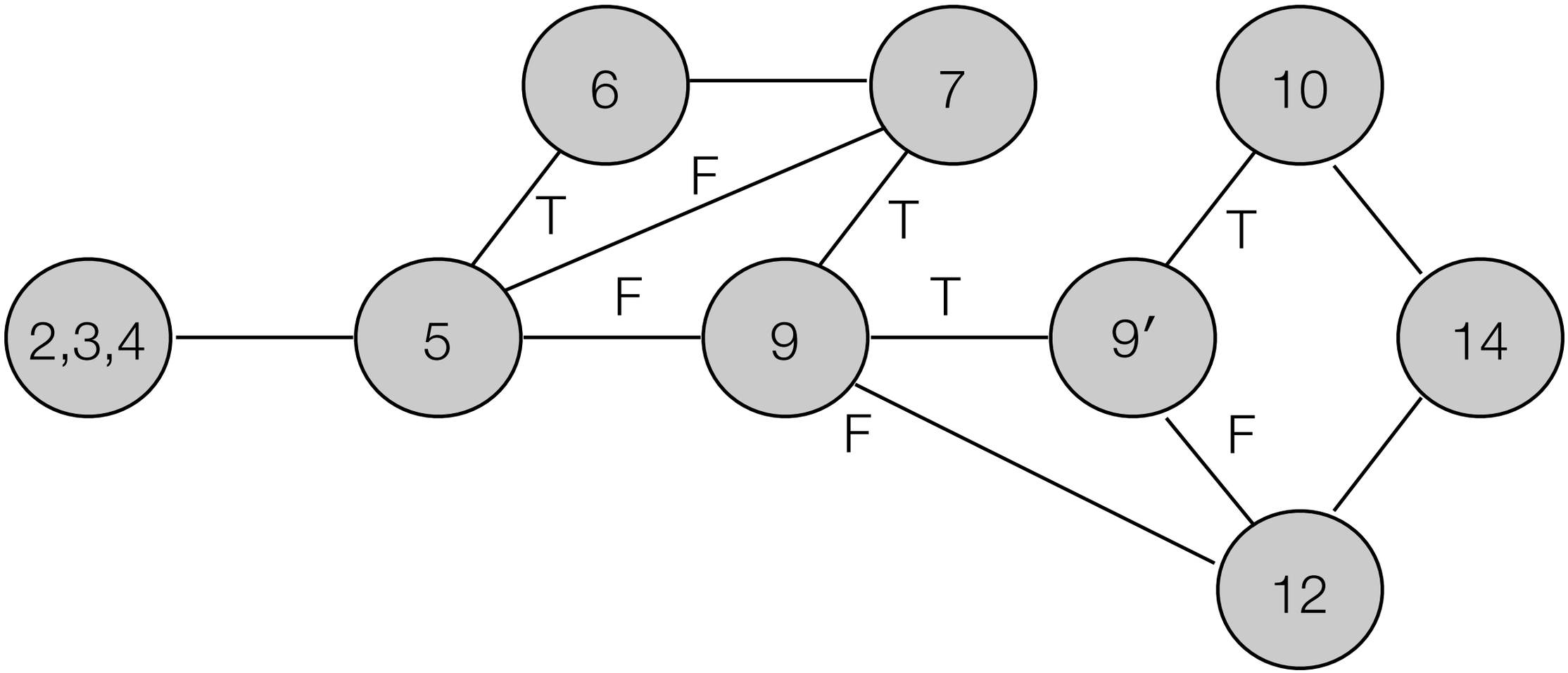
© Kenneth M. Anderson, 2012

White-box Testing Criteria

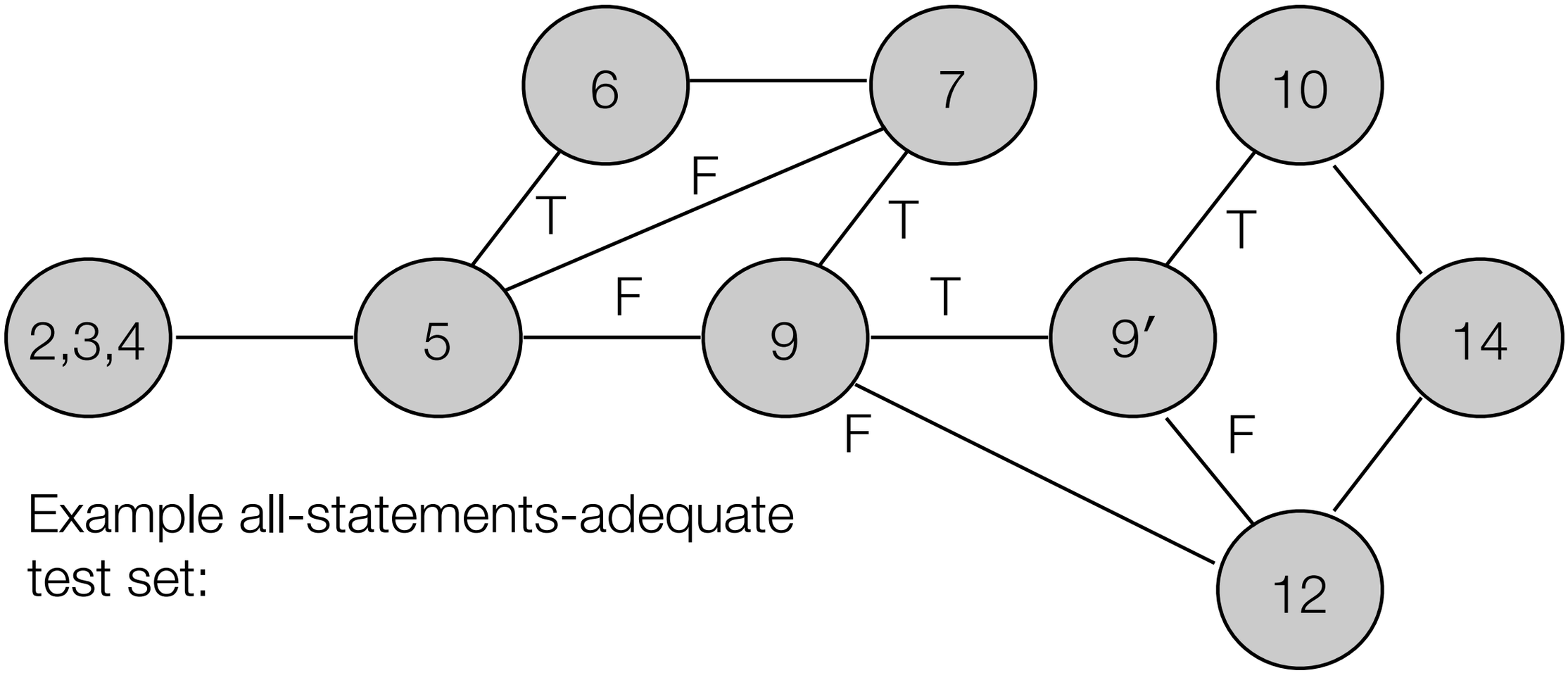
- Statement Coverage
  - Create a test set T such that
    - by executing P for each t in T
      - each elementary statement of P is executed at least once

© Kenneth M. Anderson, 2012

All-Statements Coverage of P

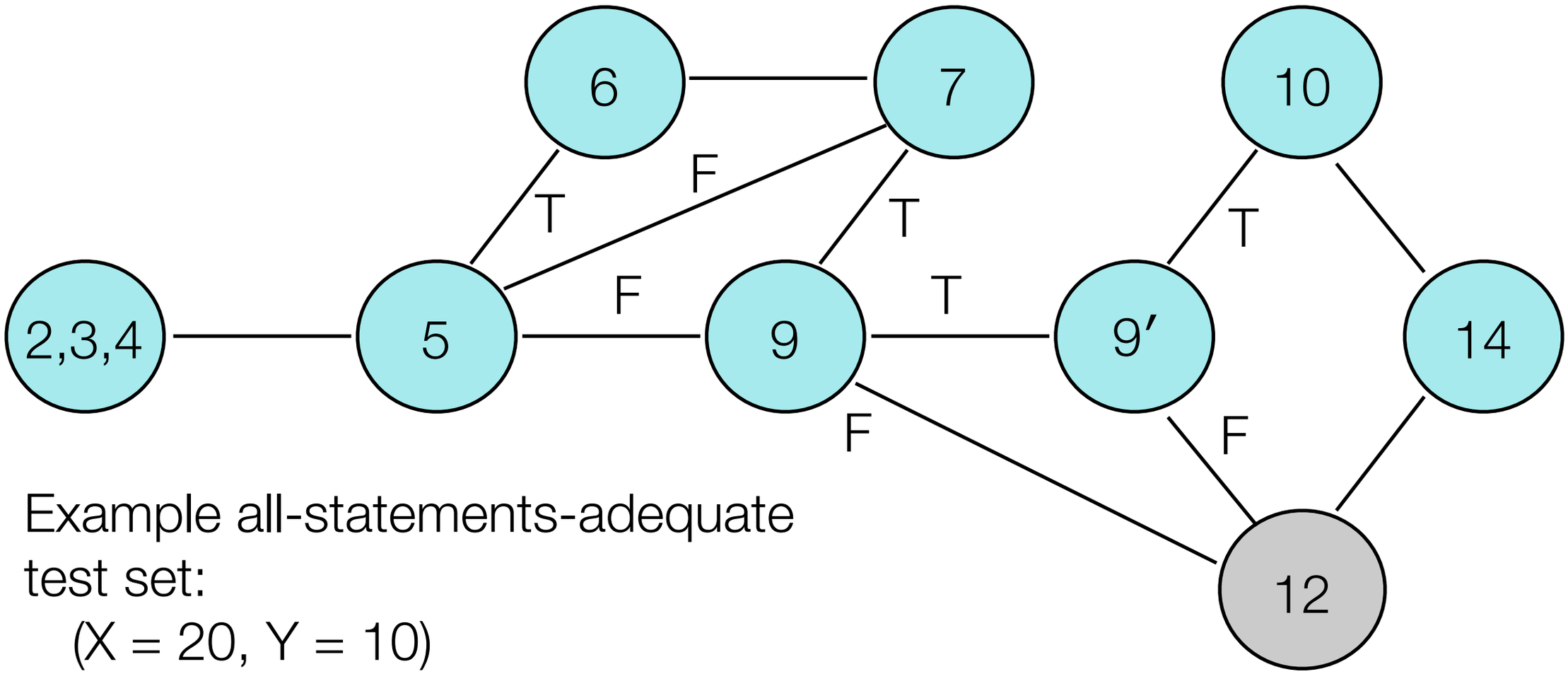


All-Statements Coverage of P



Example all-statements-adequate test set:

All-Statements Coverage of P

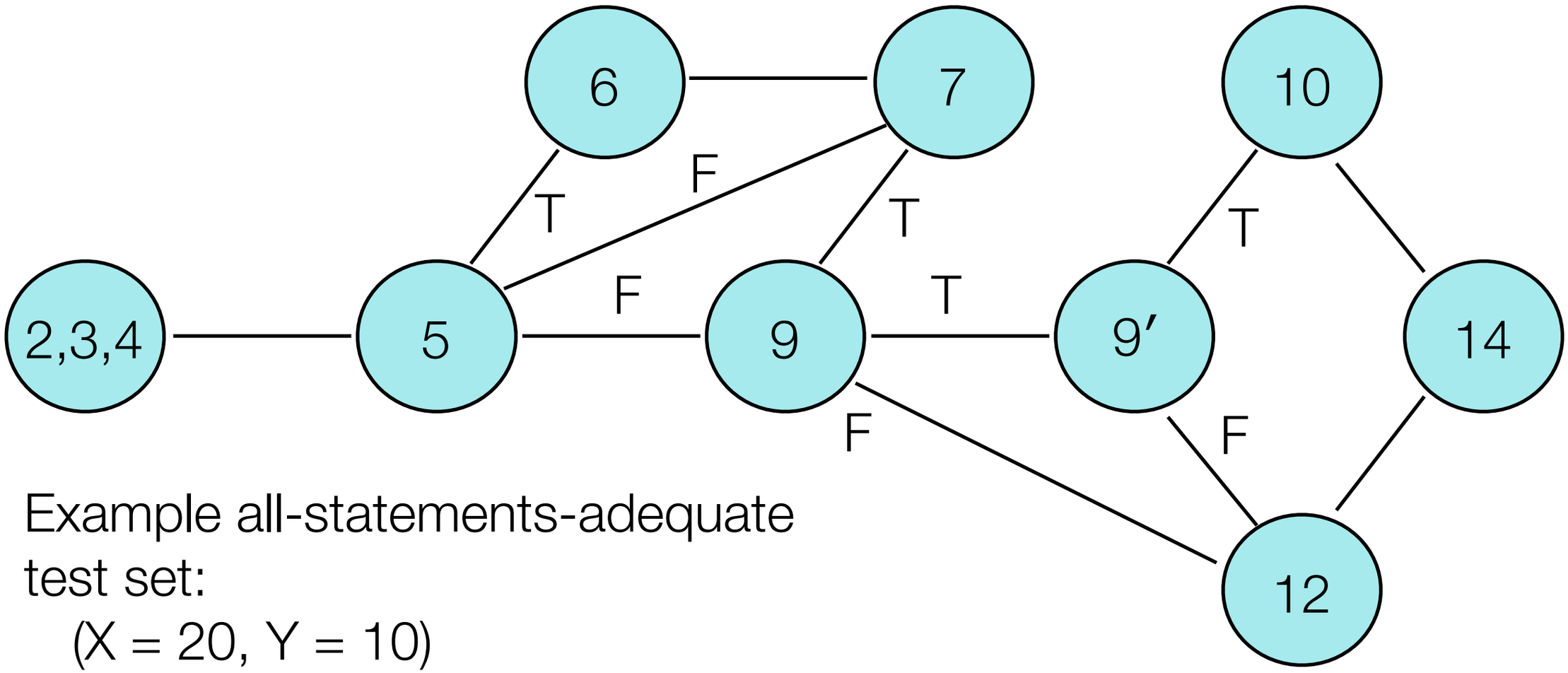


Example all-statements-adequate test set:

(X = 20, Y = 10)

© Kenneth M. Anderson, 2012

All-Statements Coverage of P



Example all-statements-adequate test set:

(X = 20, Y = 10)

(X = 20, Y = 30)

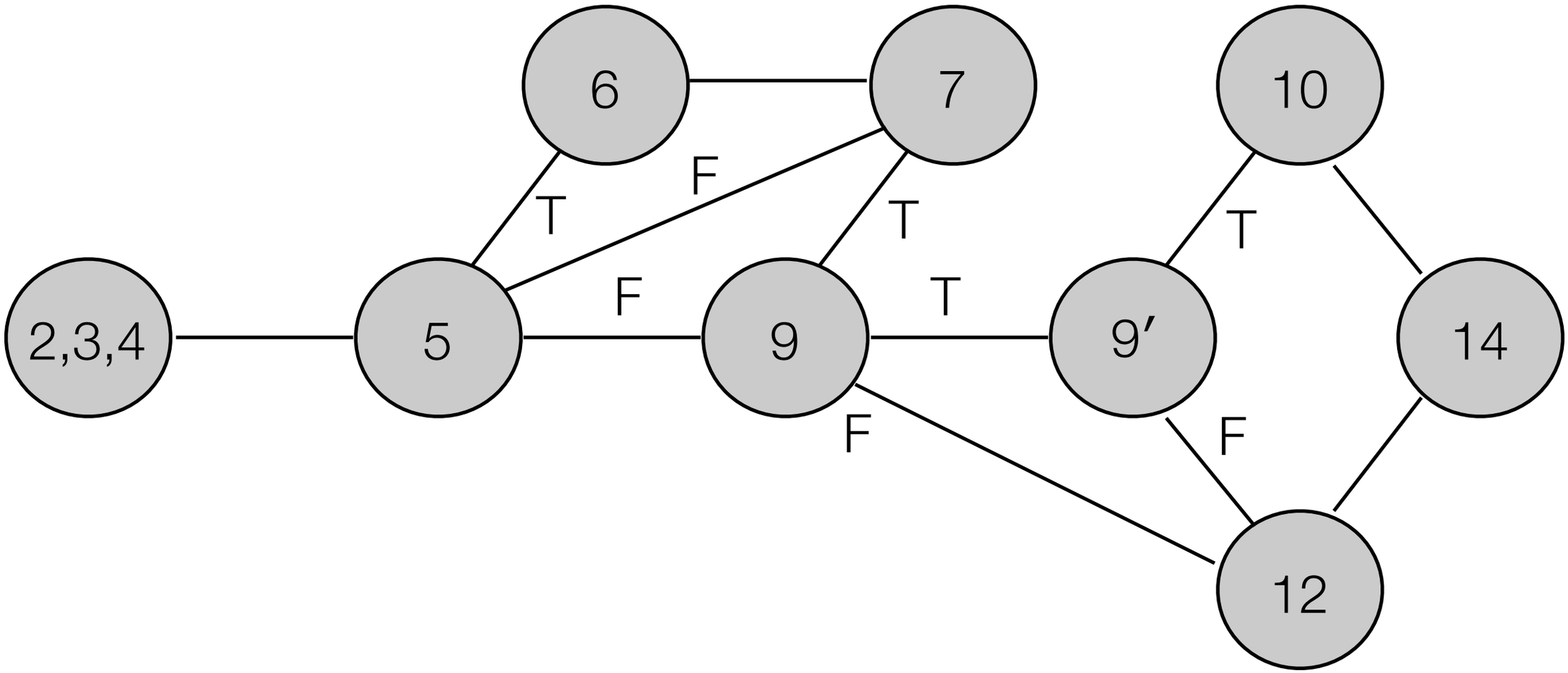
© Kenneth M. Anderson, 2012

White-box Testing Criteria

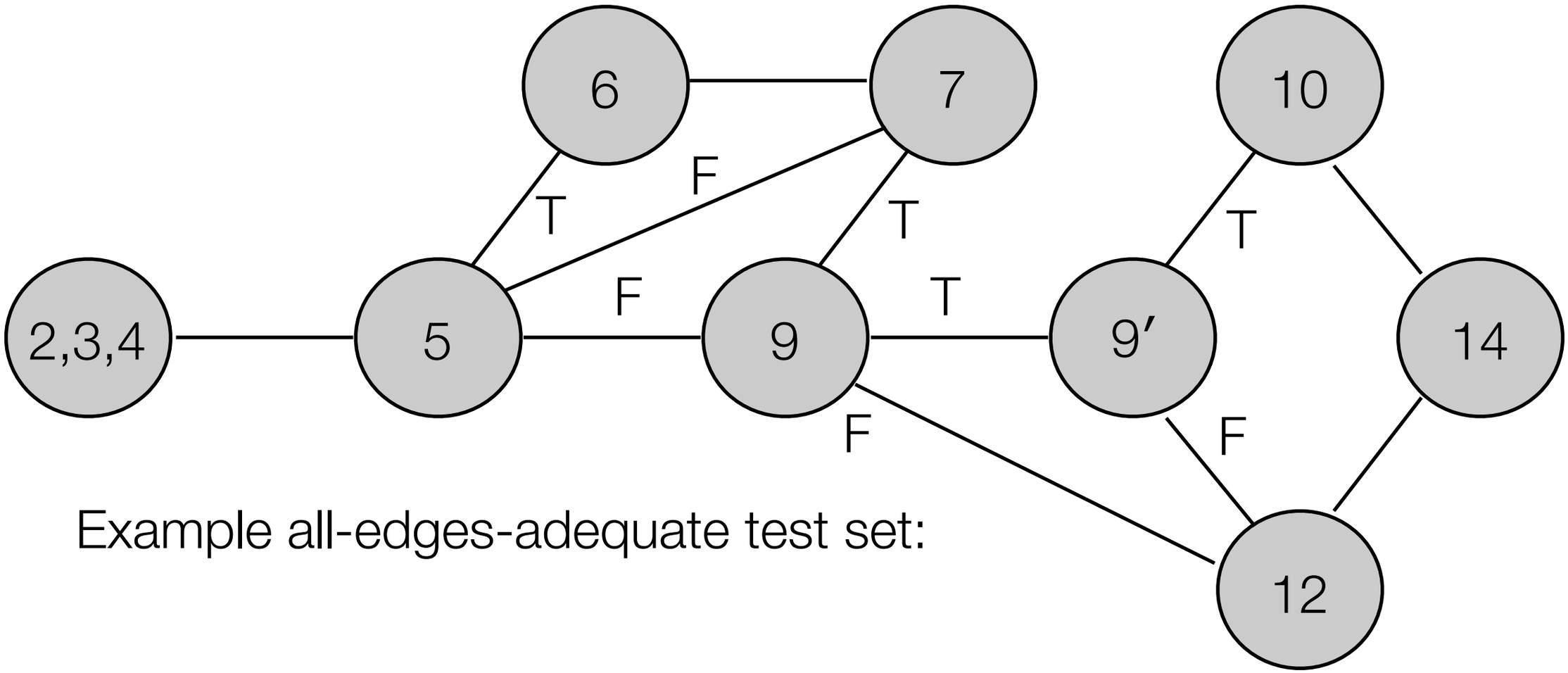
- Edge Coverage
  - Select a test set T such that
    - by executing P for each t in T
      - each edge of P's control flow graph is traversed at least once

© Kenneth M. Anderson, 2012

All-Edges Coverage of P



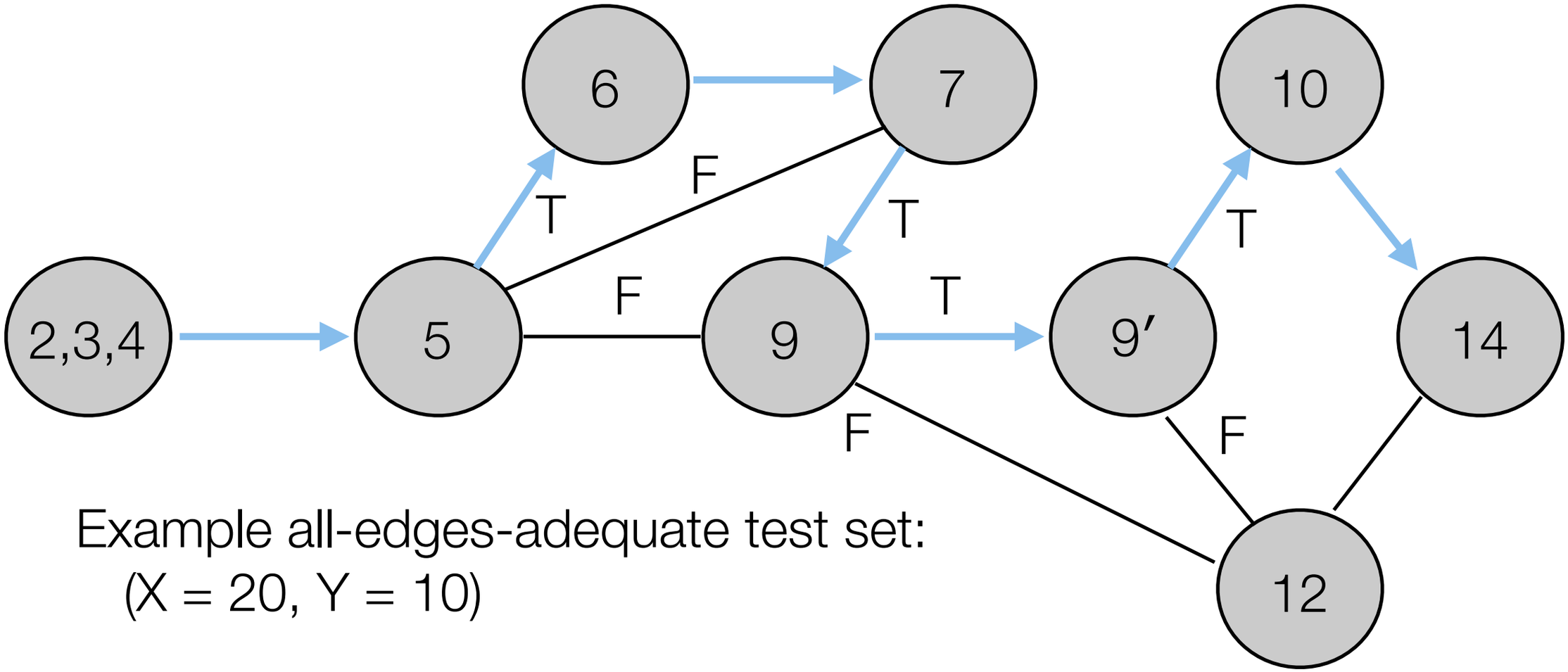
All-Edges Coverage of P



Example all-edges-adequate test set:

© Kenneth M. Anderson, 2012

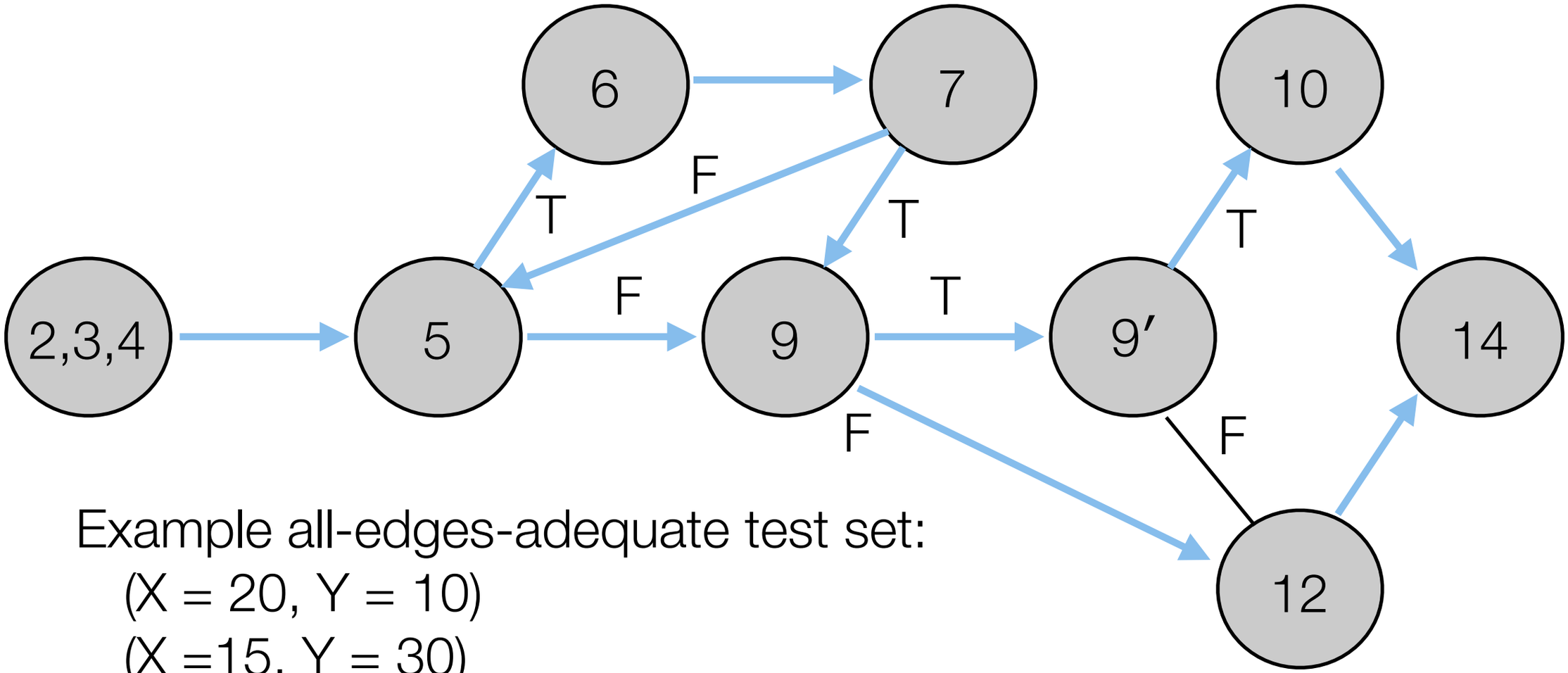
All-Edges Coverage of P



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

© Kenneth M. Anderson, 2012

All-Edges Coverage of P



Example all-edges-adequate test set:

(X = 20, Y = 10)

(X =15, Y = 30)

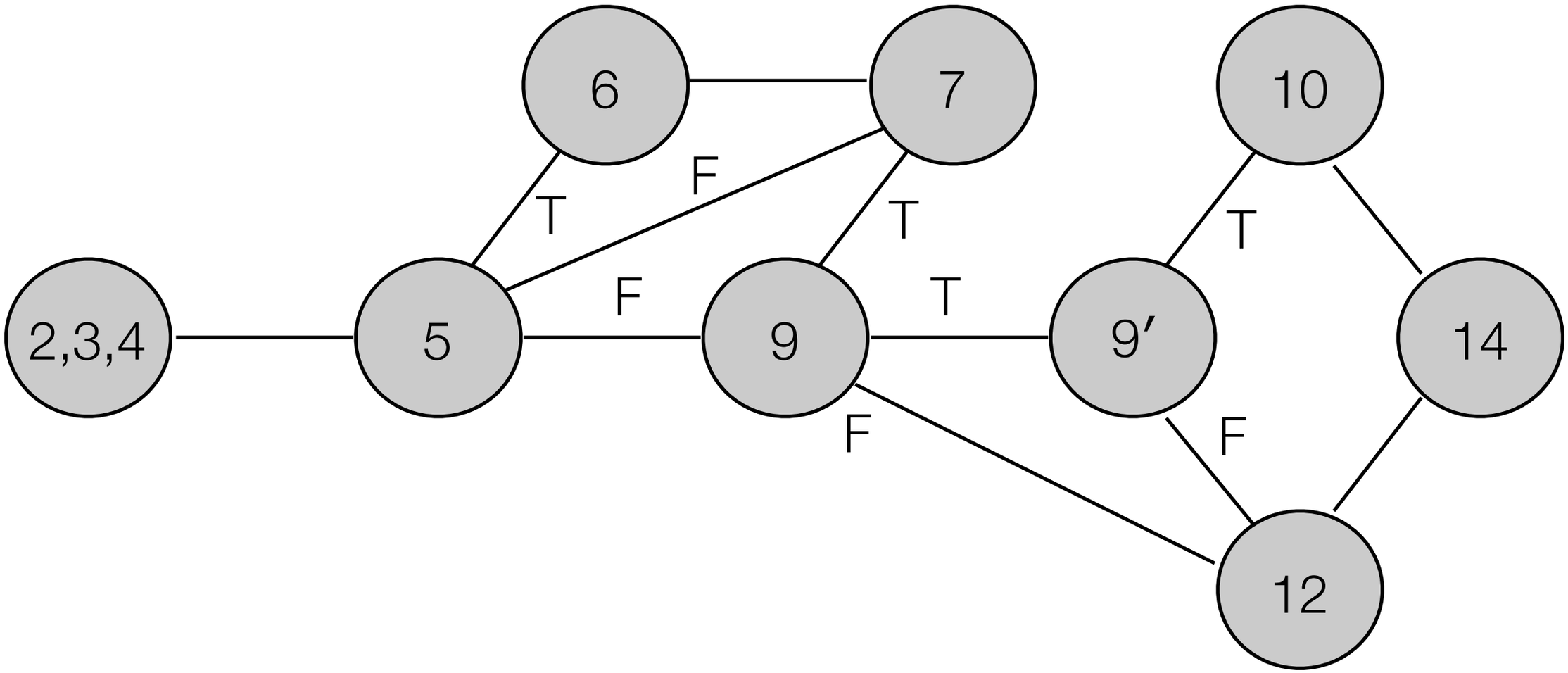
© Kenneth M. Anderson, 2012

White-box Testing Criteria

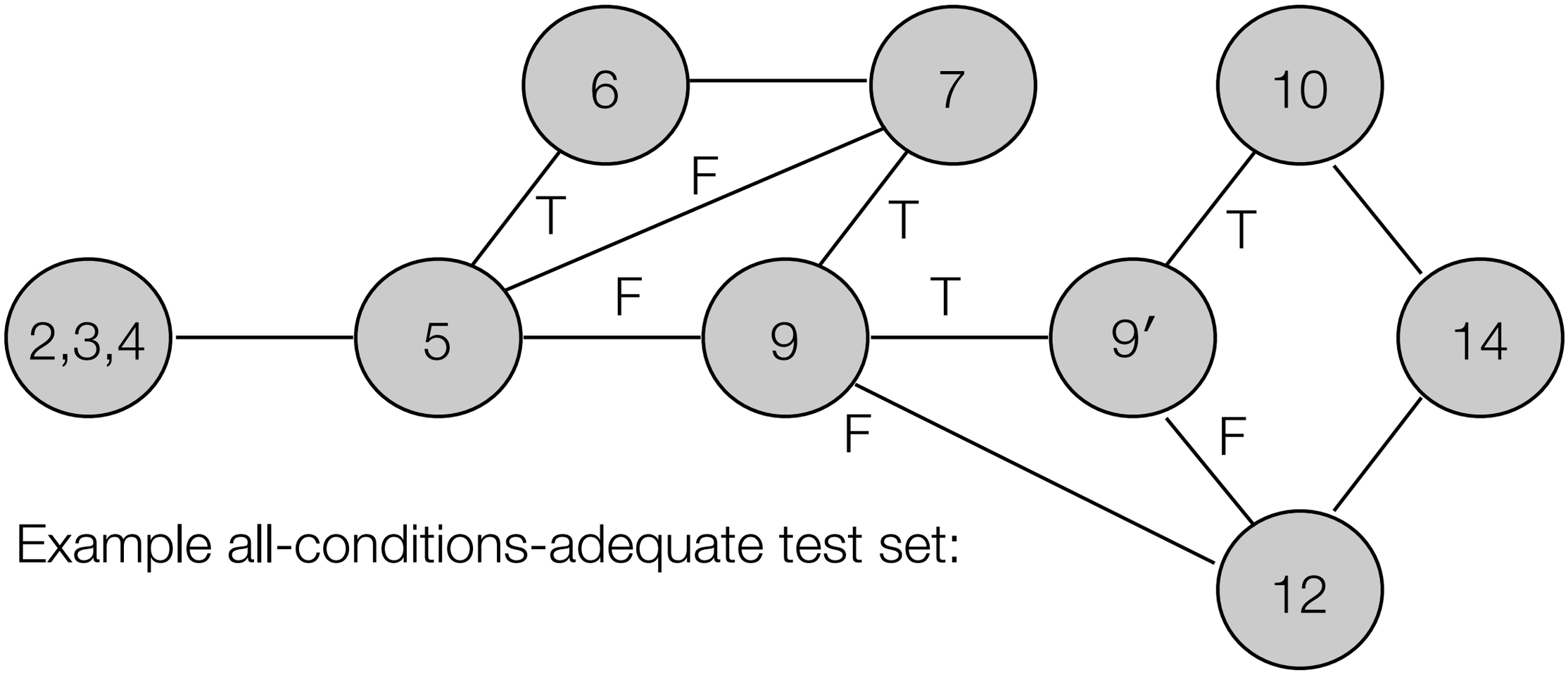
- Condition Coverage
  - Select a test set T such that
    - by executing P for each t in T
      - each edge of P's control flow graph is traversed at least once
      - and all possible values of the constituents of compound conditions are exercised at least once

© Kenneth M. Anderson, 2012

All-Conditions Coverage of P



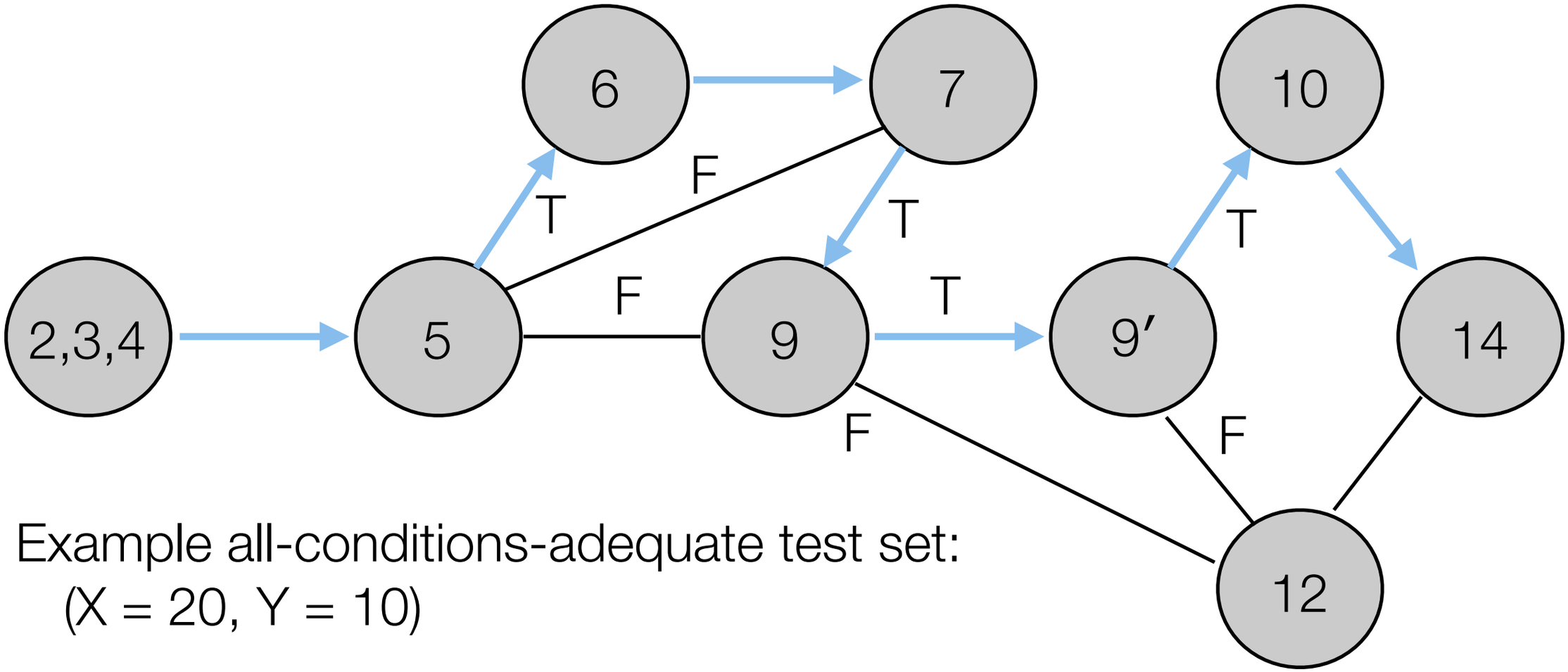
All-Conditions Coverage of P



Example all-conditions-adequate test set:

© Kenneth M. Anderson, 2012

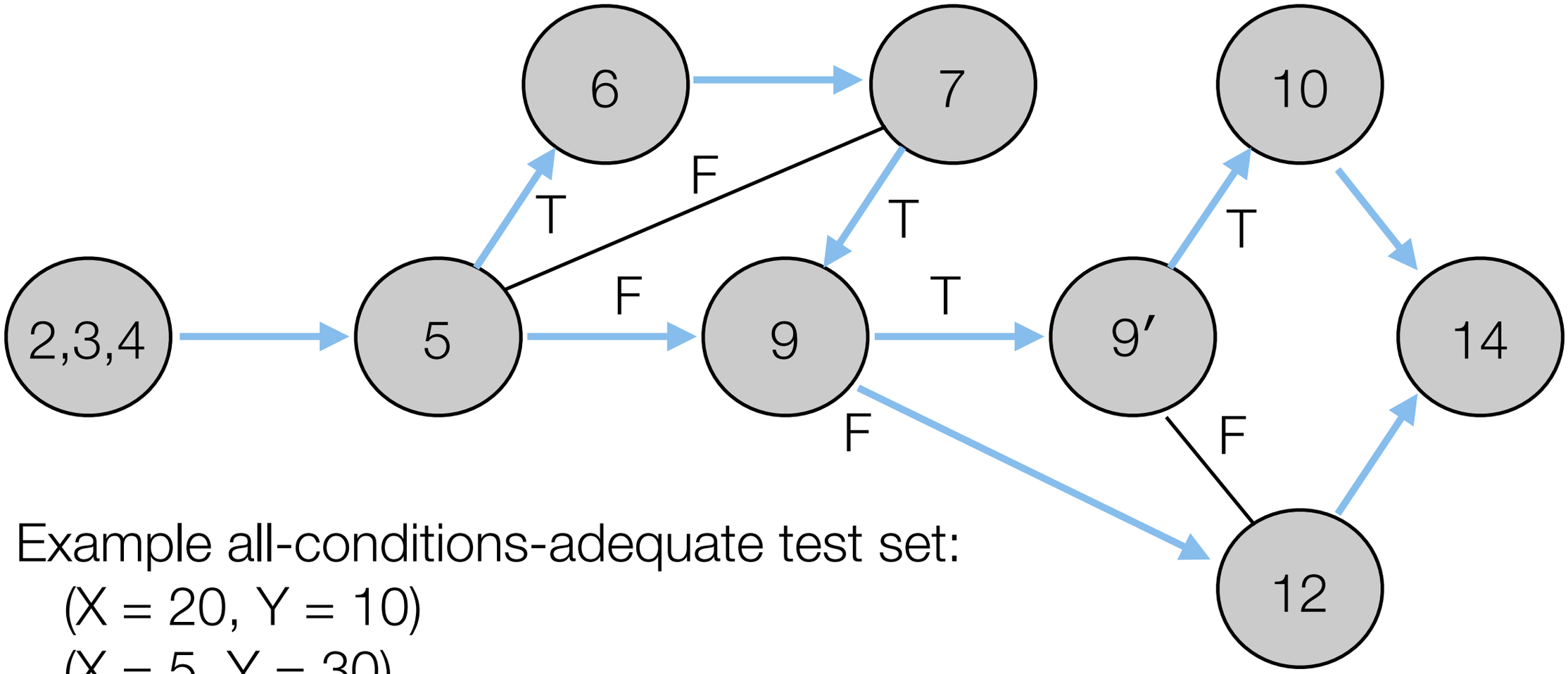
All-Conditions Coverage of P



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

© Kenneth M. Anderson, 2012

All-Conditions Coverage of P



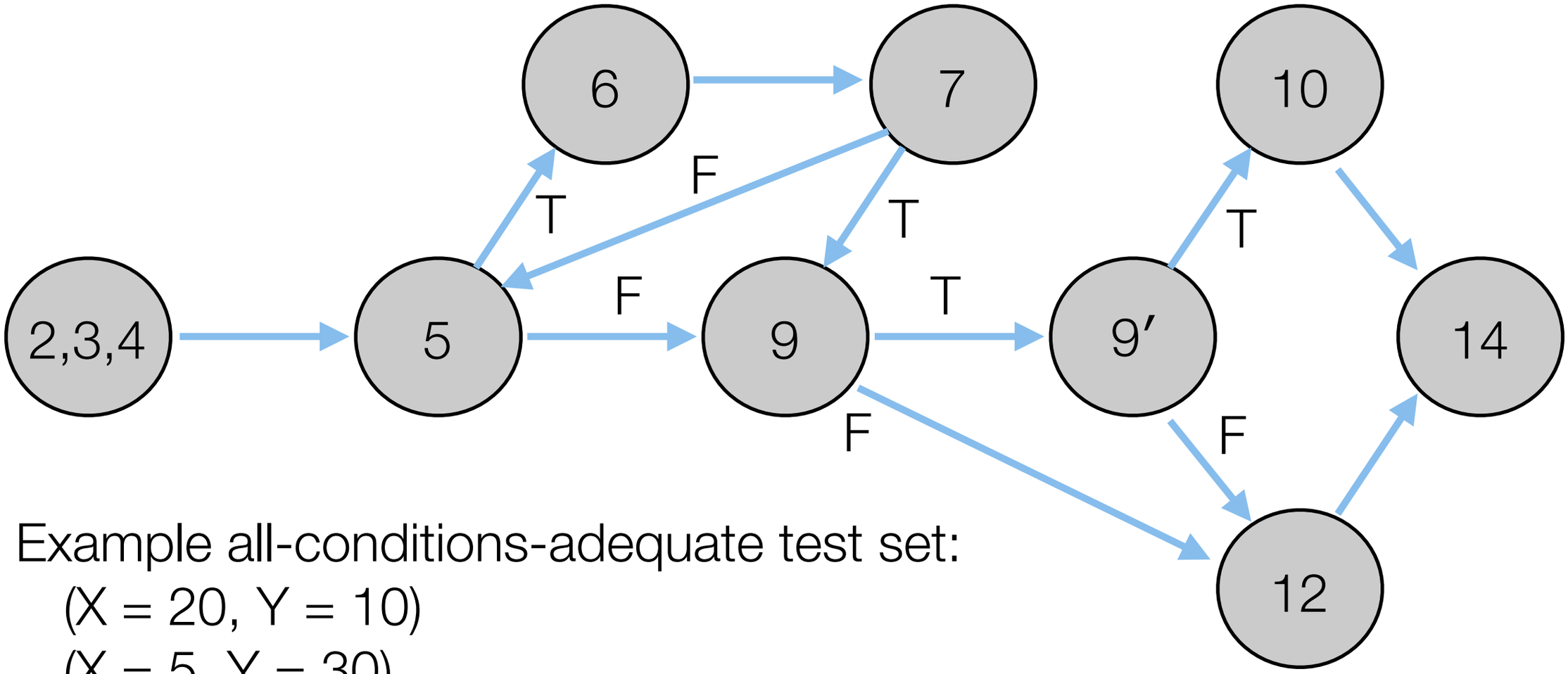
Example all-conditions-adequate test set:

(X = 20, Y = 10)

(X = 5, Y = 30)

© Kenneth M. Anderson, 2012

All-Conditions Coverage of P



Example all-conditions-adequate test set:

(X = 20, Y = 10)

(X = 5, Y = 30)

(X = 21, Y = 10)

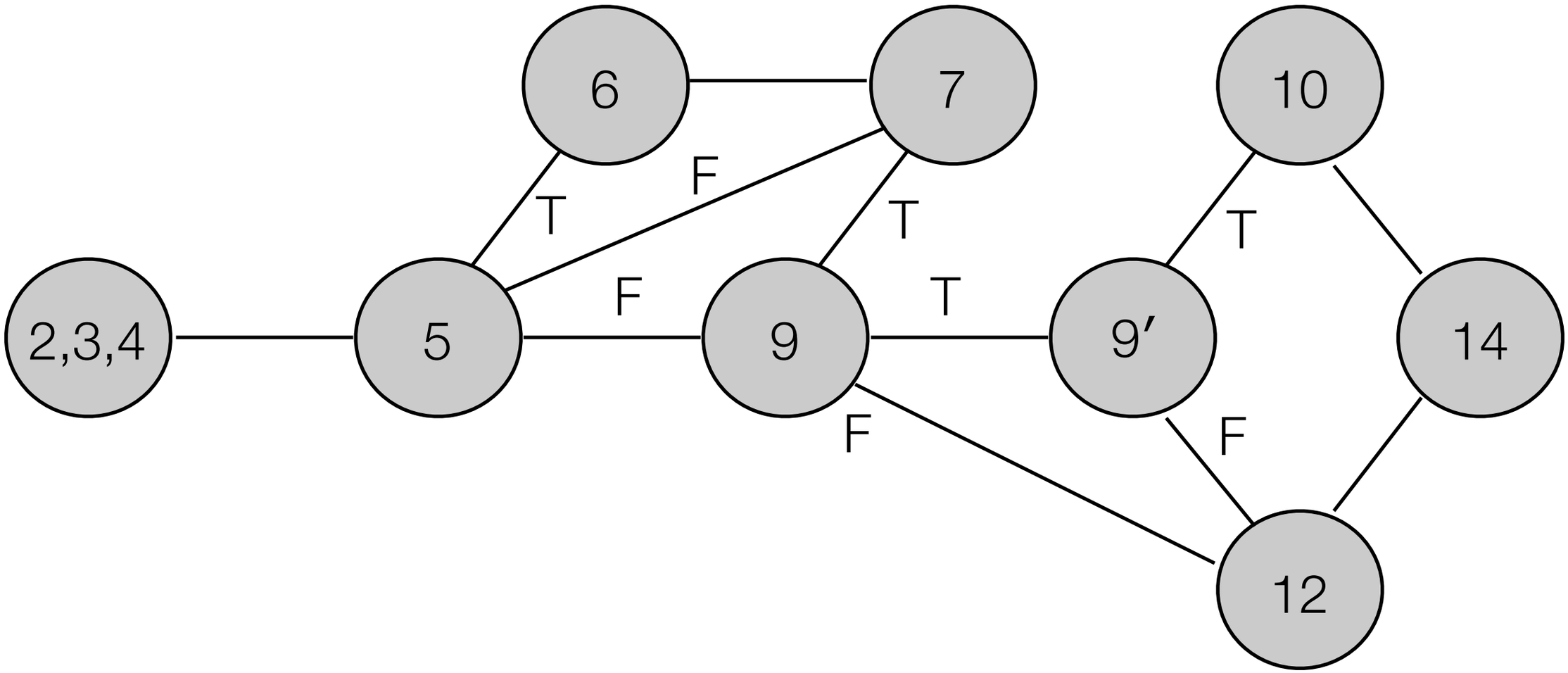
© Kenneth M. Anderson, 2012

White-box Testing Criteria

- 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

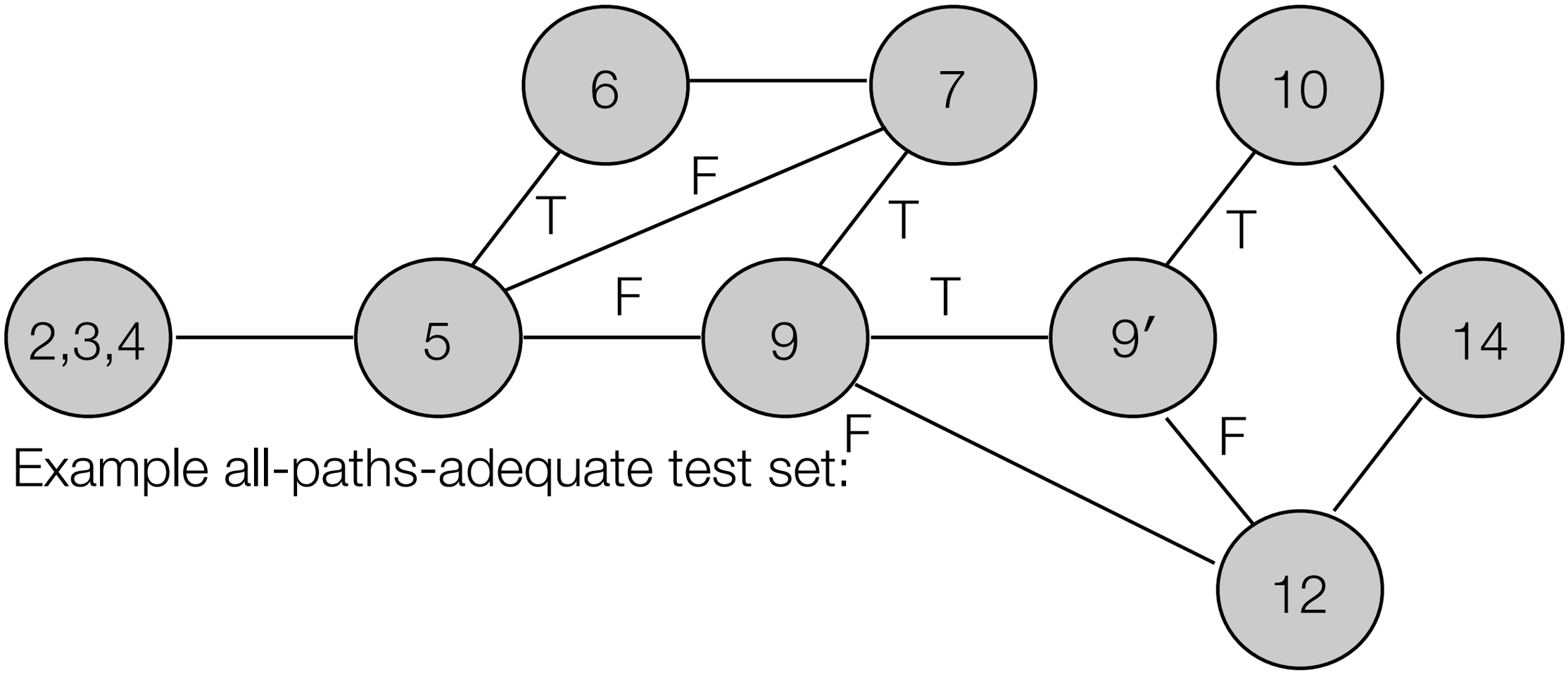
© Kenneth M. Anderson, 2012

All-Paths Coverage of P



© Kenneth M. Anderson, 2012

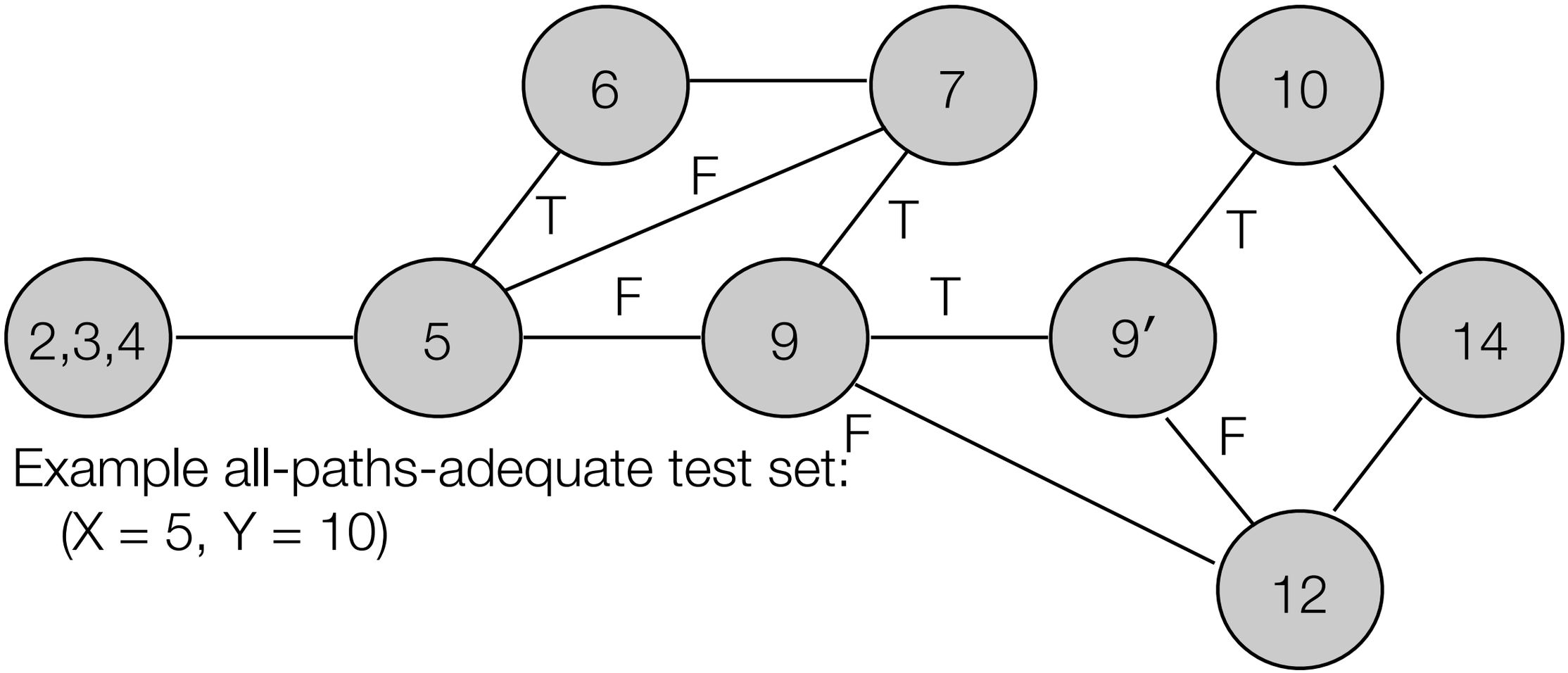
All-Paths Coverage of P



Example all-paths-adequate test set:

© Kenneth M. Anderson, 2012

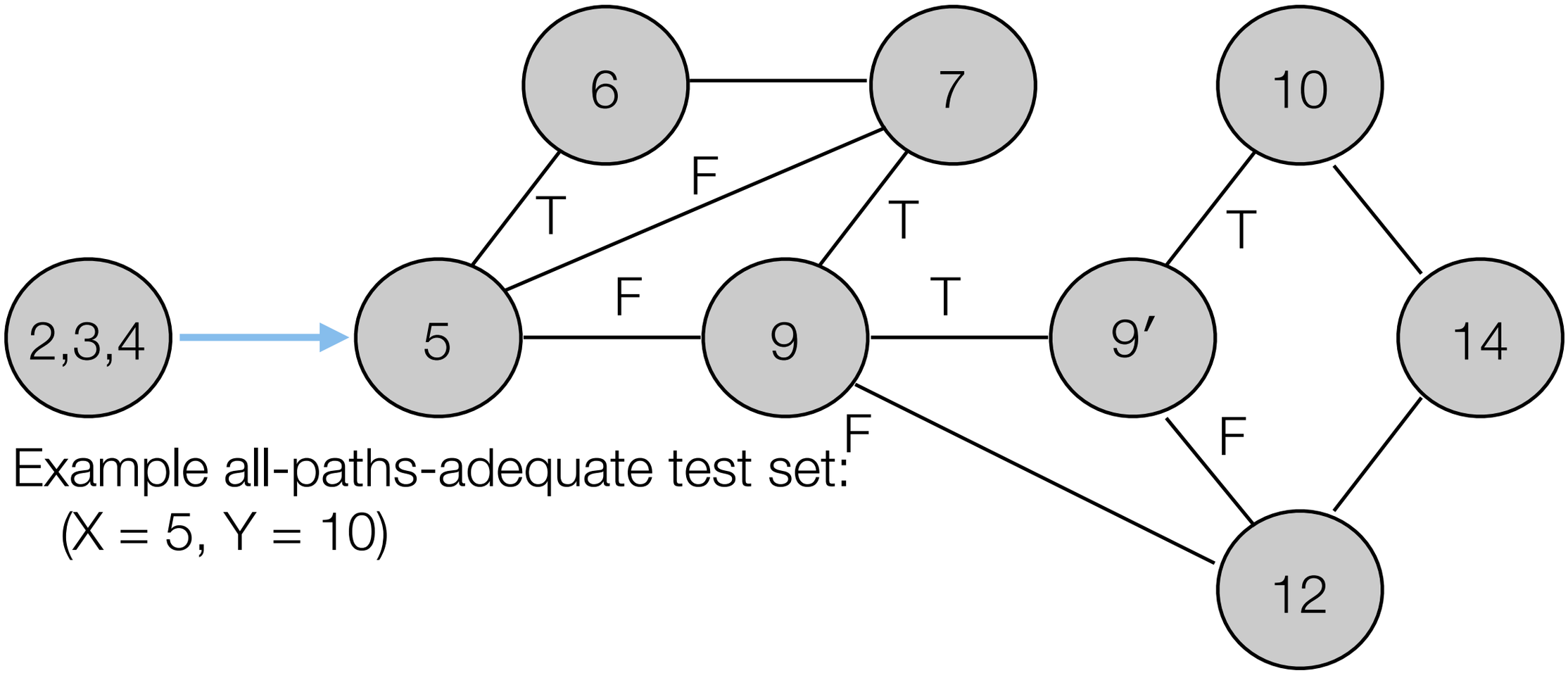
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

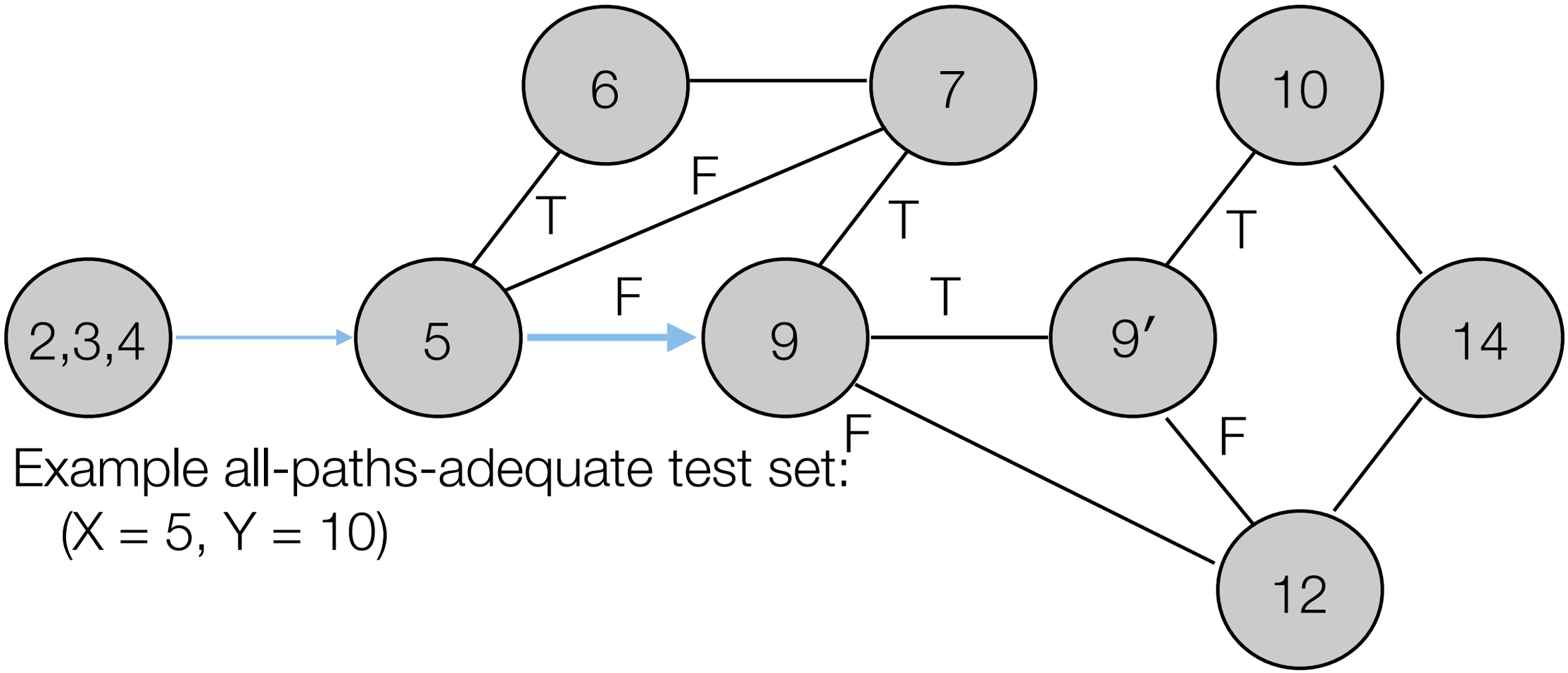
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

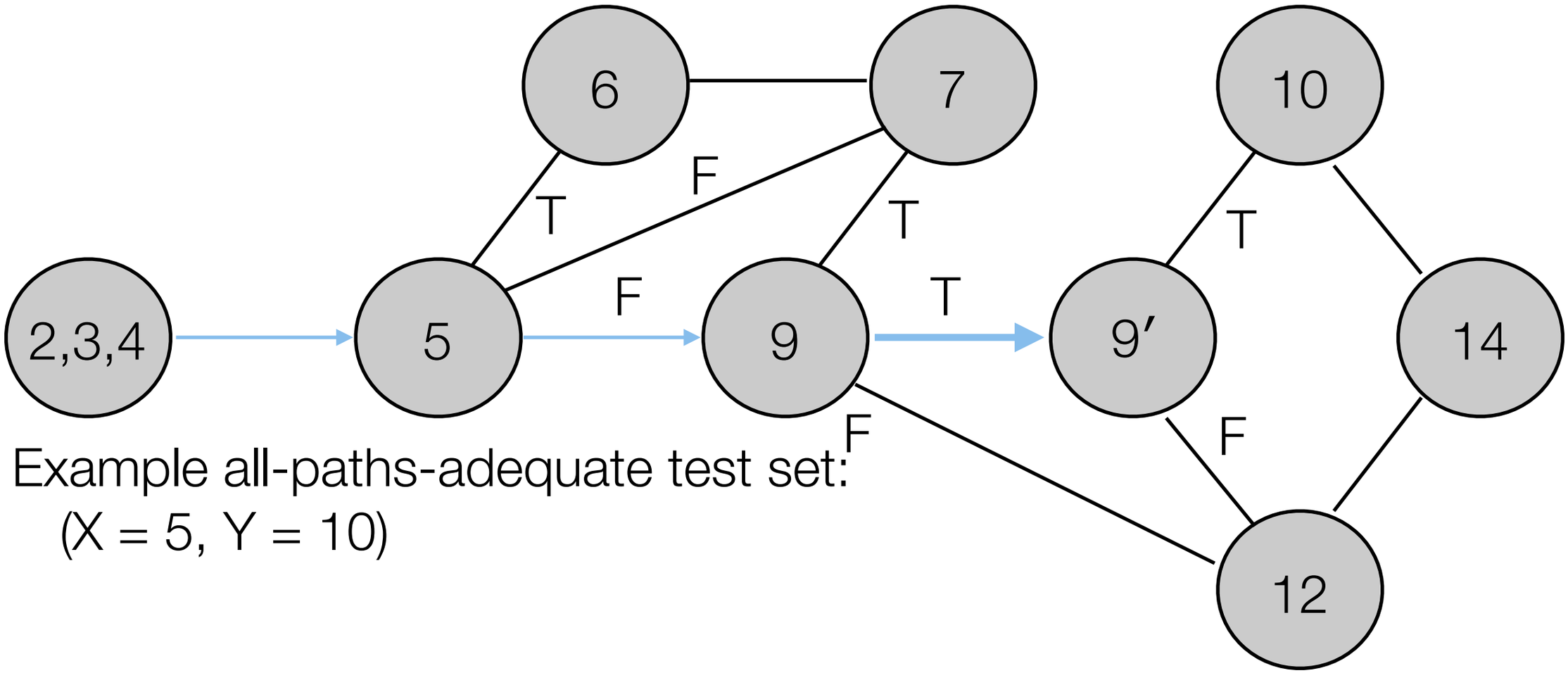
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

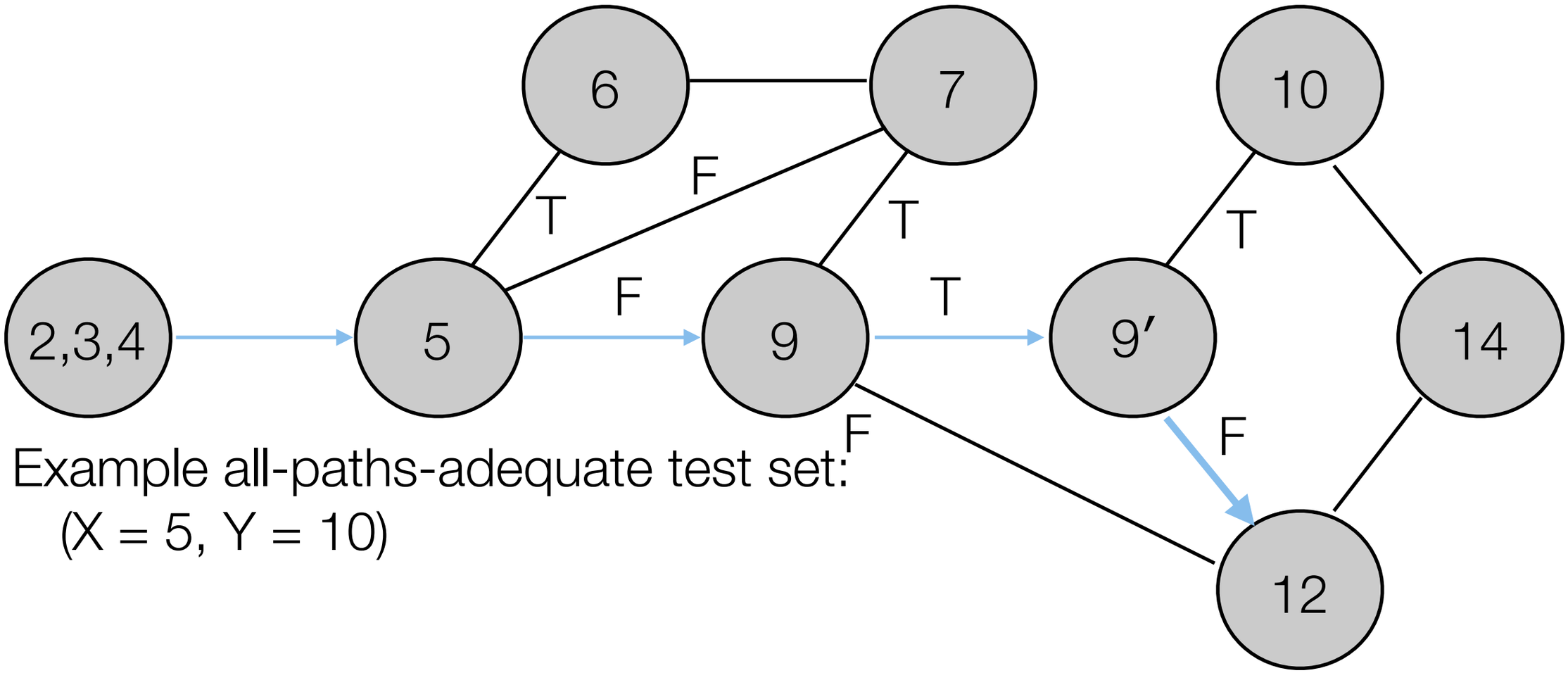
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

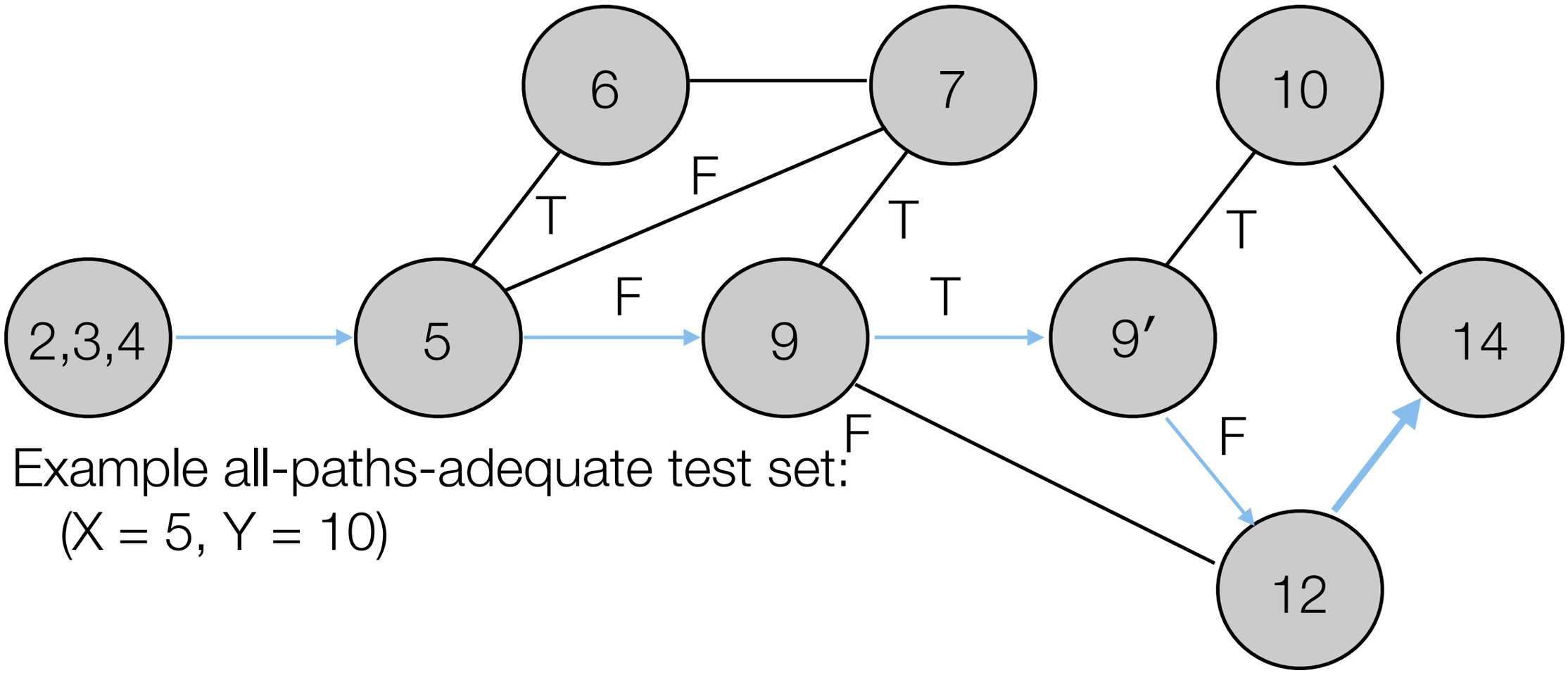
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

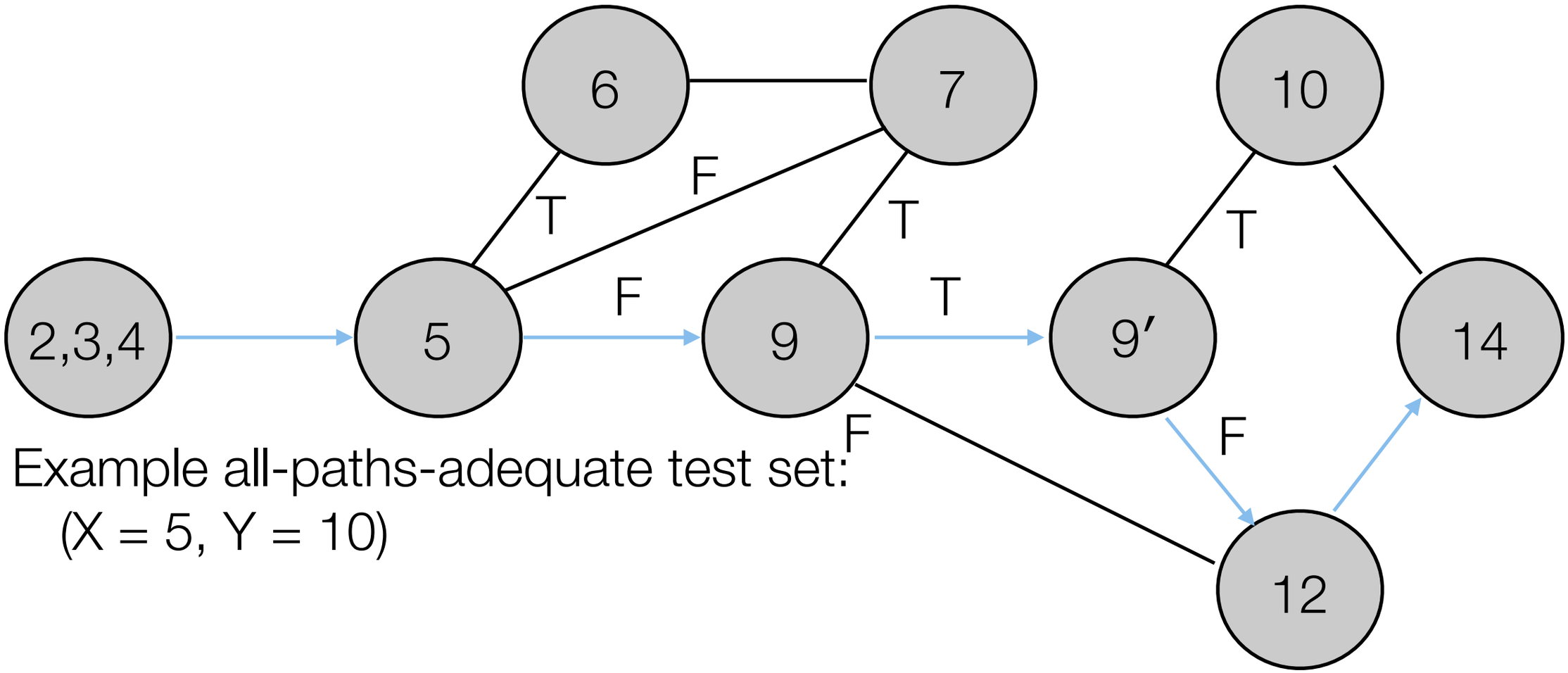
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

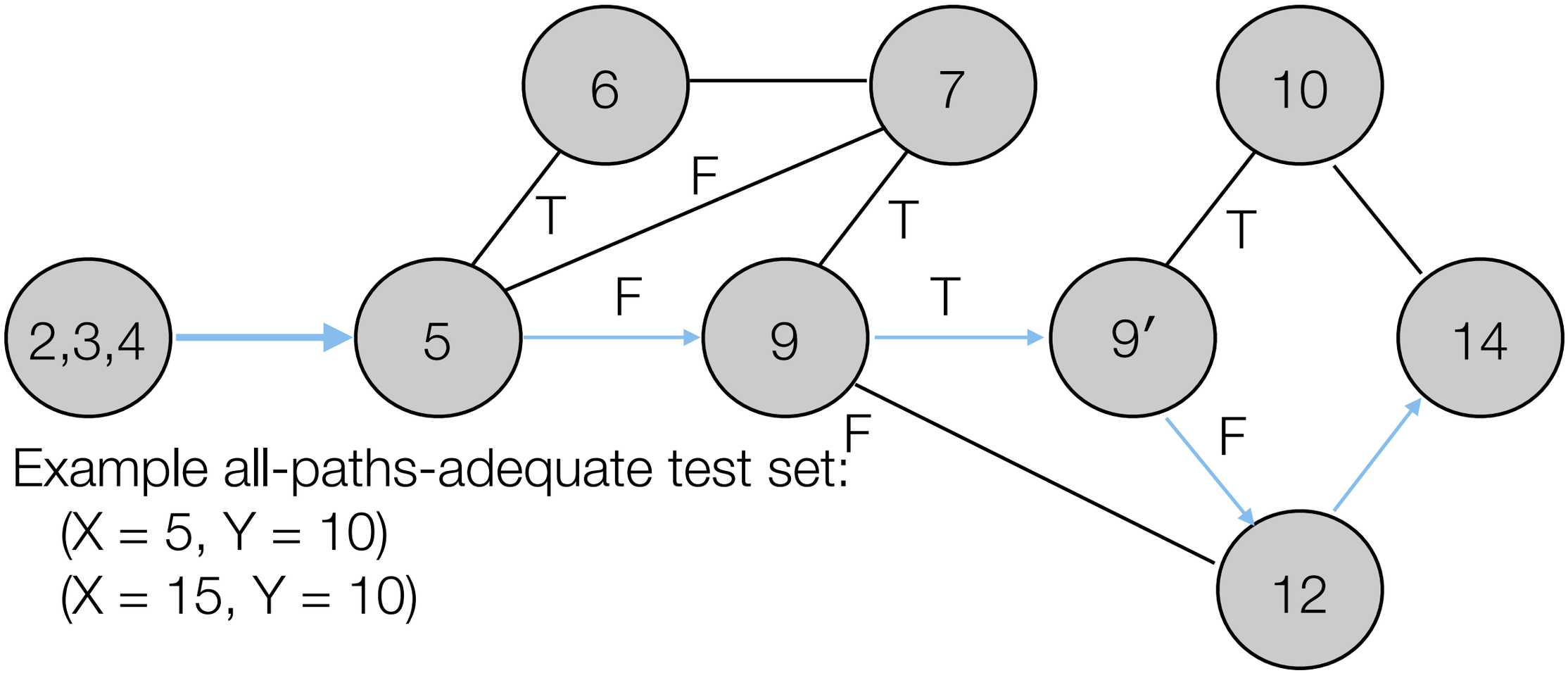
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



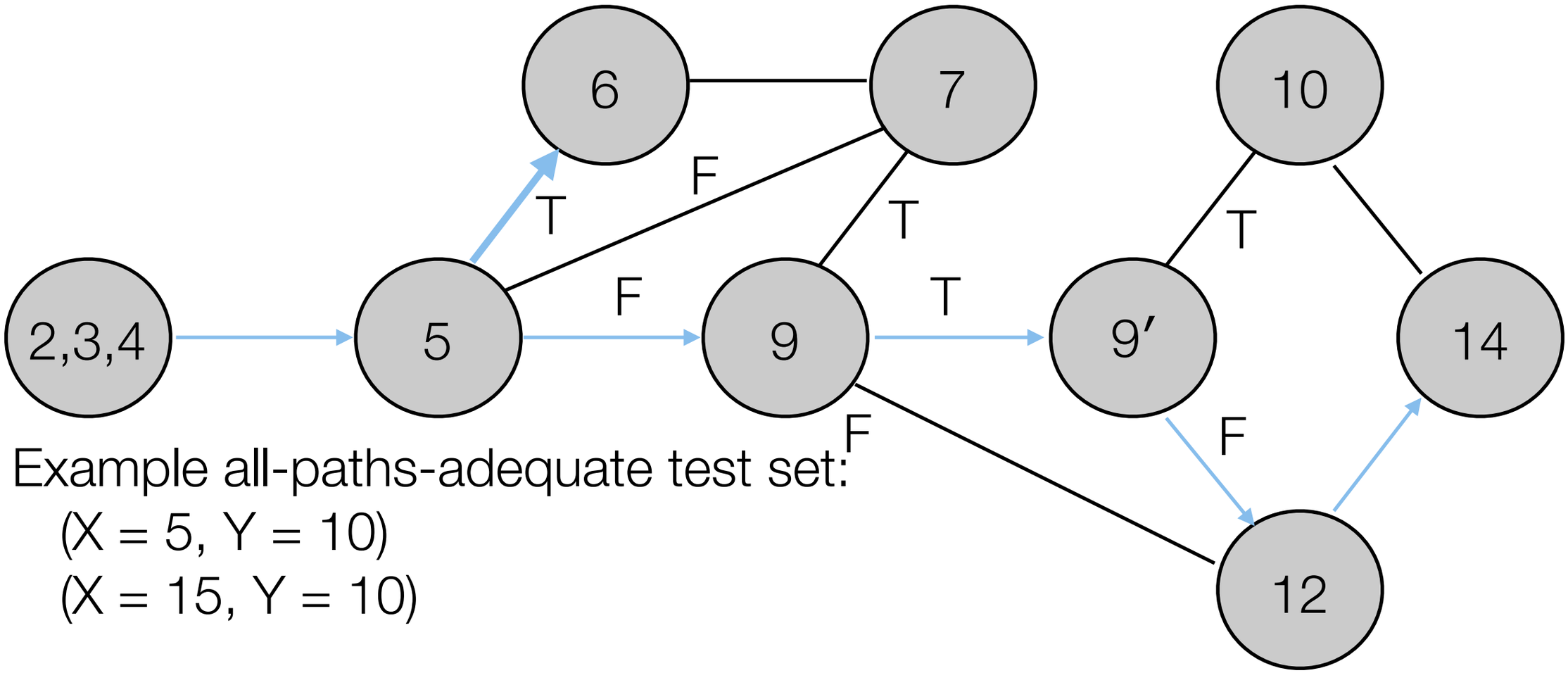
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



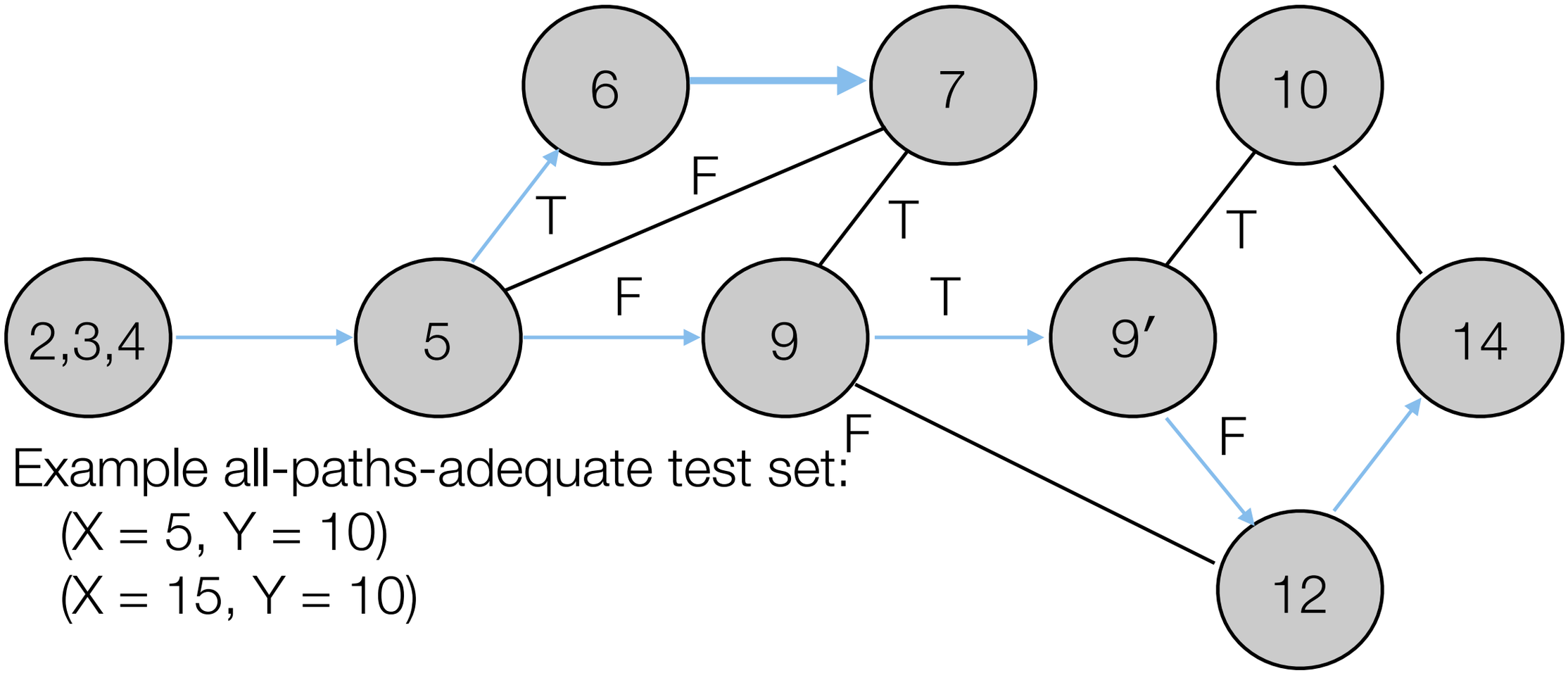
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



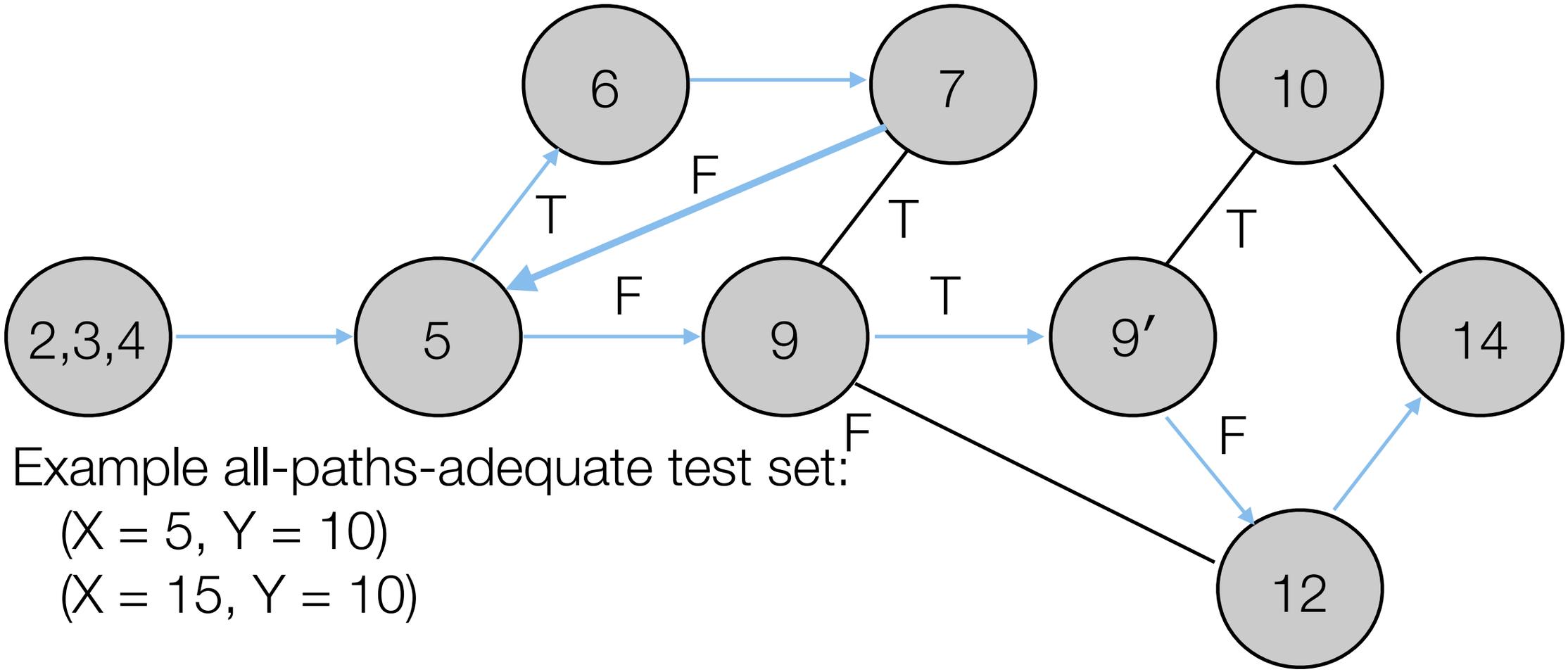
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



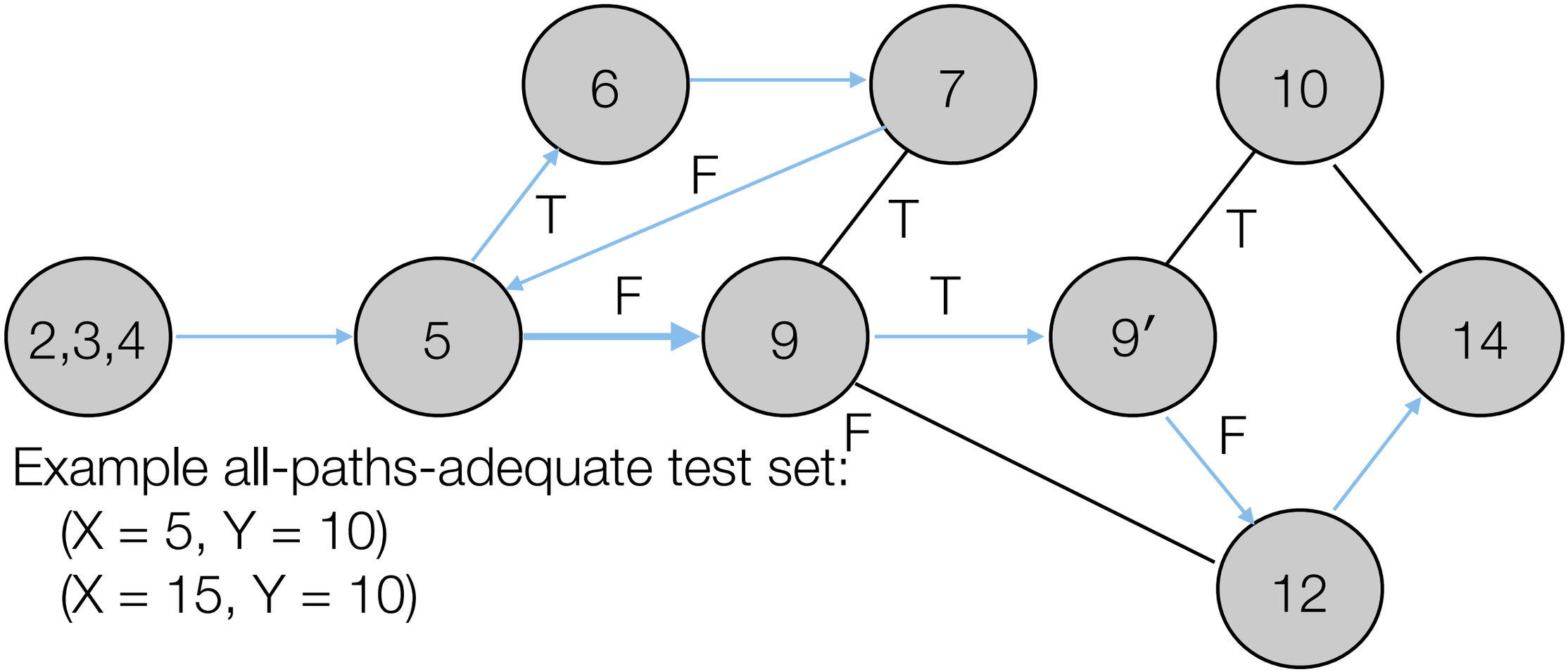
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



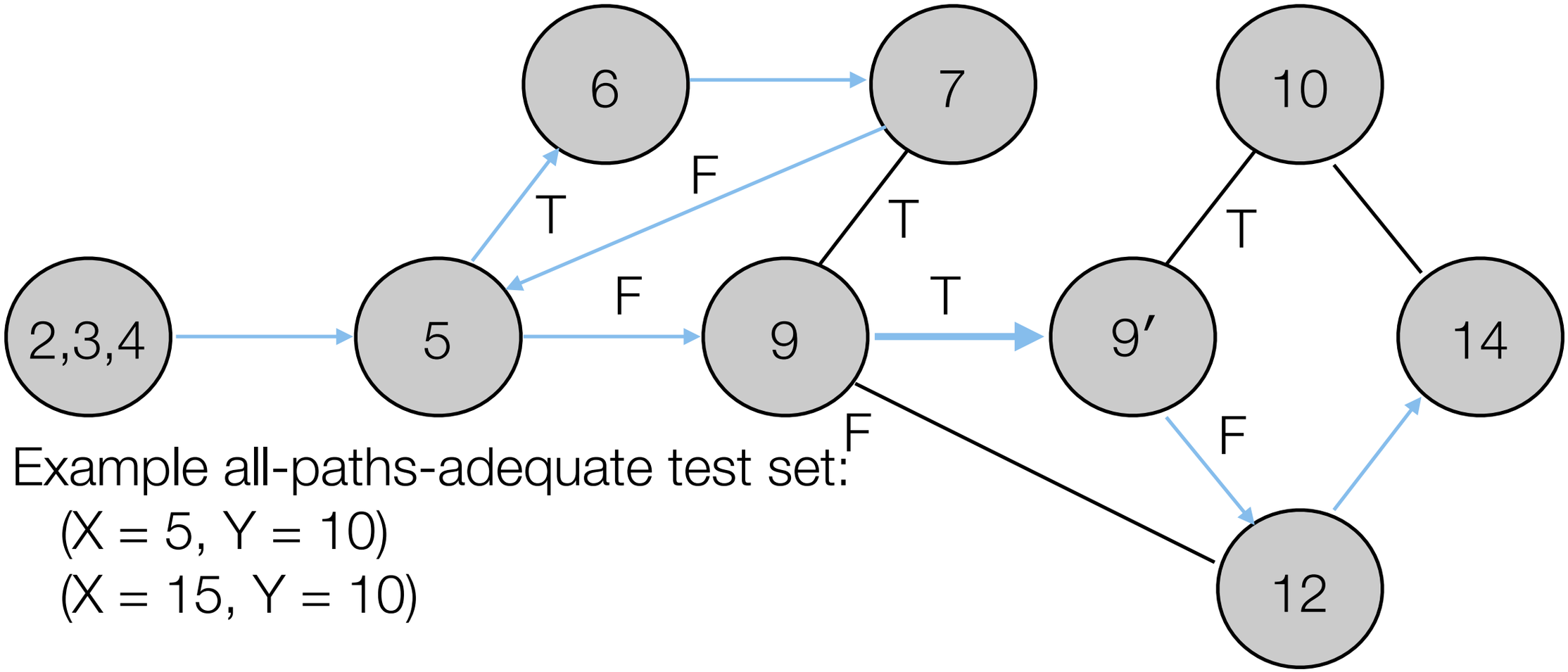
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



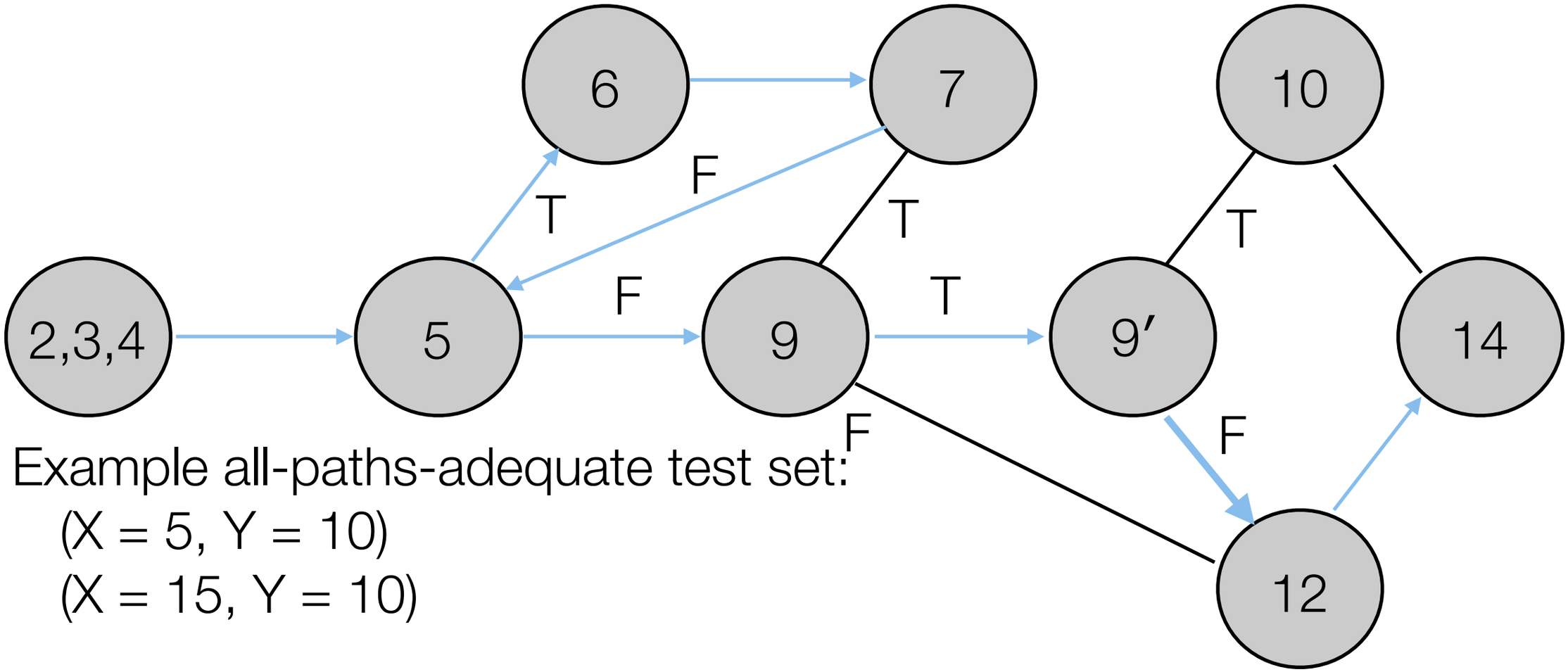
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



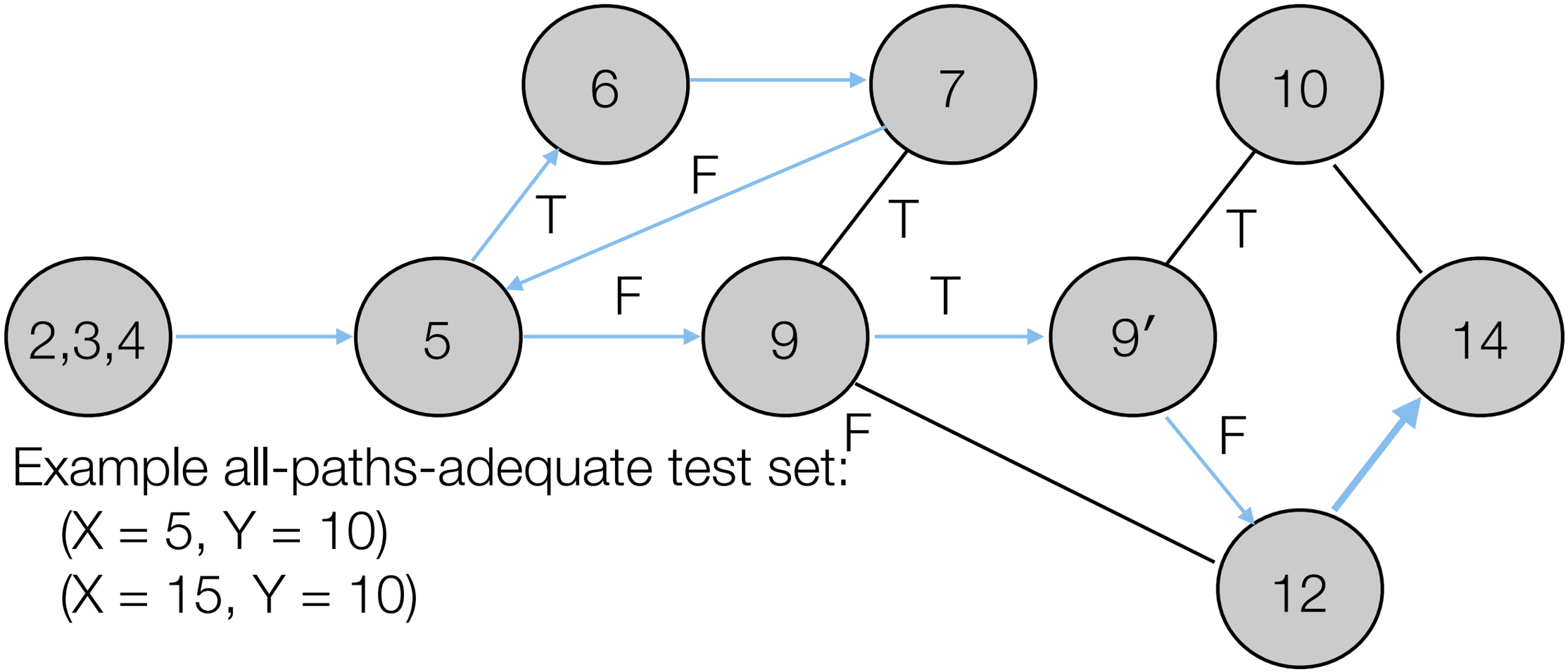
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



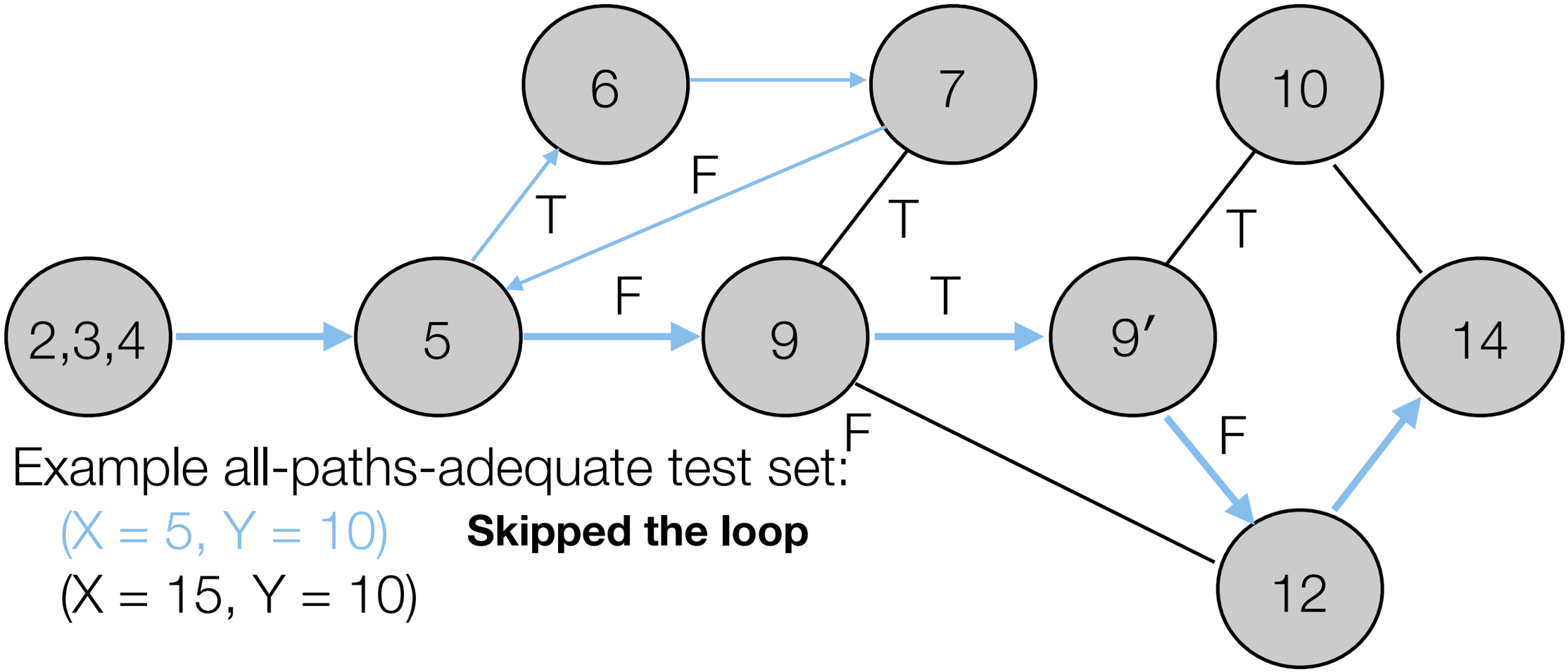
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



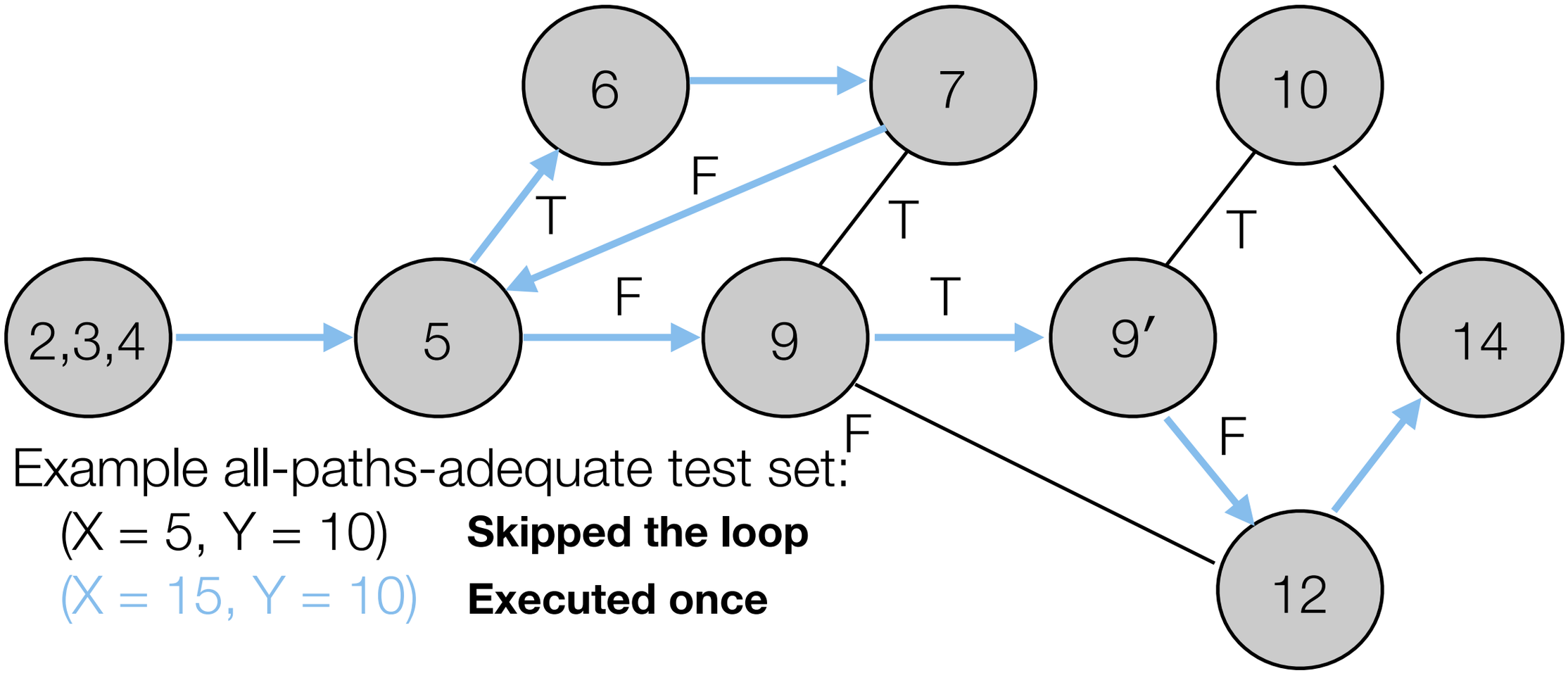
Example all-paths-adequate test set:

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

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



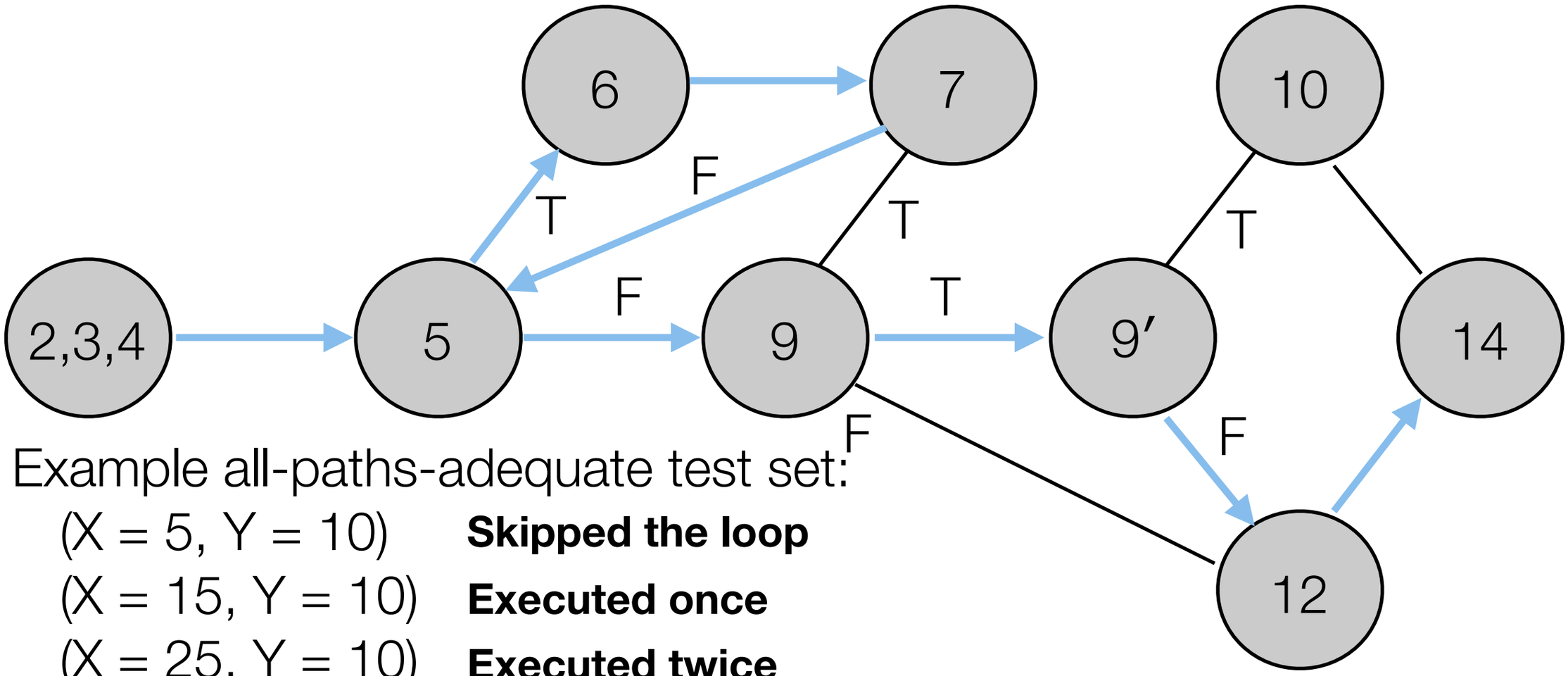
Example all-paths-adequate test set:

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

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

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



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**

And so on… you would also want permutations that exit the loop early

© Kenneth M. Anderson, 2012

Code Coverage Tools

- 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
    - These systems 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

© Kenneth M. Anderson, 2012

Testing Automation (I)

- 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!!)

© Kenneth M. Anderson, 2012

Testing Automation (II)

- 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
- We'll see examples of this later in the semester

© Kenneth M. Anderson, 2012

Continuous Integration

- Since test automation is so critical, systems known as continuous integration frameworks have emerged
- 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

© Kenneth M. Anderson, 2012

Summary

- 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 exists to integrate testing into the version control and build management processes of a development organization

© Kenneth M. Anderson, 2012

Coming Up Next:

- Lecture 6: Agile Methods and Agile Teams
- Lecture 7: Division of Labor and Design Approaches in Concurrency

© Kenneth M. Anderson, 2012

Introduction to Software Testing

CSCI 5828: Foundations of Software Engineering
Lecture 05 — 01/31/2012

© Kenneth M. Anderson, 2012

Goals

- Provide introduction to fundamental concepts of software testing
    - Terminology
    - Testing of Systems
        - unit tests, integration tests, system tests, acceptance tests
    - Testing of Code
        - Black Box
        - Gray Box
        - White Box
        - Code Coverage

© Kenneth M. Anderson, 2012

Testing

- Testing is a **critical element** of software development life cycles
  - called **software quality control** or **software quality assurance**
  - basic goals: **validation** and **verification**
    - validation: **are we building the right product?**
      - verification: **does "X" meet its specification?**
        - where "X" 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

© Kenneth M. Anderson, 2012

Terminology

- An **error** is a mistake made by an engineer
  - often a misunderstanding of a requirement or design specification
- A **fault** is a manifestation of that error in the code
  - what we often call “a bug”
- A **failure** is an incorrect output/behavior that is caused by executing a fault
  - The failure may occur immediately (crash!) or much, much later in the execution
- **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?

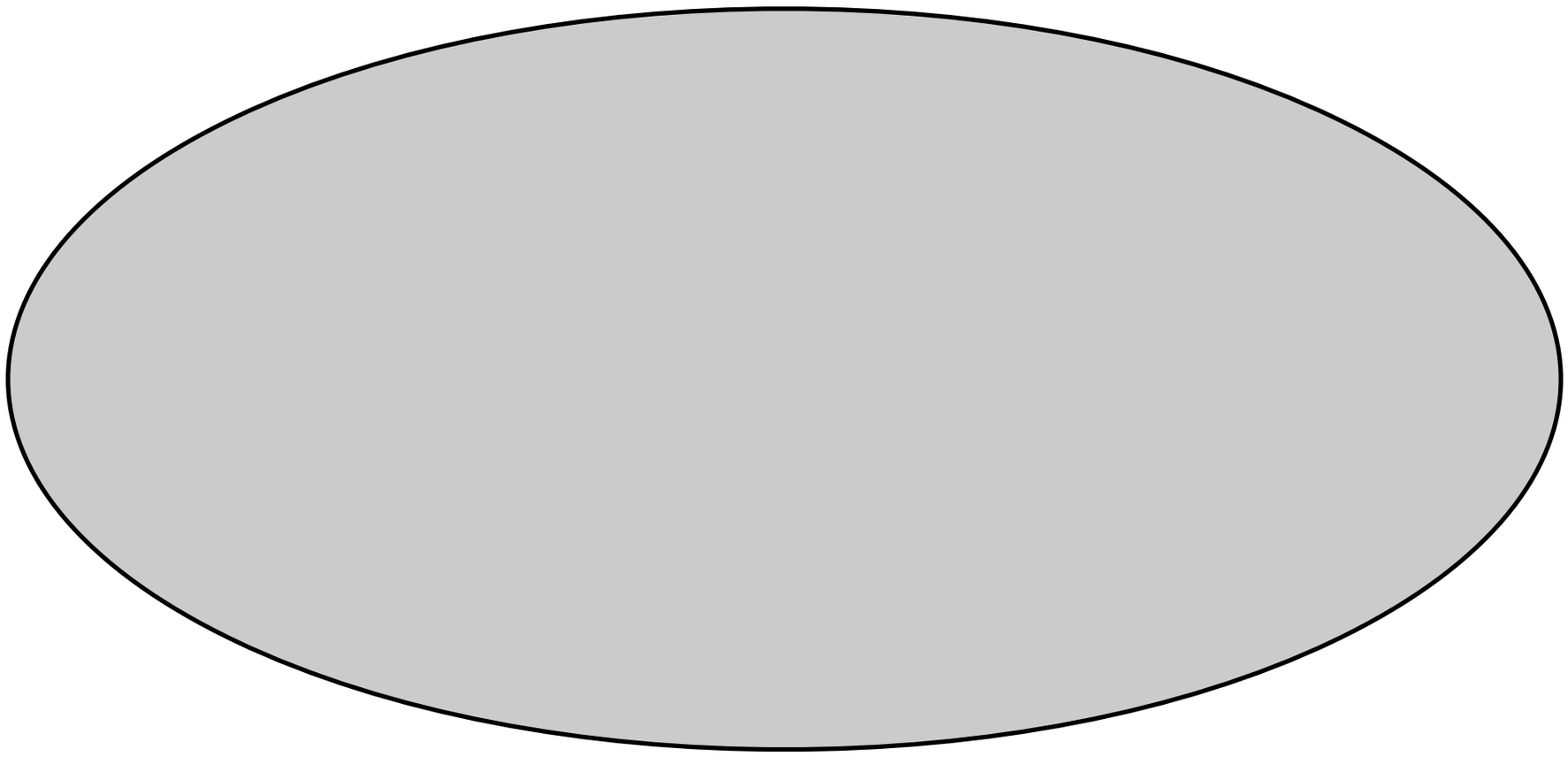
© Kenneth M. Anderson, 2012

No!

- 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 resulting in no visible failures!
      - this is particularly insidious
- However, if we do a good job in creating a test set that
  - covers all functional capabilities of a system
  - and 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

© Kenneth M. Anderson, 2012

Looking for Faults

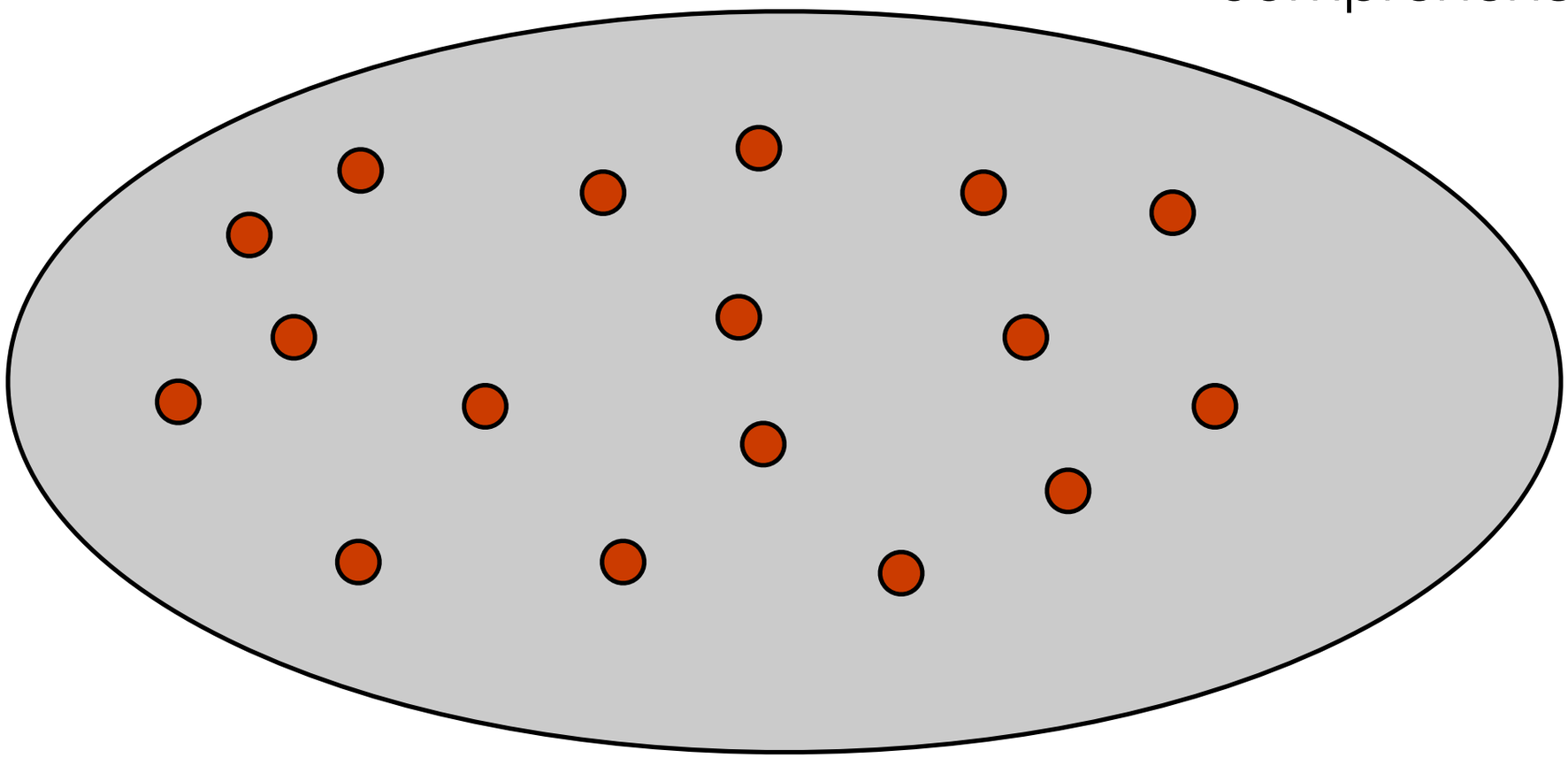


All possible states/behaviors of a system

© Kenneth M. Anderson, 2012

Looking for Faults

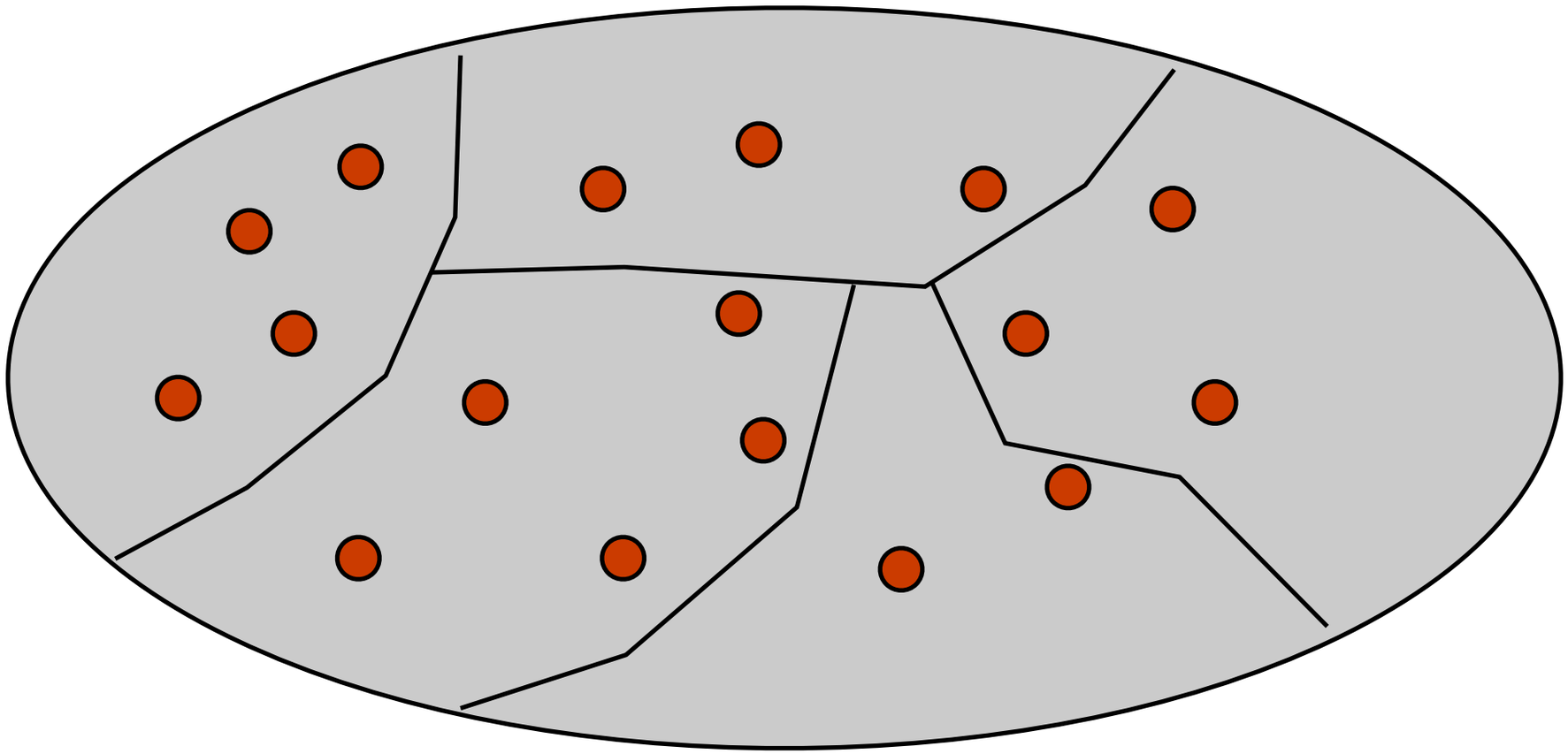
As you can see, its not very comprehensive



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

© Kenneth M. Anderson, 2012

One way forward? Fold



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

© Kenneth M. Anderson, 2012

What does that mean?

- 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

© Kenneth M. Anderson, 2012

Completeness

- 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?

© Kenneth M. Anderson, 2012

Continuous Testing

- Testing is a continuous process that should be performed at every stage of a software development process
    - During requirements gathering, for instance, we must continually query the user, “Did we get this right?”
    - Facilitated by an emphasis on iteration throughout a life cycle
- at the end of each iteration
    - we check our results to see if what we built is meeting our requirements (specification)

© Kenneth M. Anderson, 2012

Testing the System (I)

- **Unit Tests**
  - Tests that cover low-level aspects of a system
    - For each module, does each operation perform as expected
    - For method **foo()**, we'd like to see another method **testFoo()**
- **Integration Tests**
  - Tests that check that modules work together in combination
  - Most projects on schedule until they hit this point (MMM, Brooks)
    - 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 (Mars Polar Lander)

© Kenneth M. Anderson, 2012

Testing the System (II)

- **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

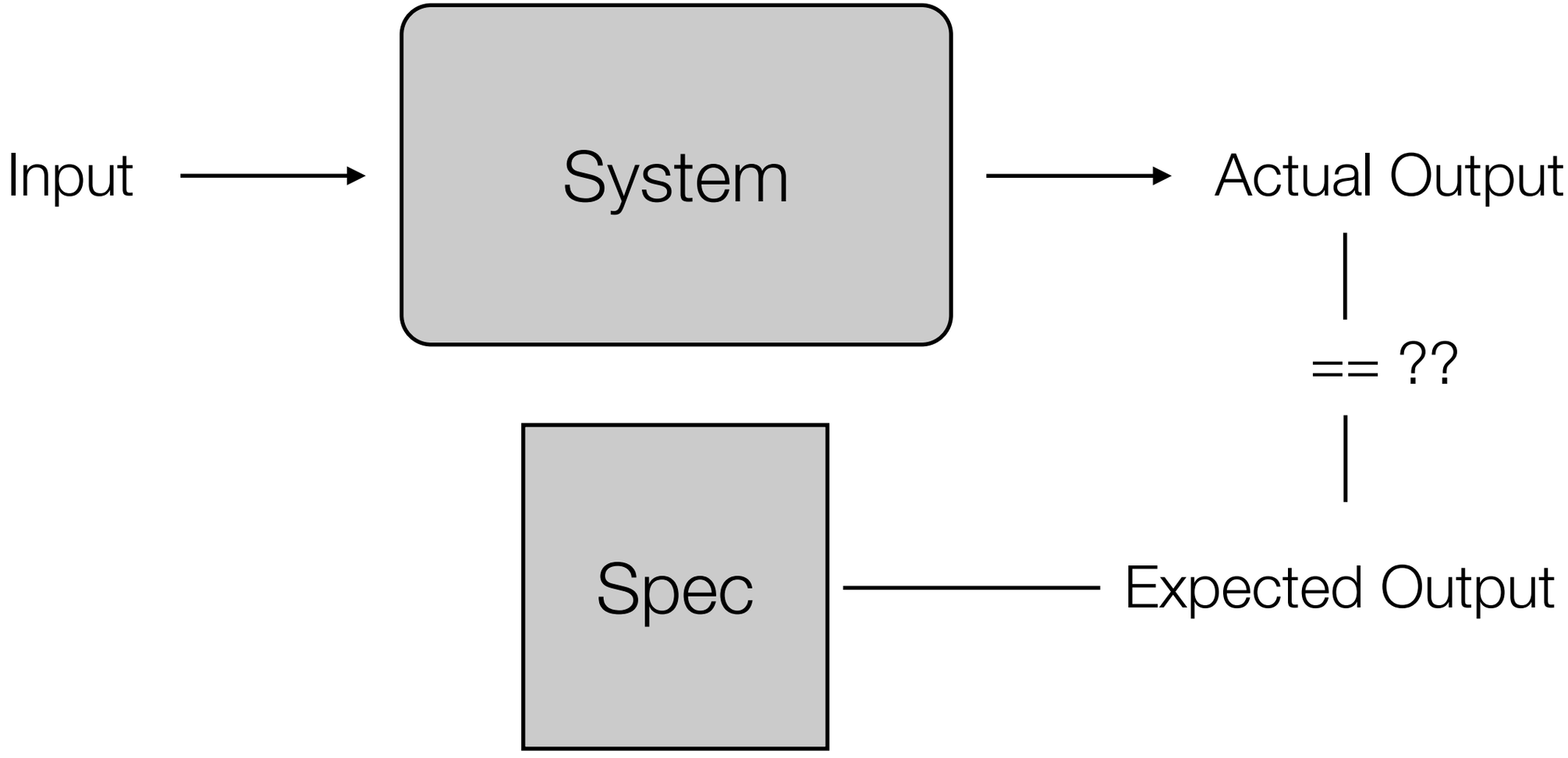
© Kenneth M. Anderson, 2012

Multi-Level Testing

- 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, …

© Kenneth M. Anderson, 2012

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!

© Kenneth M. Anderson, 2012

Results

- 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

© Kenneth M. Anderson, 2012

Black Box Categories

- 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

© Kenneth M. Anderson, 2012

Grey Box Testing

- 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

© Kenneth M. Anderson, 2012

White Box Testing

- 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?

© Kenneth M. Anderson, 2012

Code Coverage (I)

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

© Kenneth M. Anderson, 2012

Code Coverage (II)

- 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)

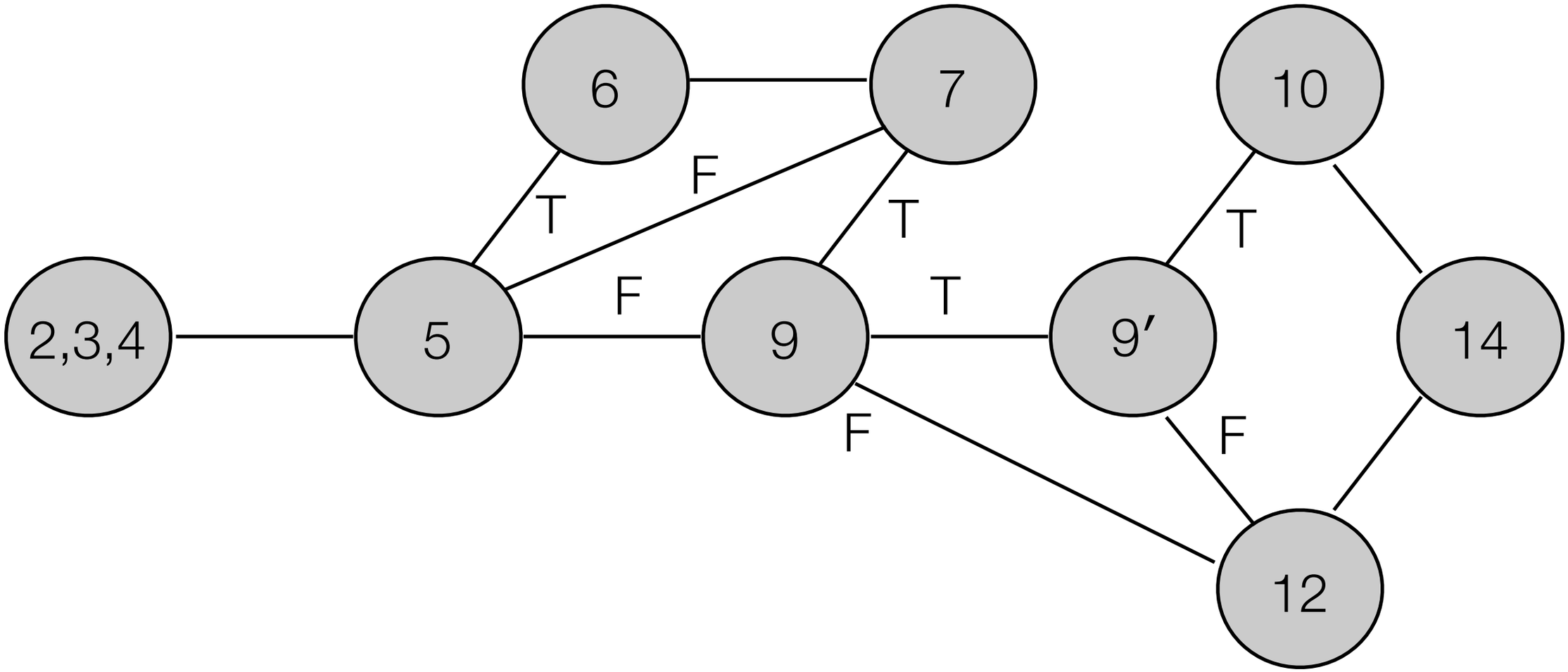
© Kenneth M. Anderson, 2012

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;
```

© Kenneth M. Anderson, 2012

P’s Control Flow Graph (CFG)



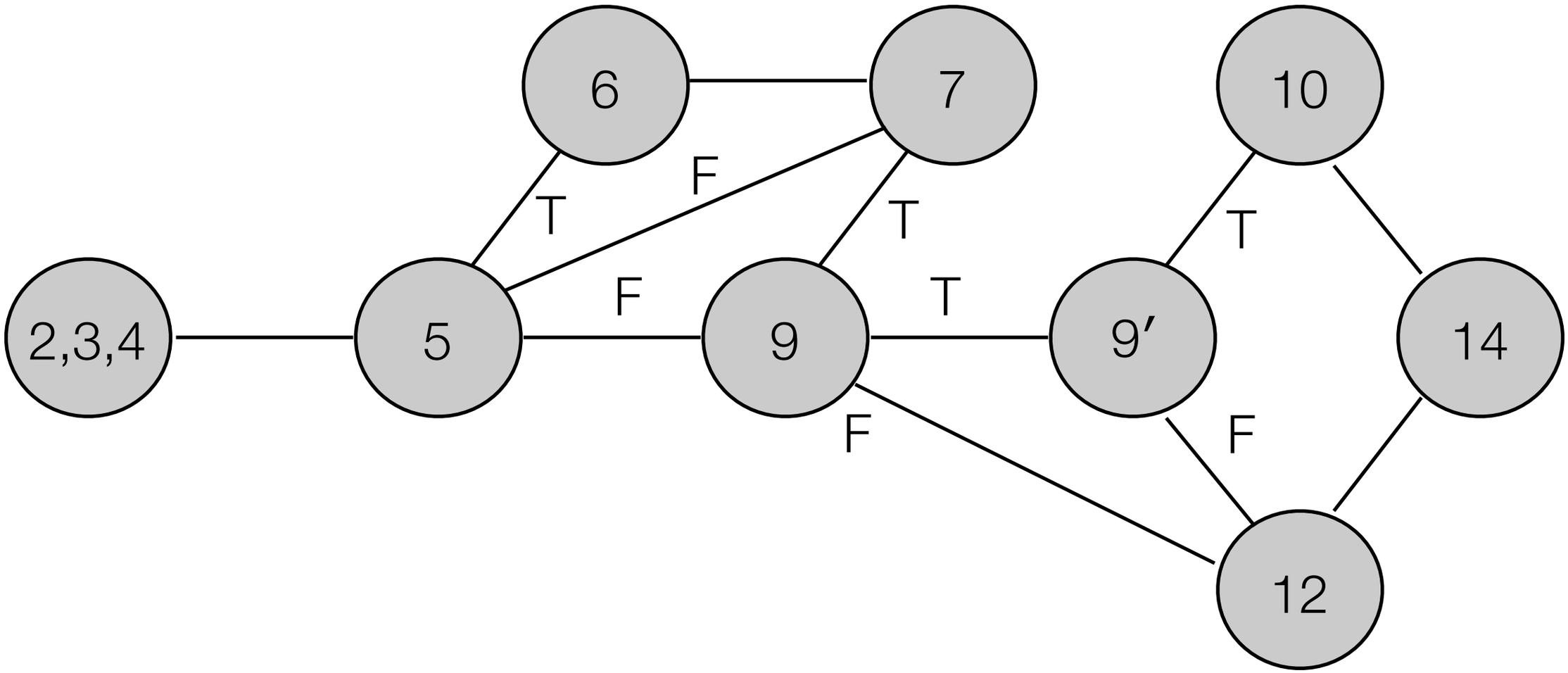
© Kenneth M. Anderson, 2012

White-box Testing Criteria

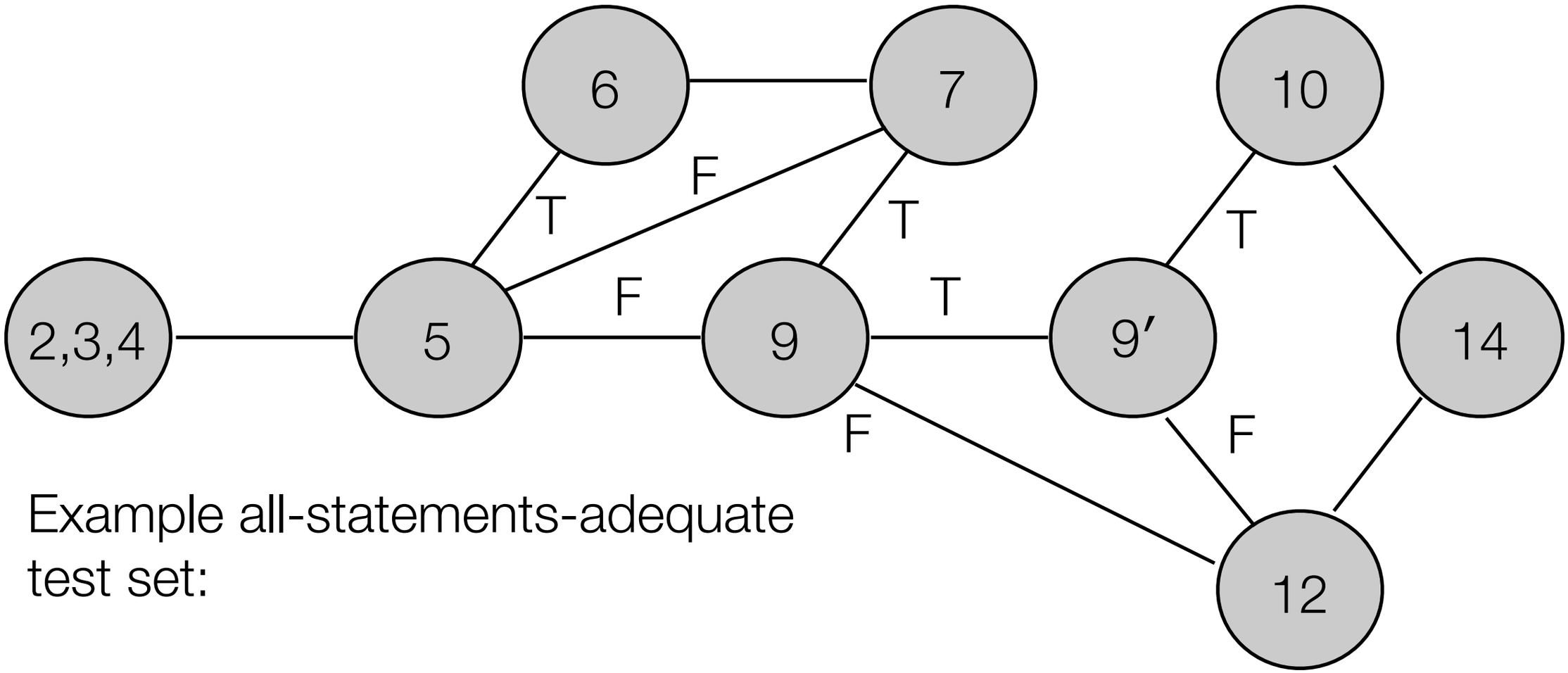
- Statement Coverage
  - Create a test set T such that
    - by executing P for each t in T
      - each elementary statement of P is executed at least once

© Kenneth M. Anderson, 2012

All-Statements Coverage of P

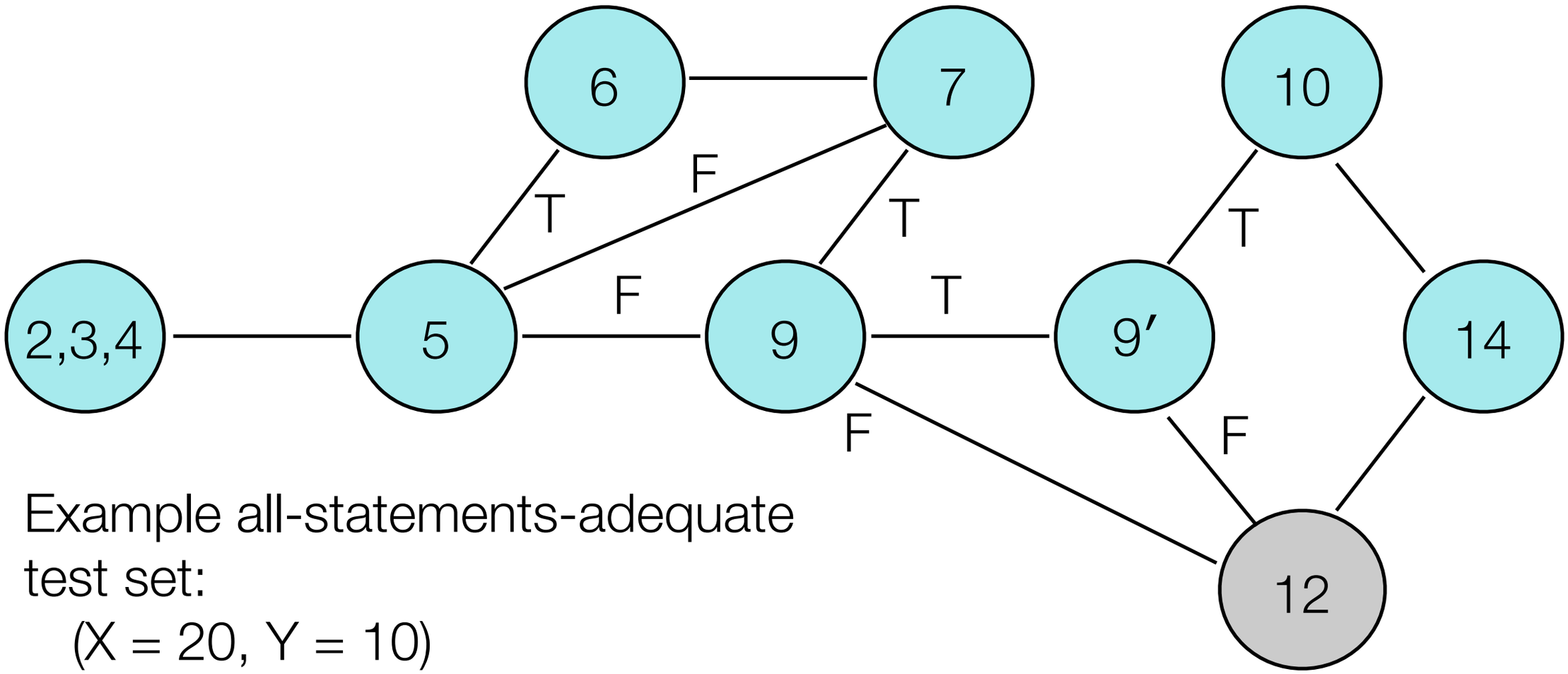


All-Statements Coverage of P



Example all-statements-adequate test set:

All-Statements Coverage of P

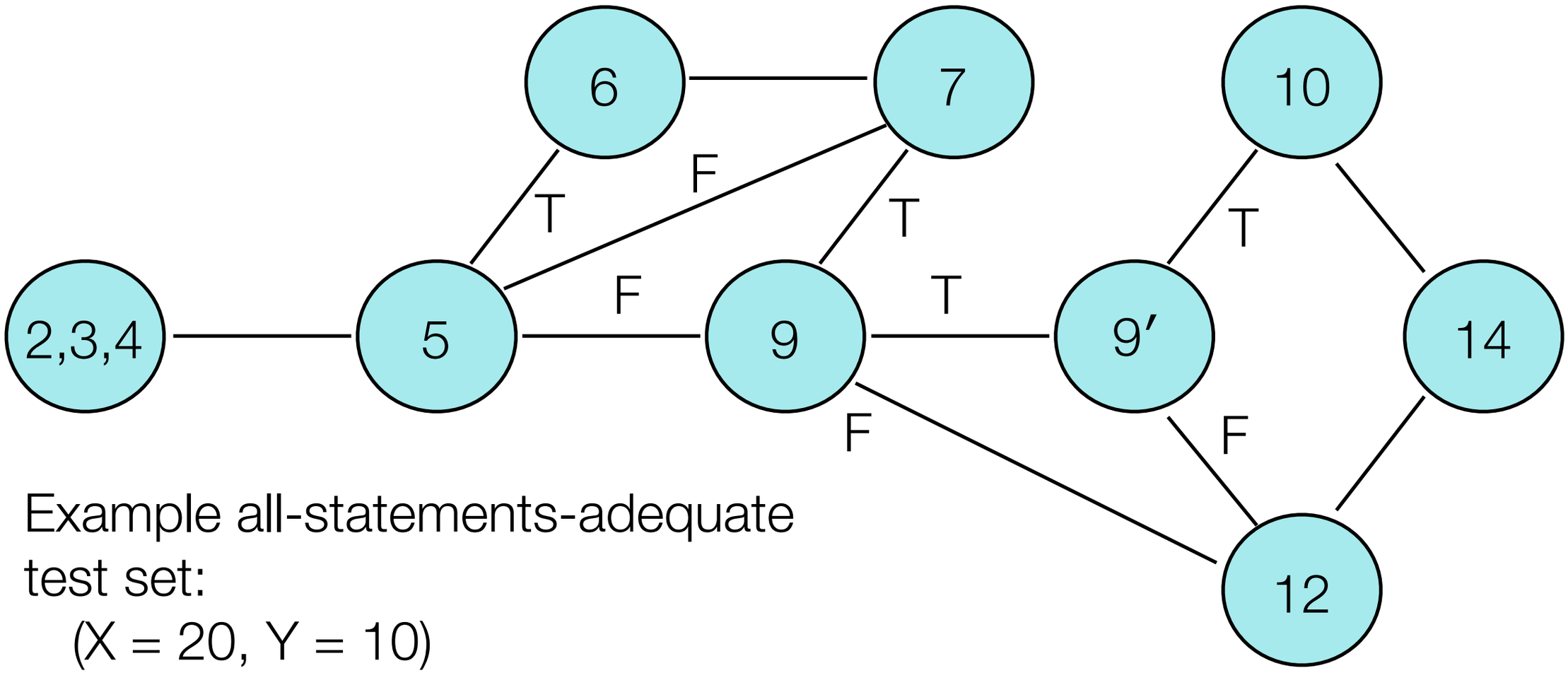


Example all-statements-adequate test set:

(X = 20, Y = 10)

© Kenneth M. Anderson, 2012

All-Statements Coverage of P



Example all-statements-adequate test set:

(X = 20, Y = 10)

(X = 20, Y = 30)

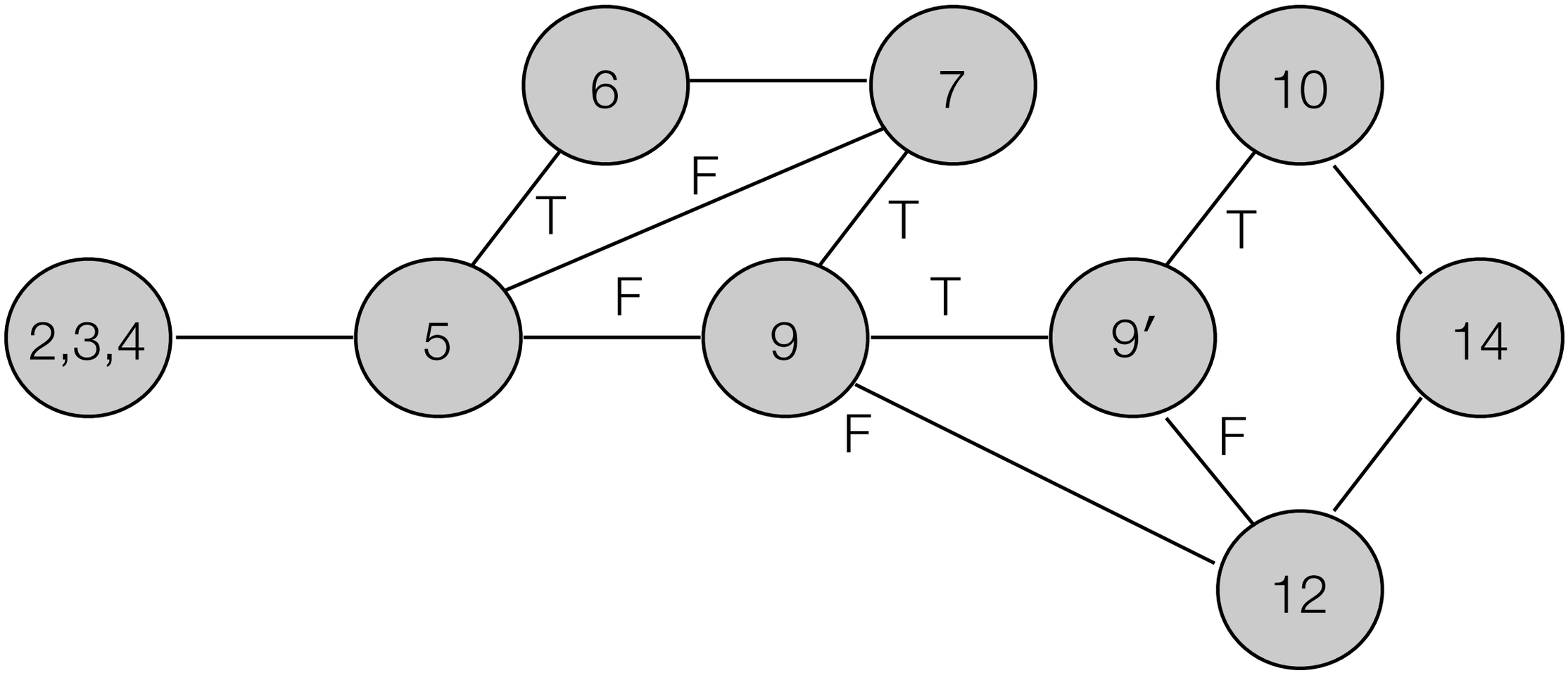
© Kenneth M. Anderson, 2012

White-box Testing Criteria

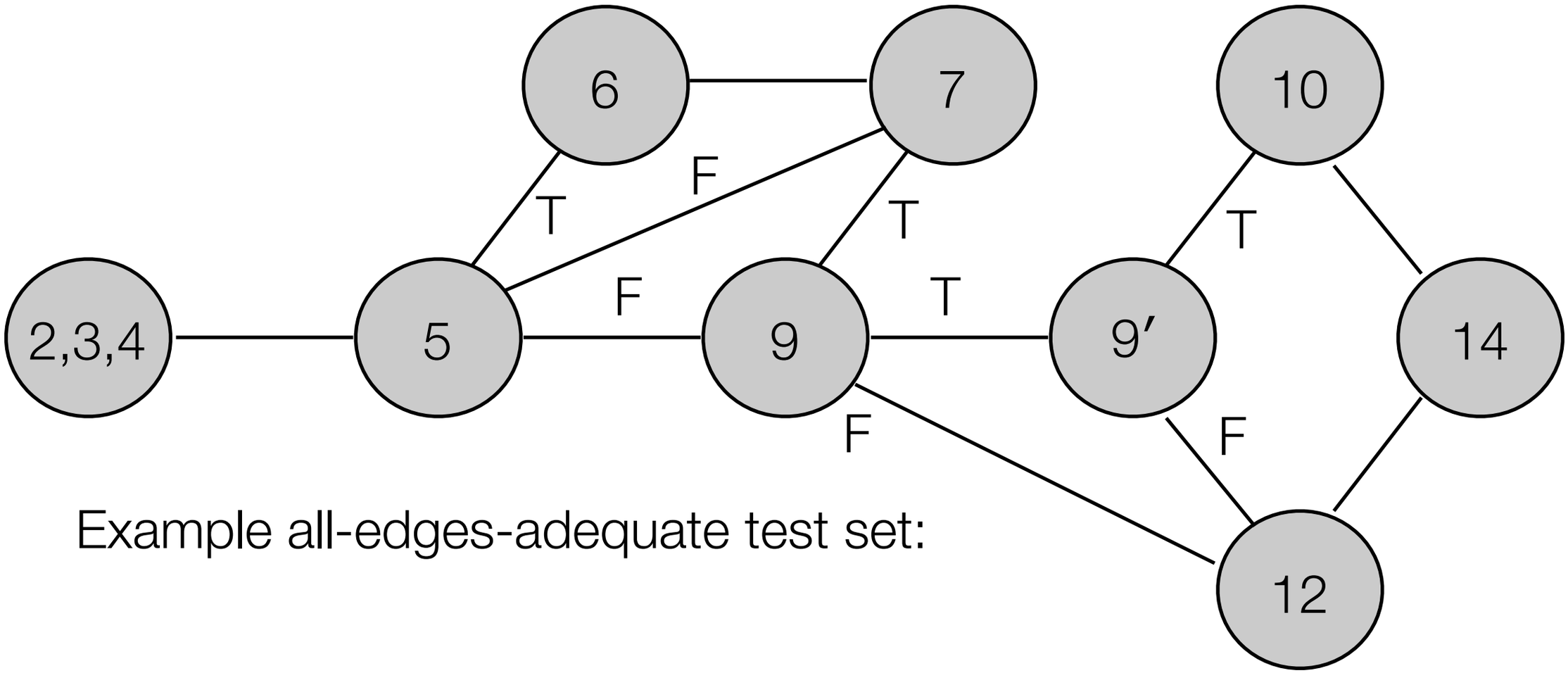
- Edge Coverage
  - Select a test set T such that
    - by executing P for each t in T
      - each edge of P's control flow graph is traversed at least once

© Kenneth M. Anderson, 2012

All-Edges Coverage of P



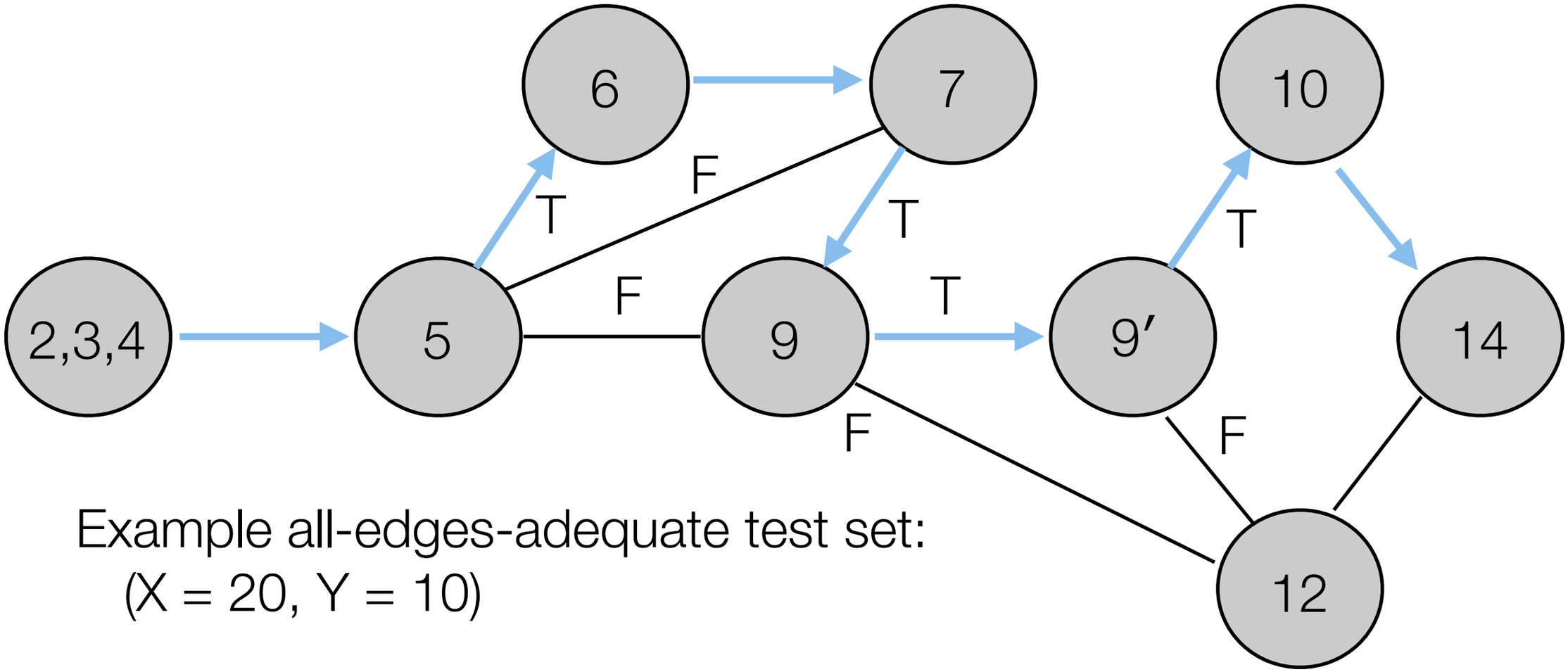
All-Edges Coverage of P



Example all-edges-adequate test set:

© Kenneth M. Anderson, 2012

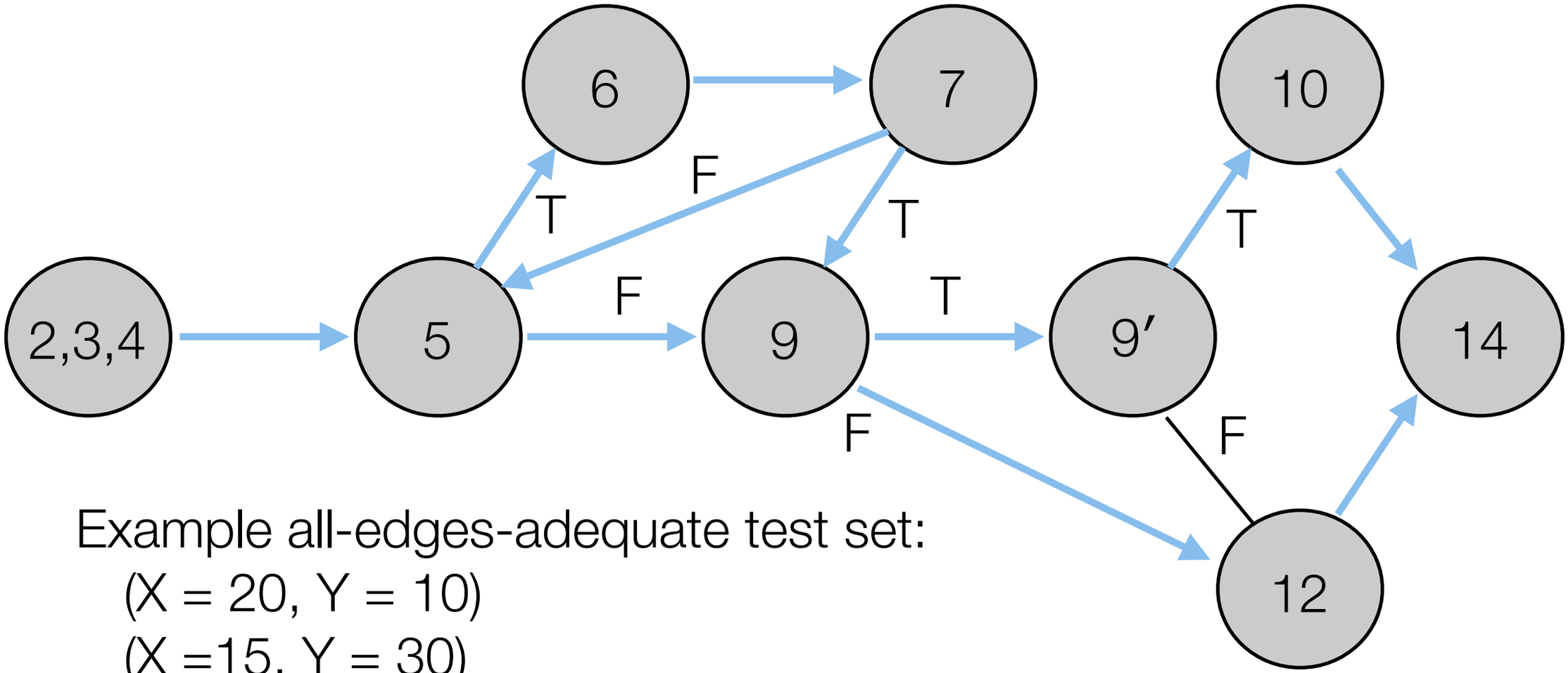
All-Edges Coverage of P



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

© Kenneth M. Anderson, 2012

All-Edges Coverage of P



Example all-edges-adequate test set:

(X = 20, Y = 10)

(X =15, Y = 30)

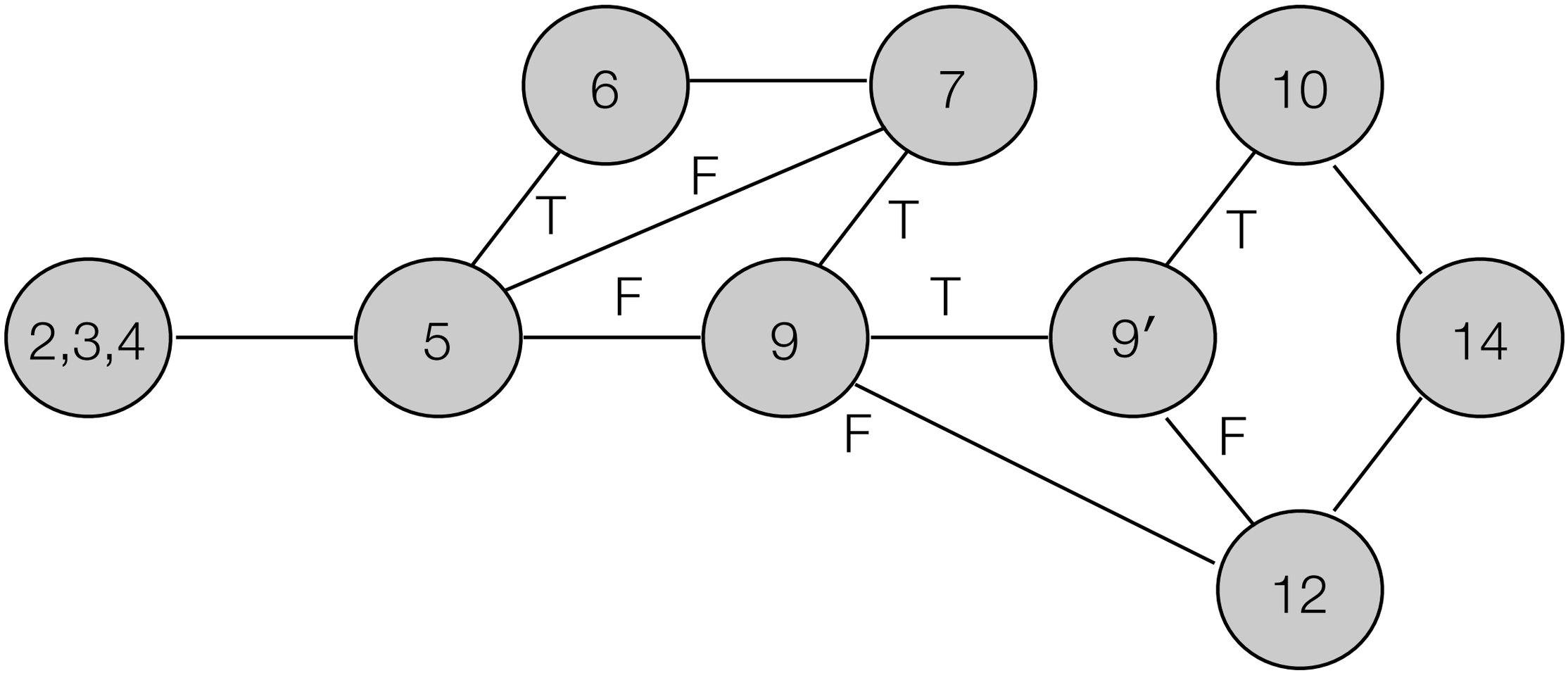
© Kenneth M. Anderson, 2012

White-box Testing Criteria

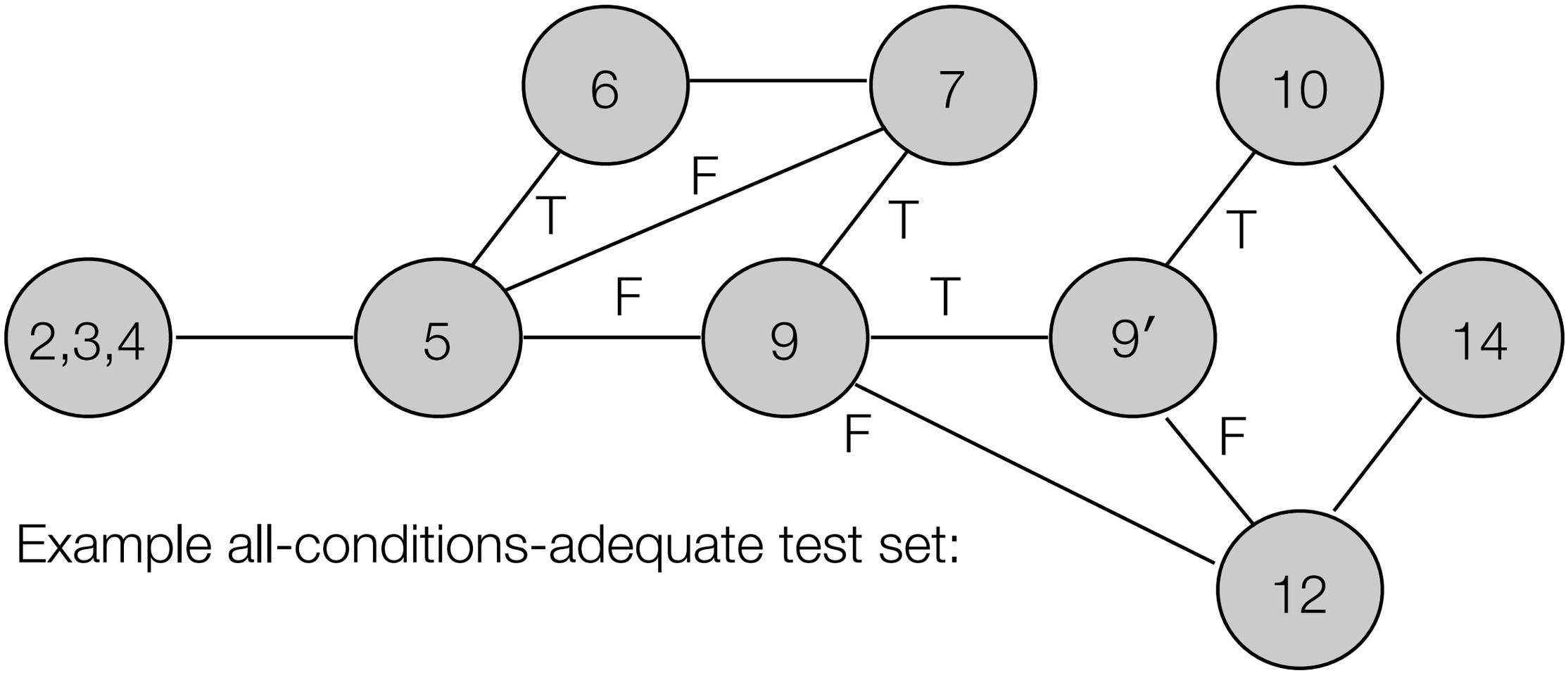
- Condition Coverage
  - Select a test set T such that
    - by executing P for each t in T
      - each edge of P's control flow graph is traversed at least once
      - and all possible values of the constituents of compound conditions are exercised at least once

© Kenneth M. Anderson, 2012

All-Conditions Coverage of P



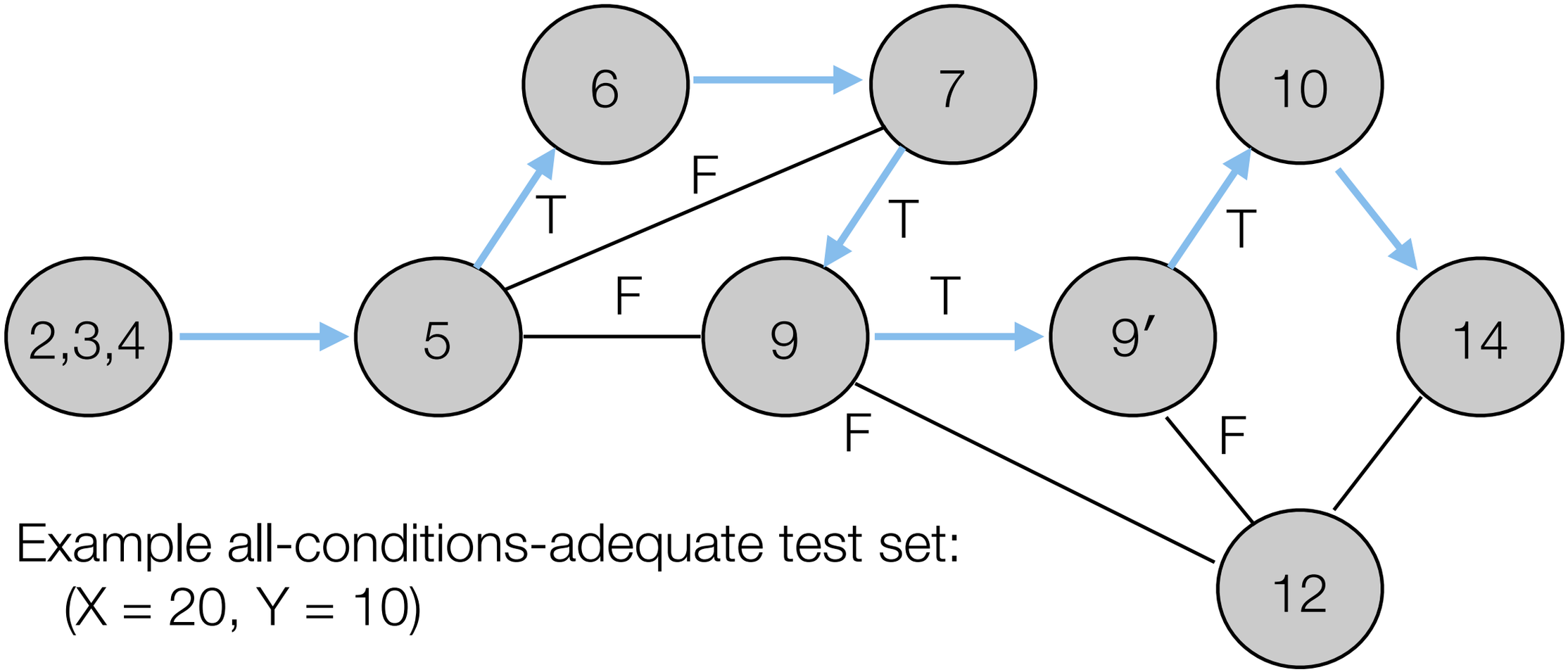
All-Conditions Coverage of P



Example all-conditions-adequate test set:

© Kenneth M. Anderson, 2012

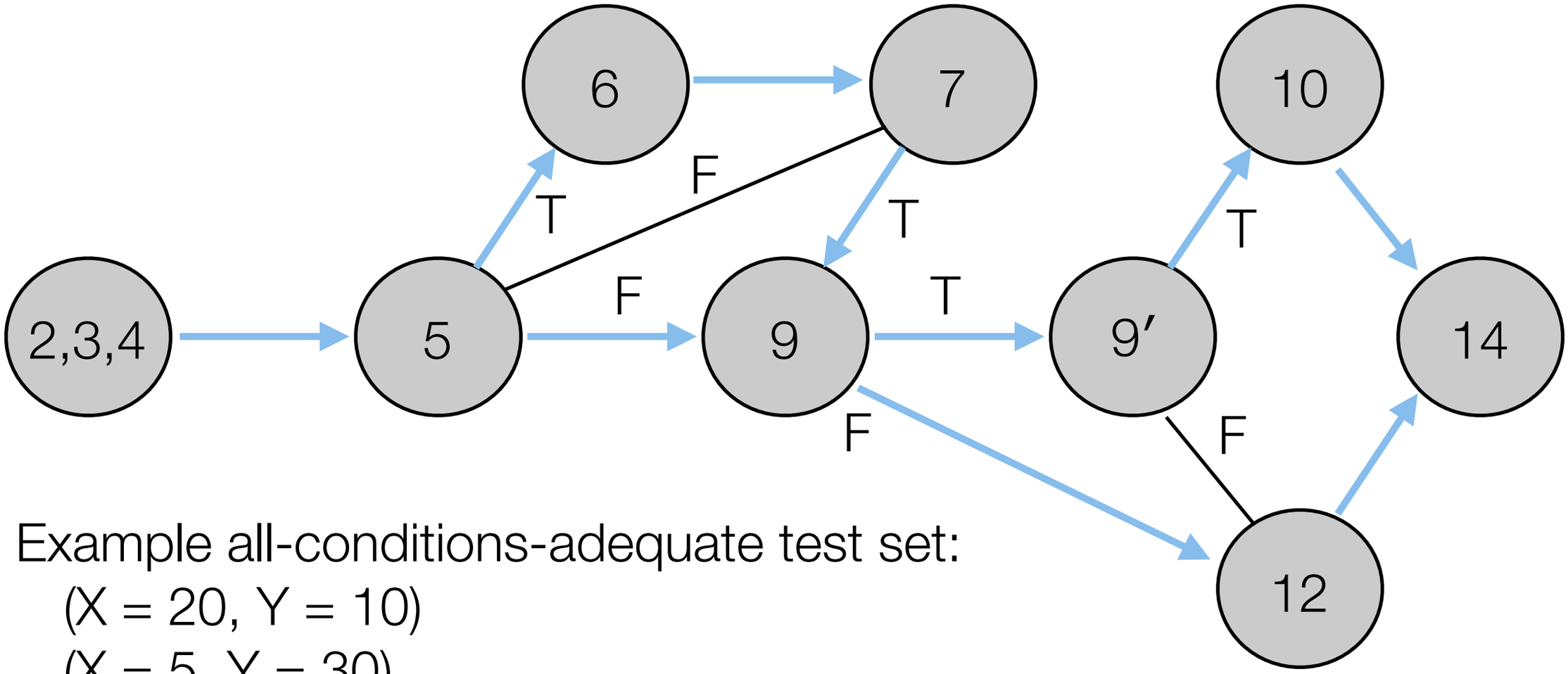
All-Conditions Coverage of P



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

© Kenneth M. Anderson, 2012

All-Conditions Coverage of P



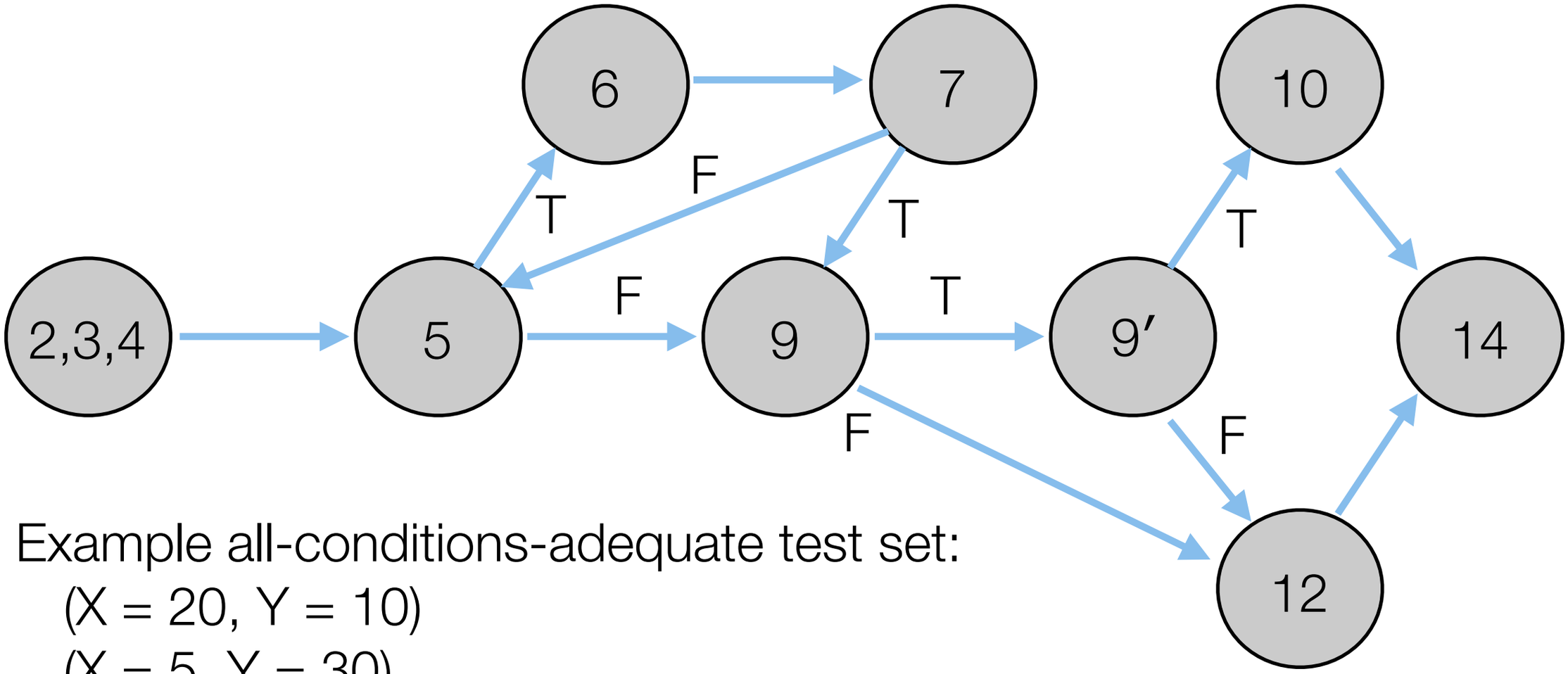
Example all-conditions-adequate test set:

(X = 20, Y = 10)

(X = 5, Y = 30)

© Kenneth M. Anderson, 2012

All-Conditions Coverage of P



Example all-conditions-adequate test set:

(X = 20, Y = 10)

(X = 5, Y = 30)

(X = 21, Y = 10)

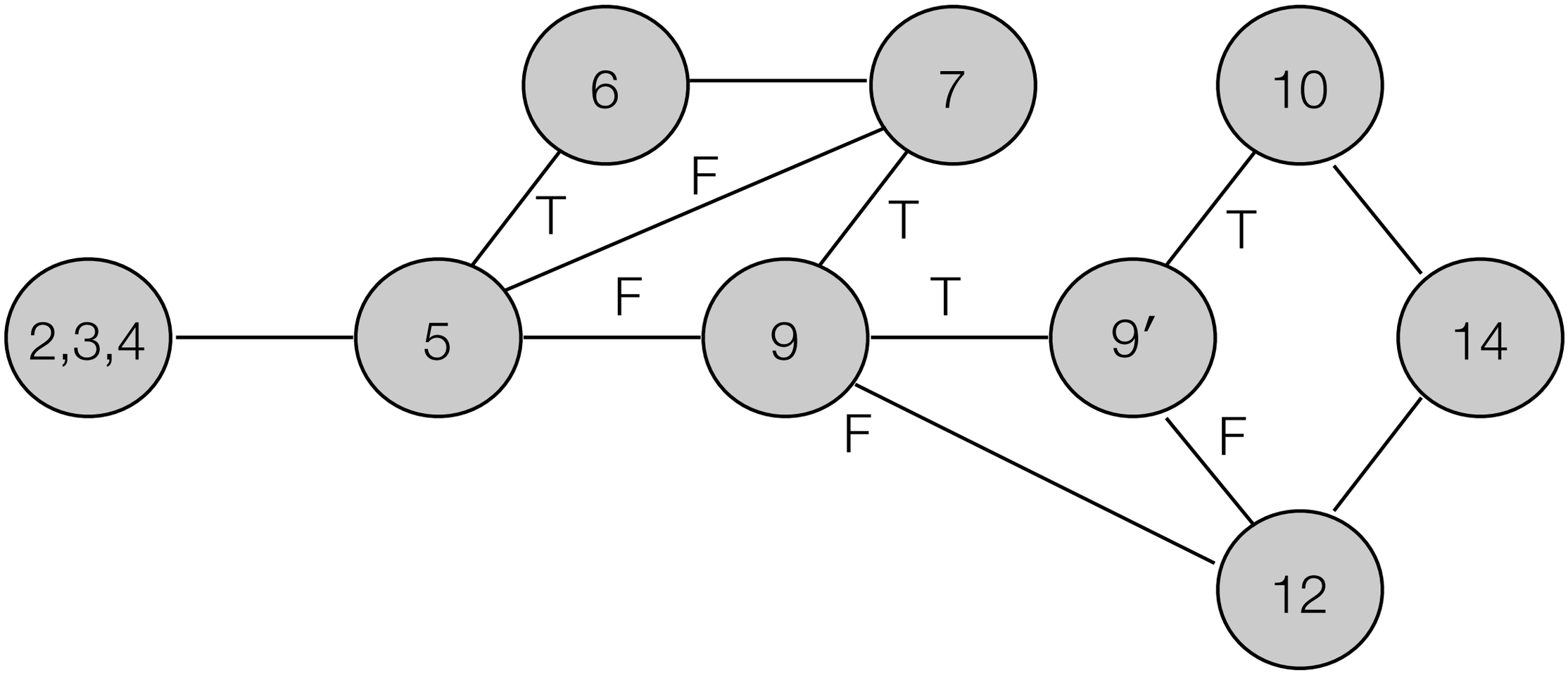
© Kenneth M. Anderson, 2012

White-box Testing Criteria

- 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

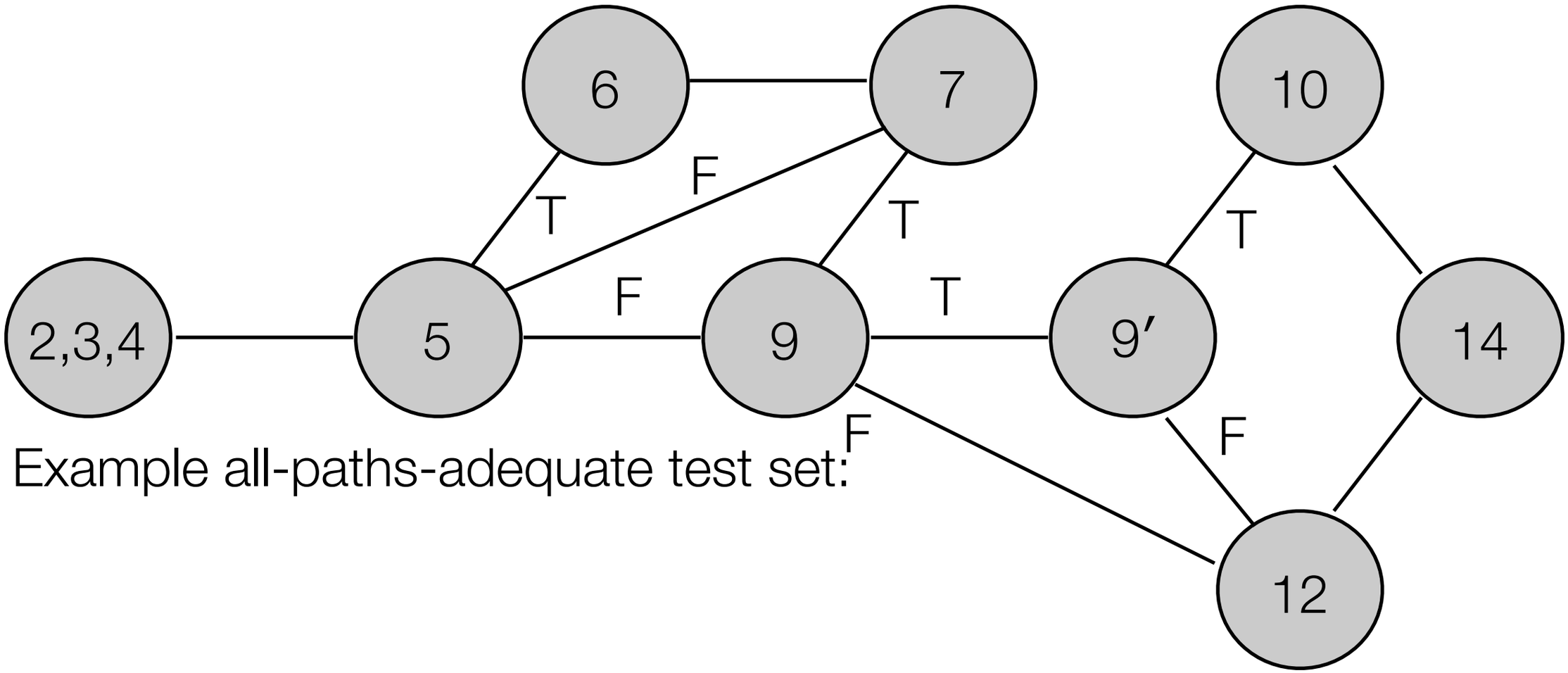
© Kenneth M. Anderson, 2012

All-Paths Coverage of P



© Kenneth M. Anderson, 2012

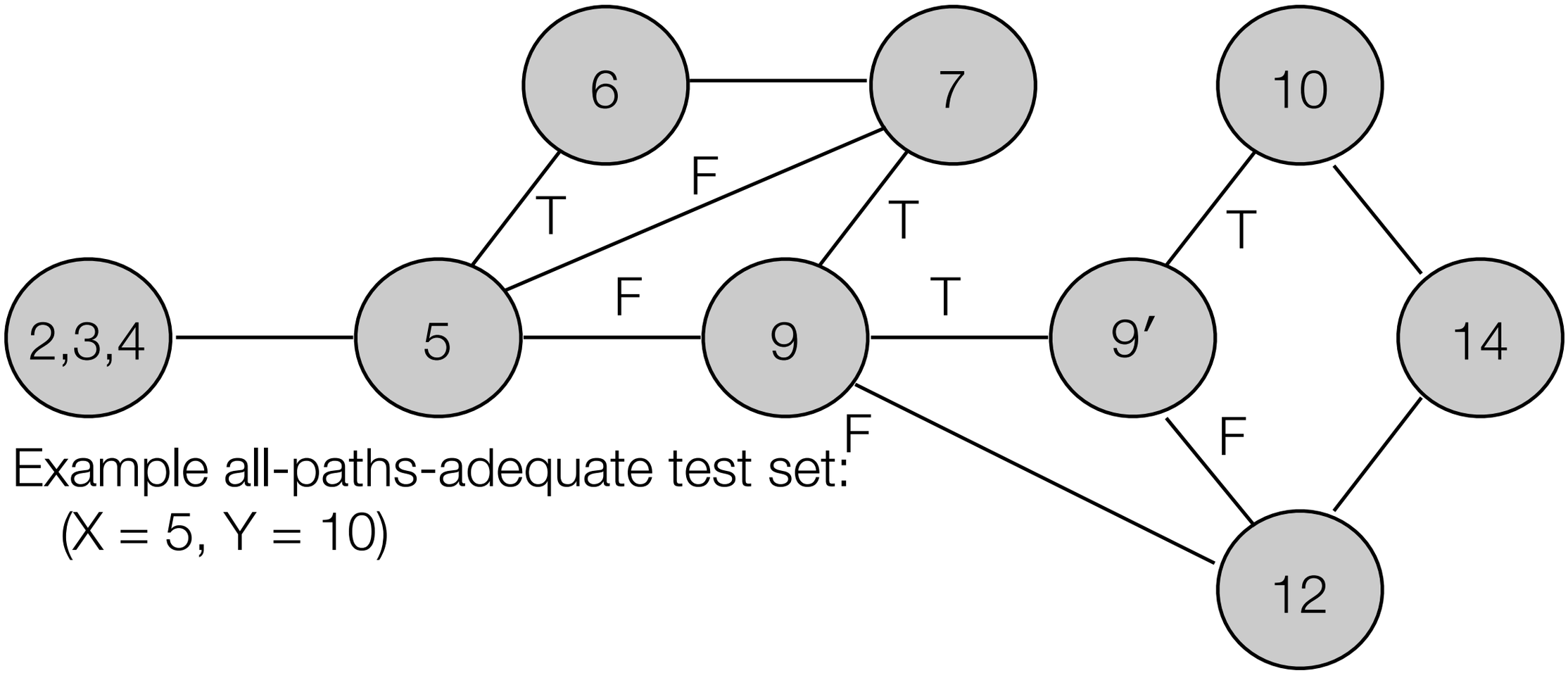
All-Paths Coverage of P



Example all-paths-adequate test set:

© Kenneth M. Anderson, 2012

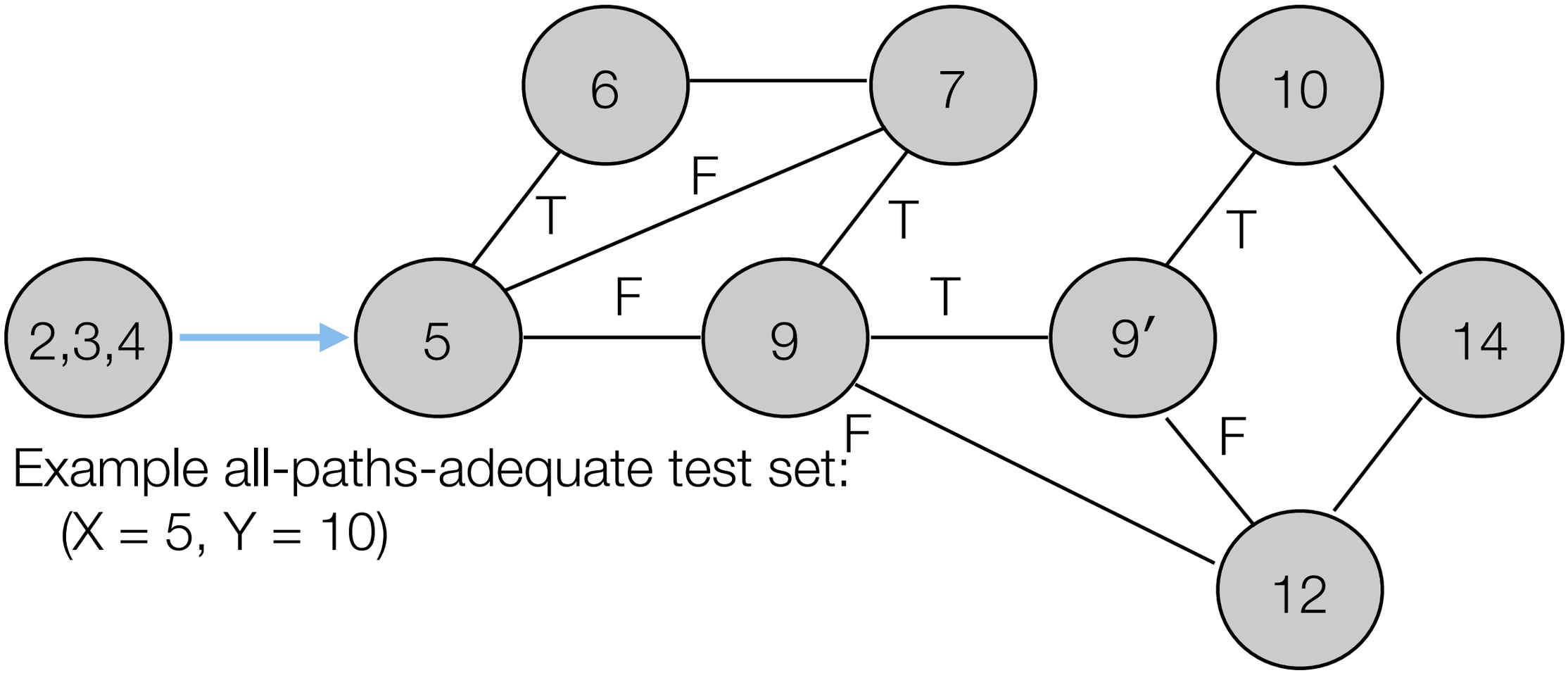
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

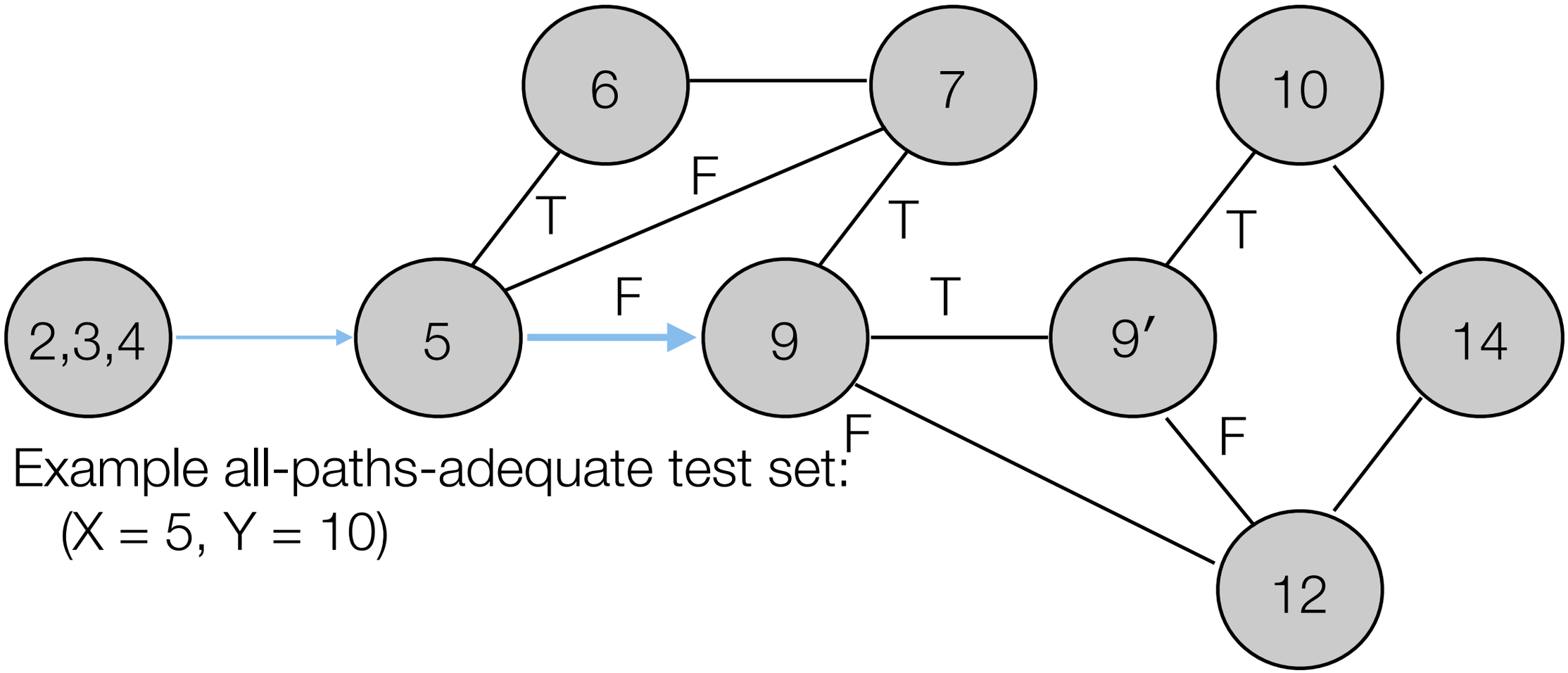
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

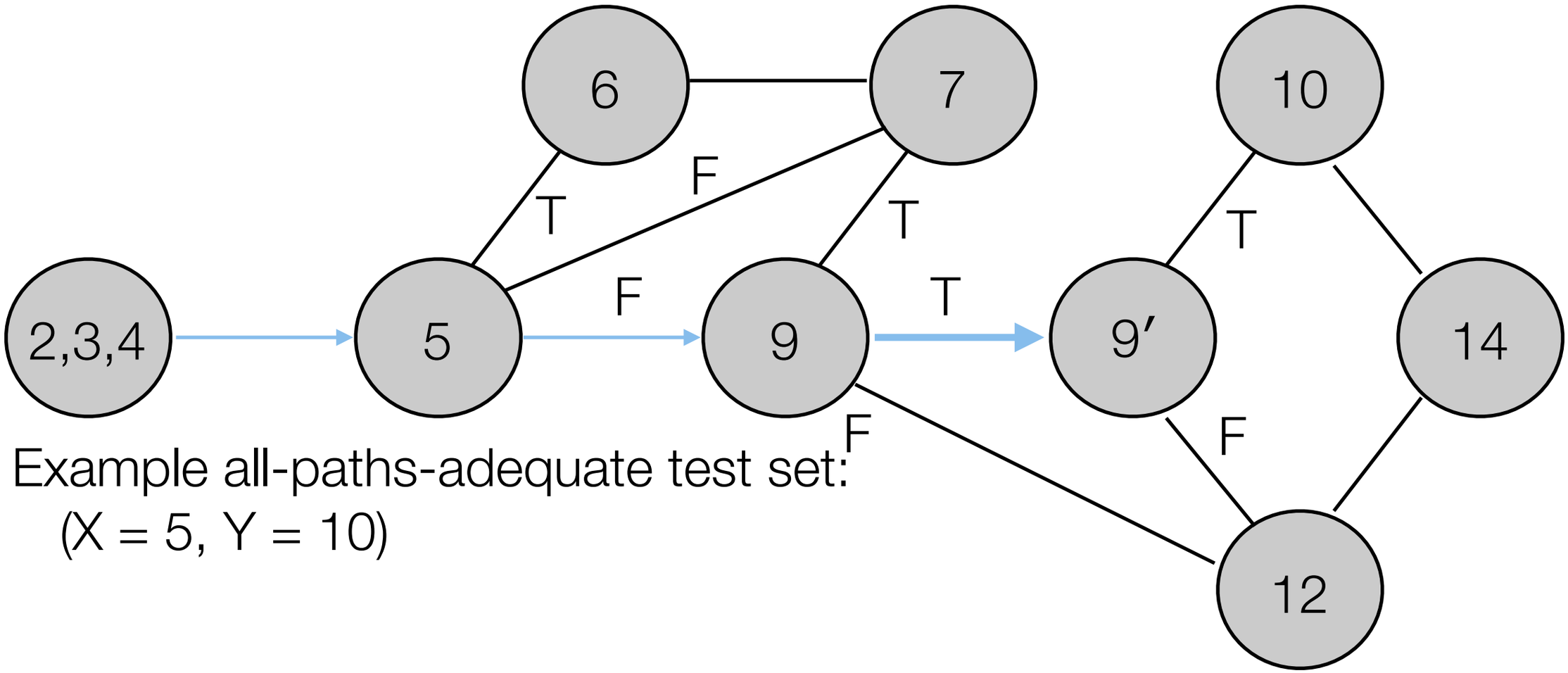
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

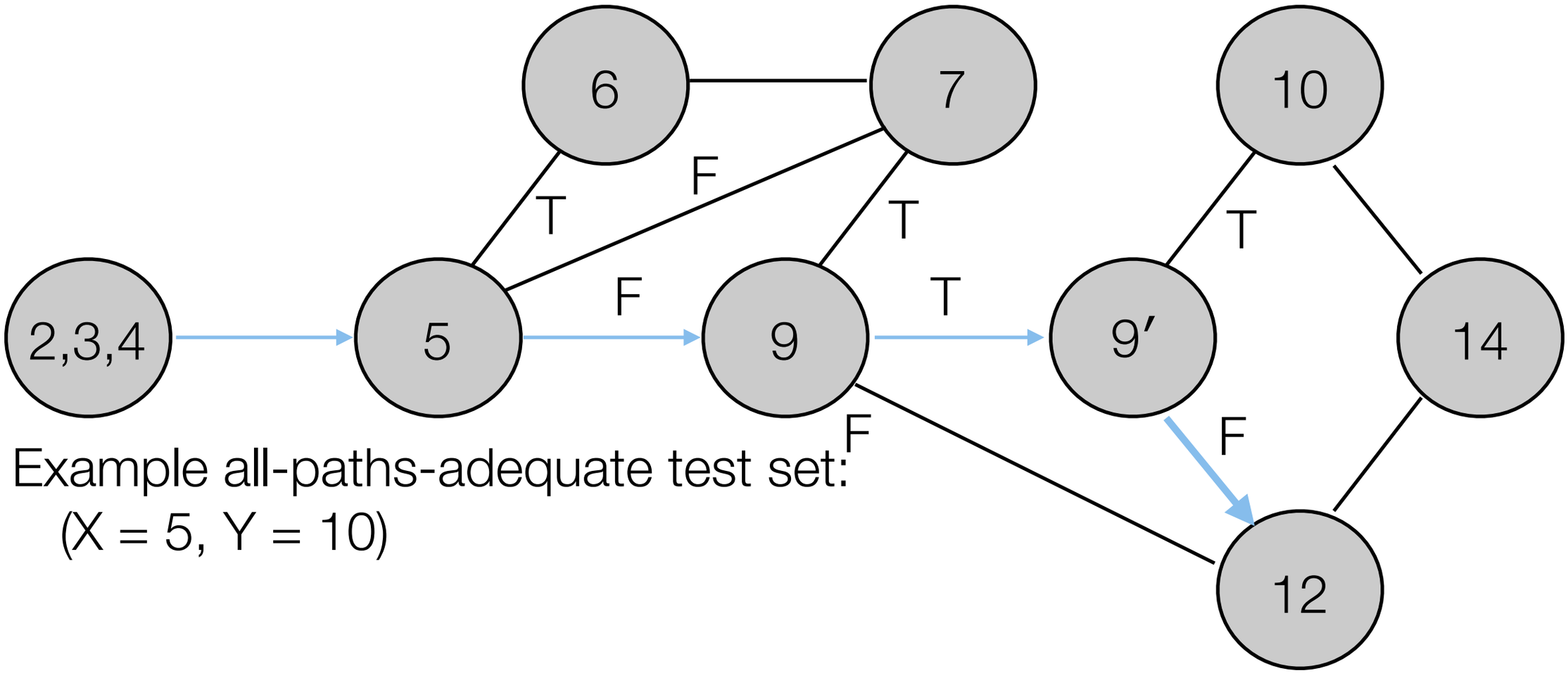
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

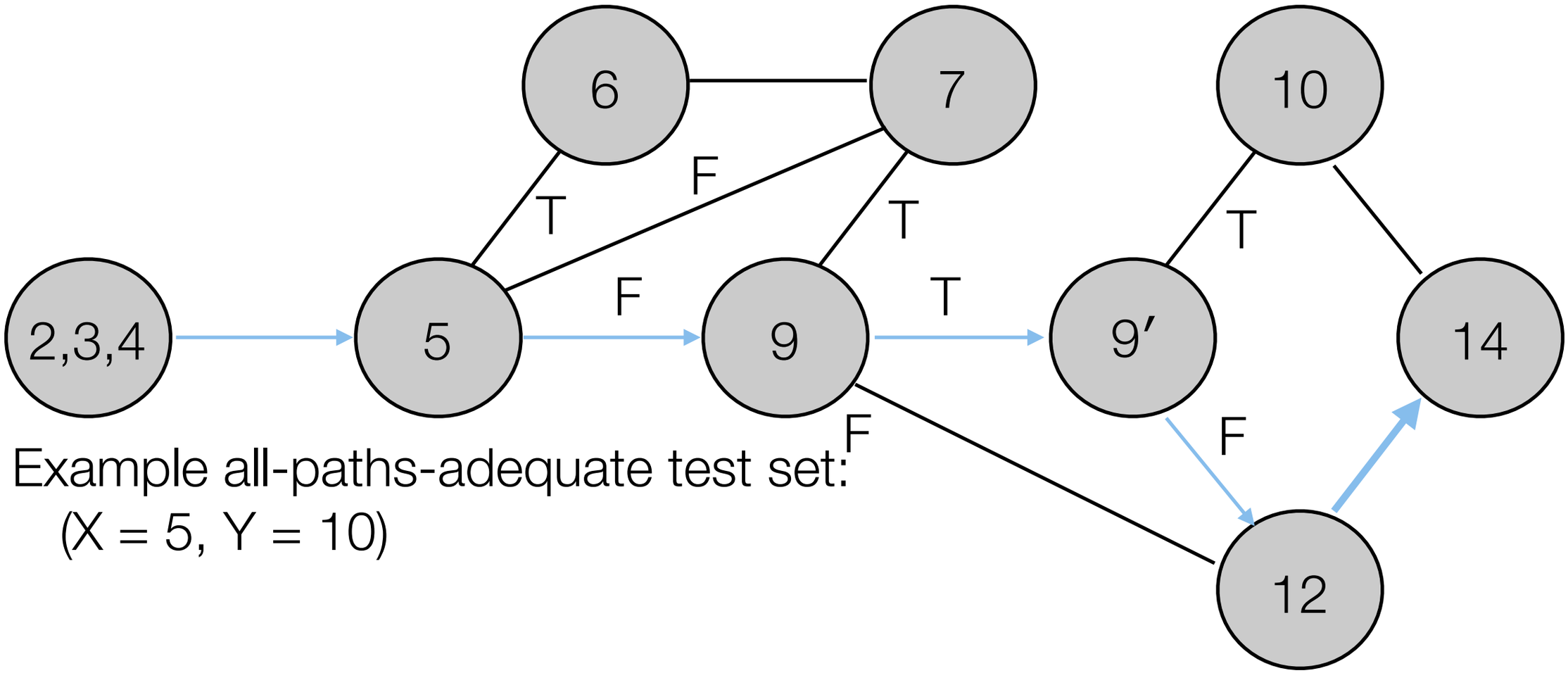
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

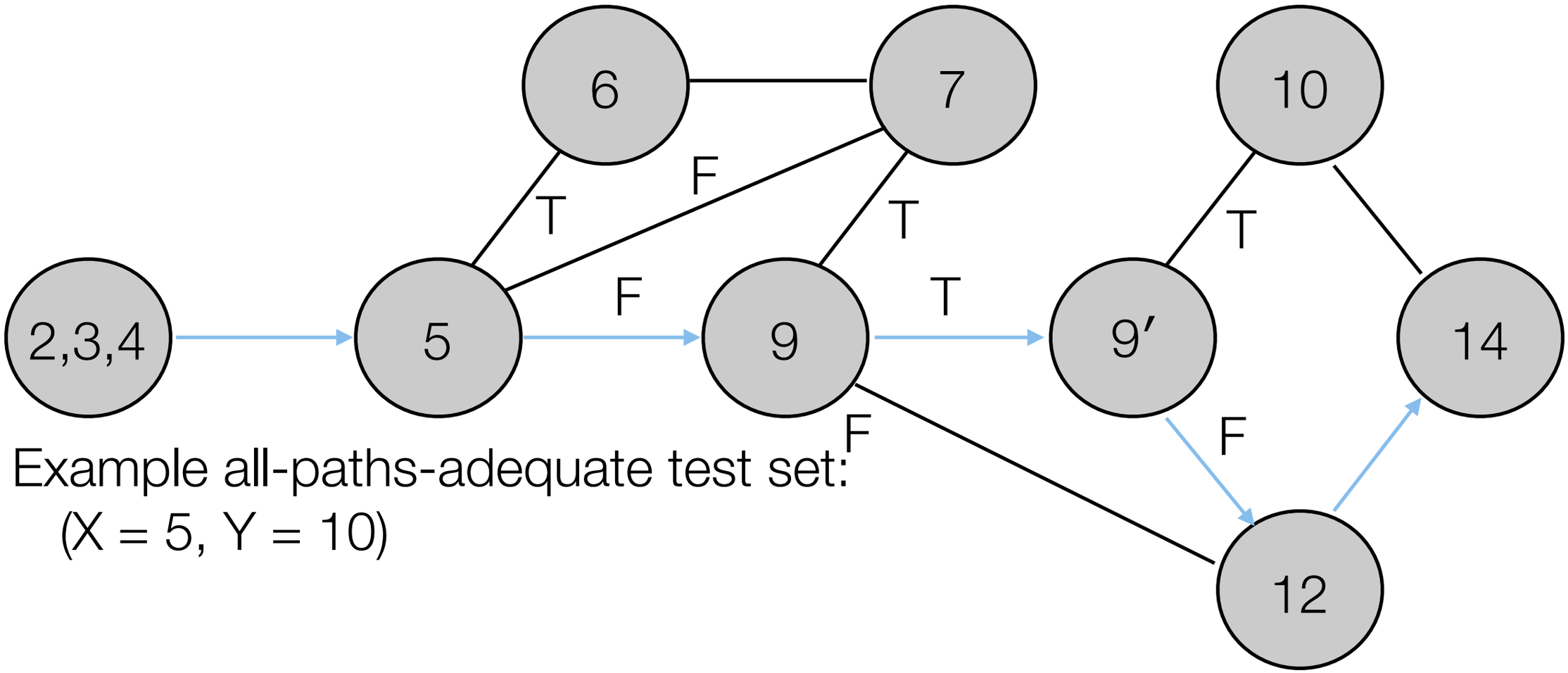
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

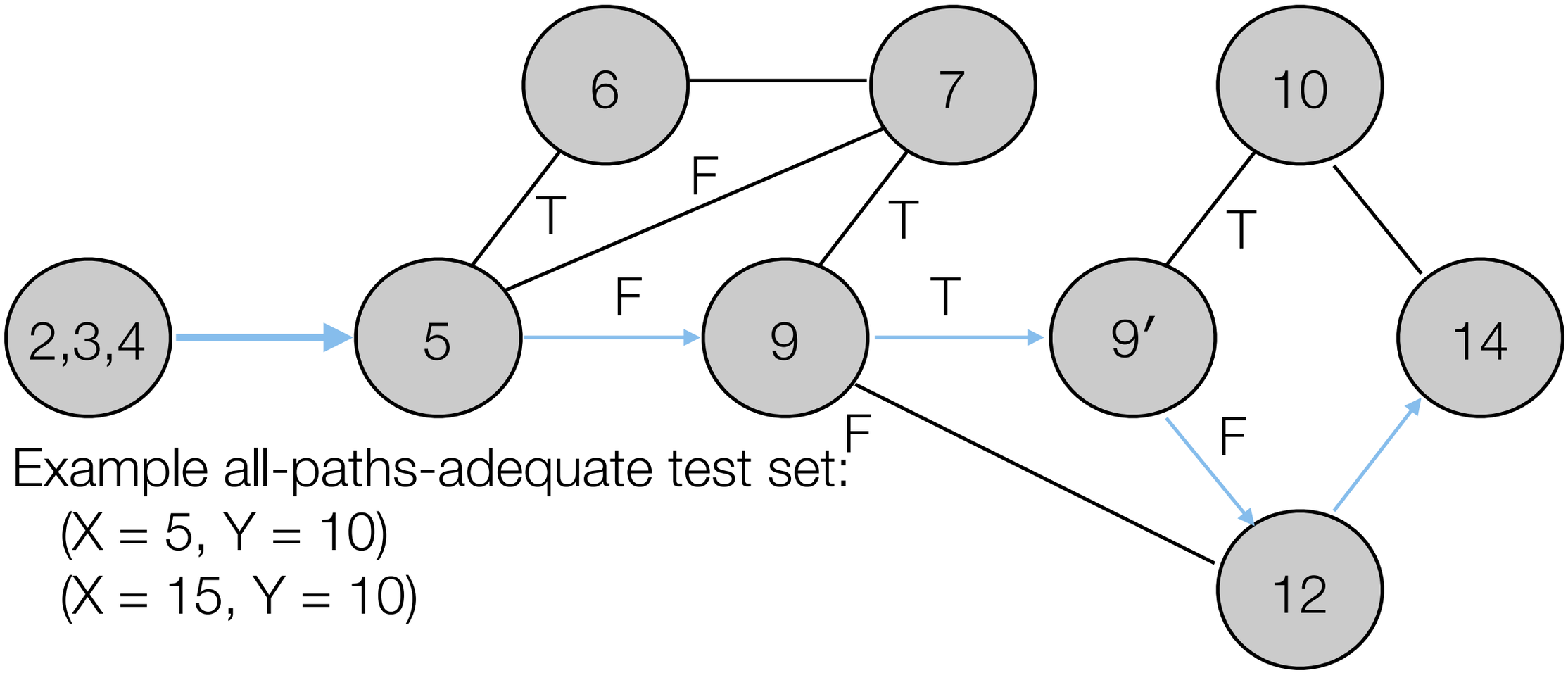
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



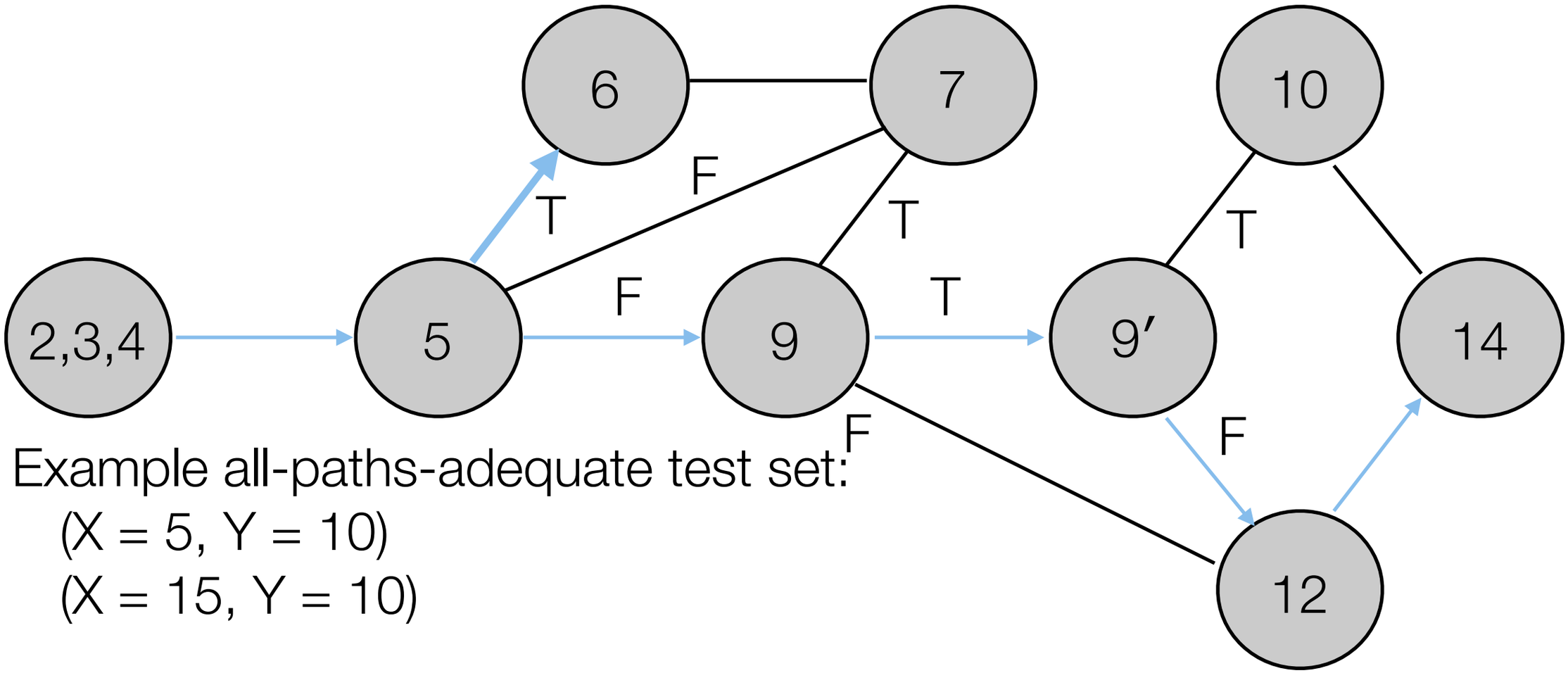
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



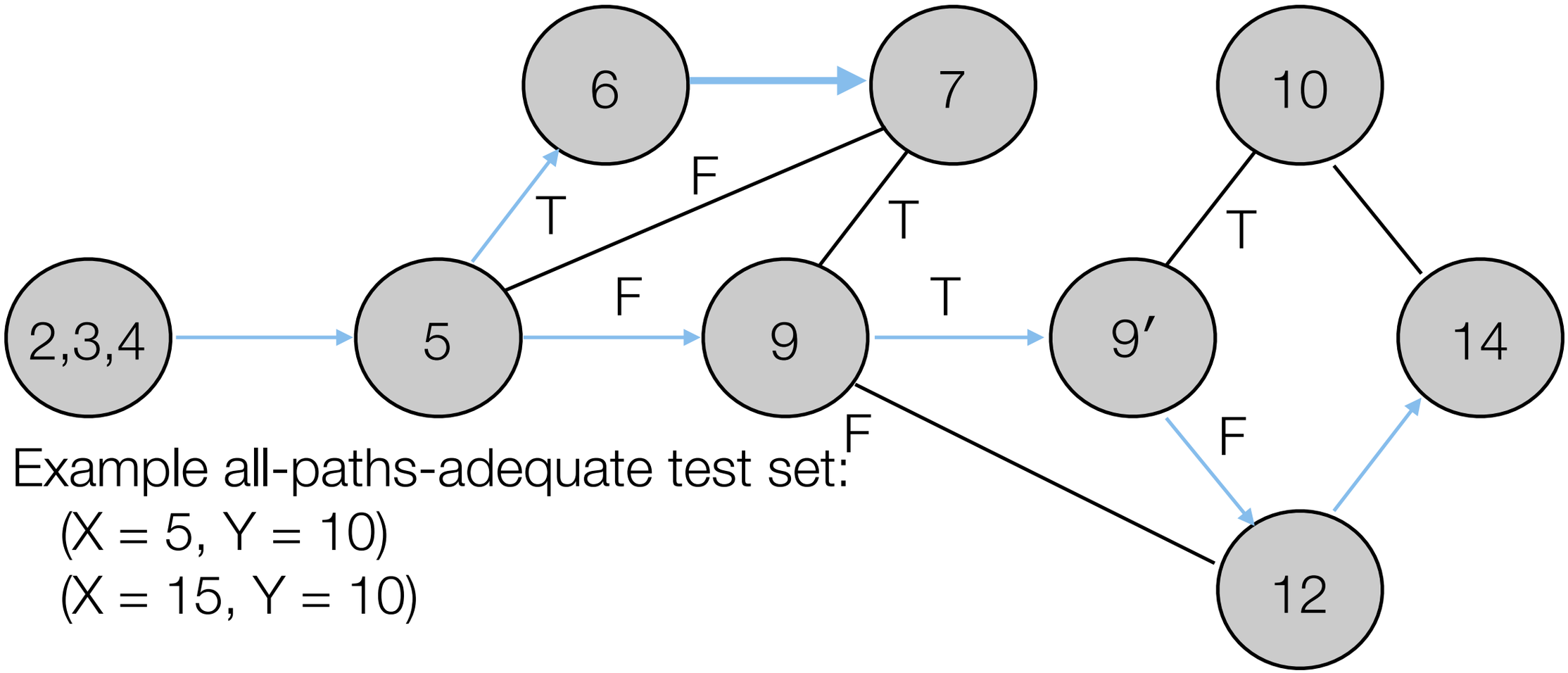
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



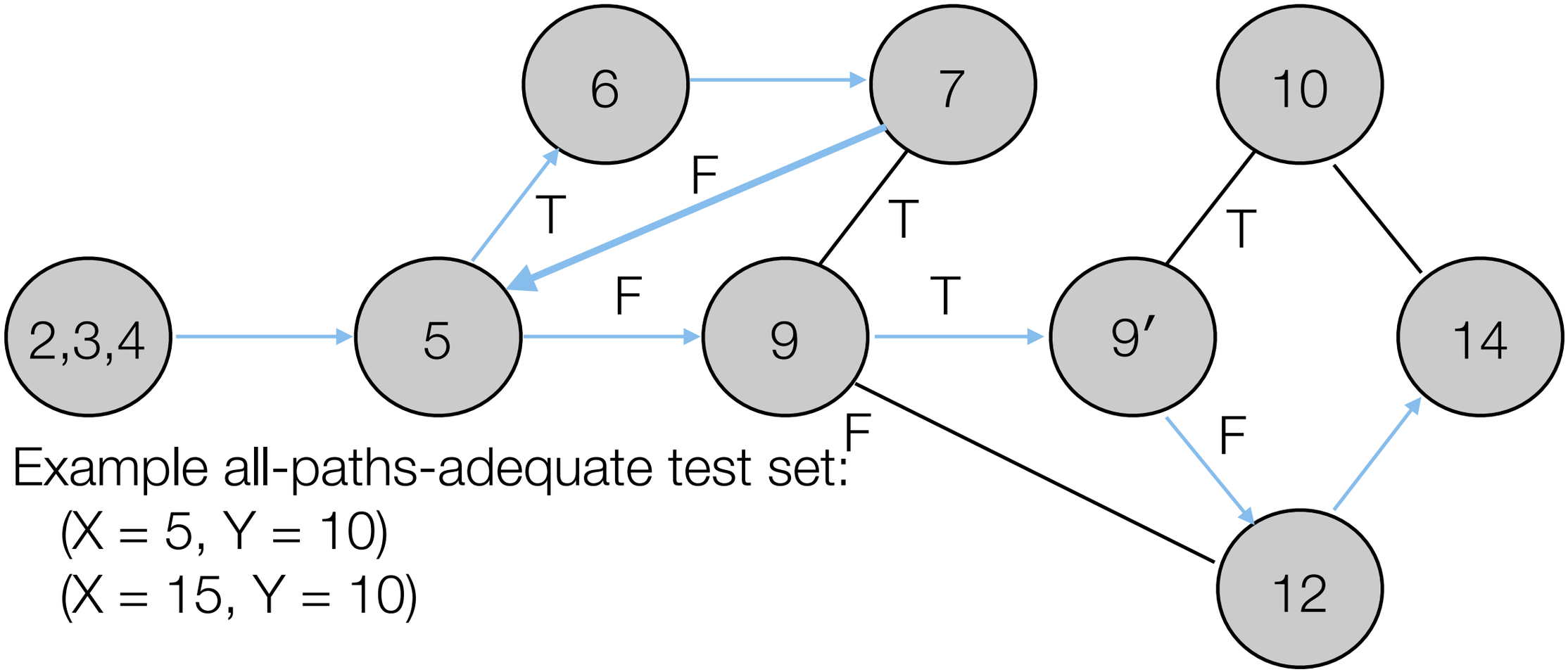
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



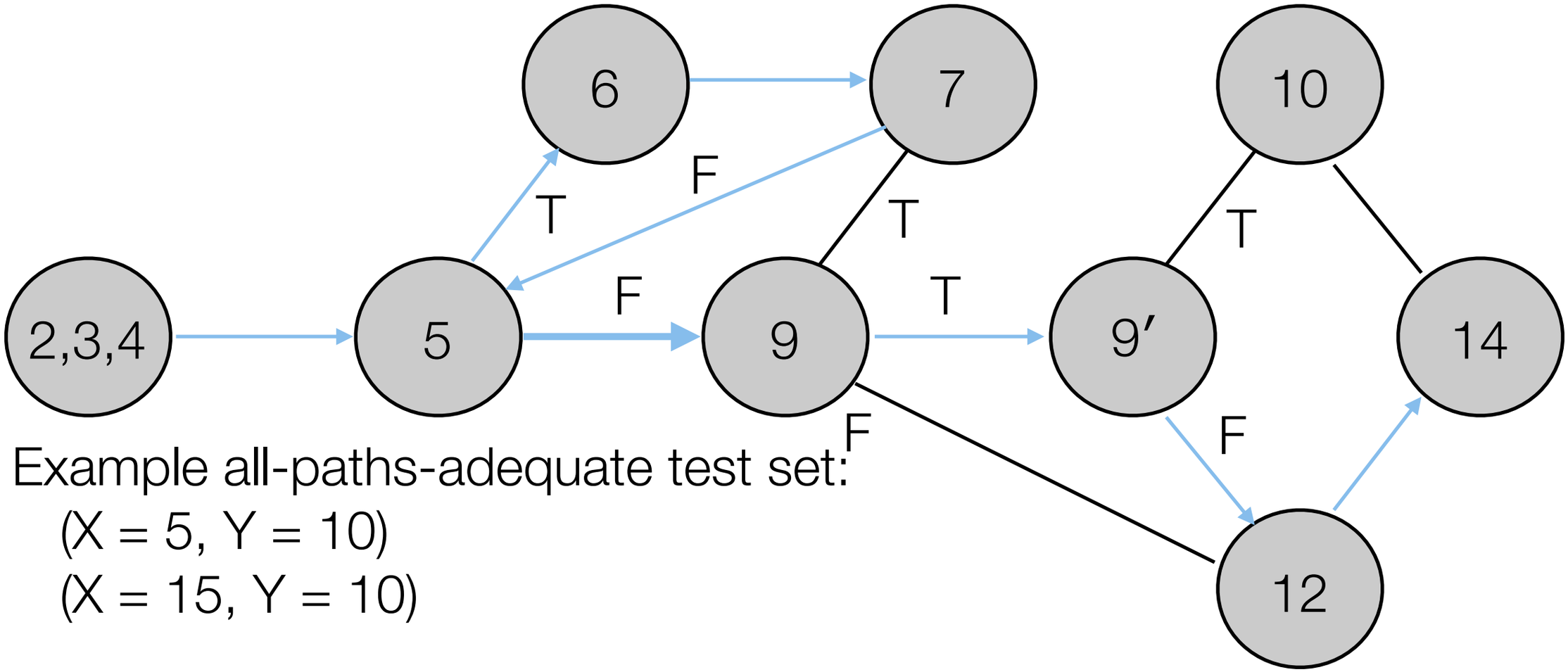
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



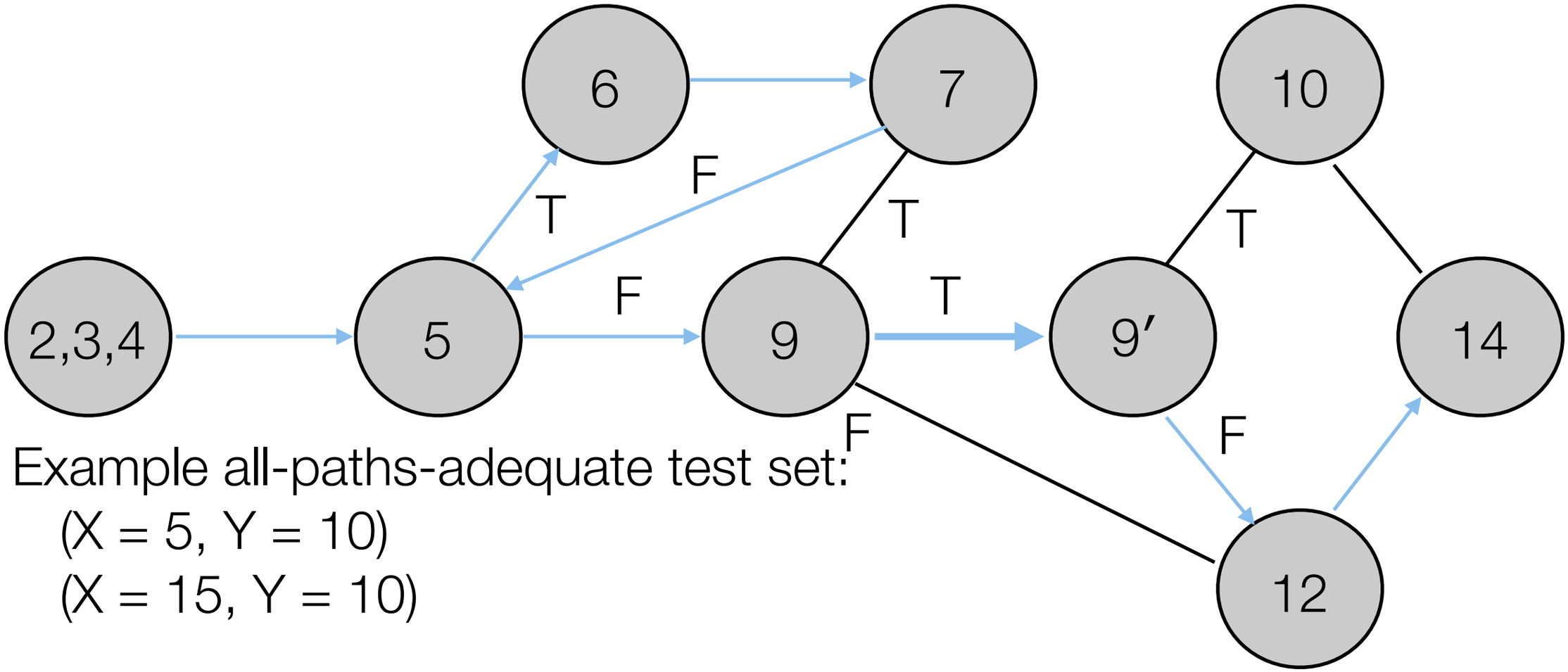
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



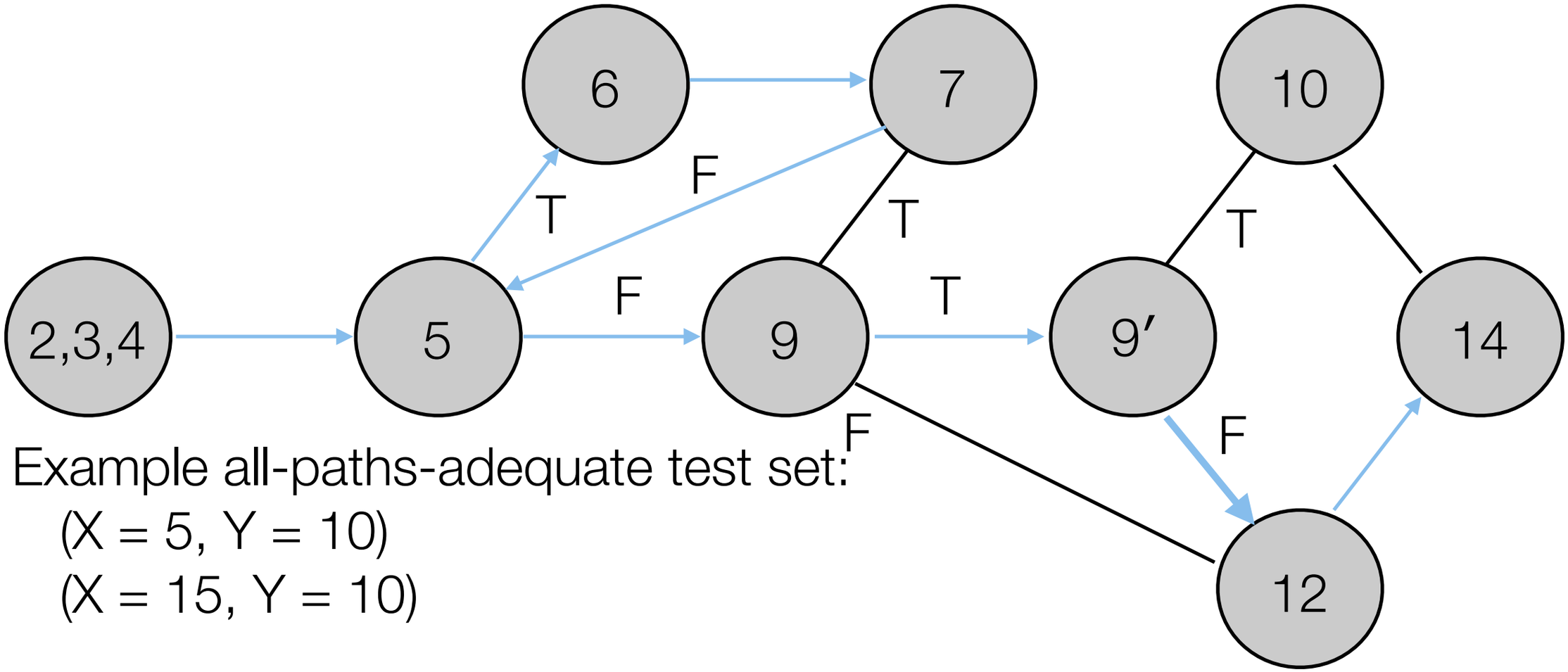
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



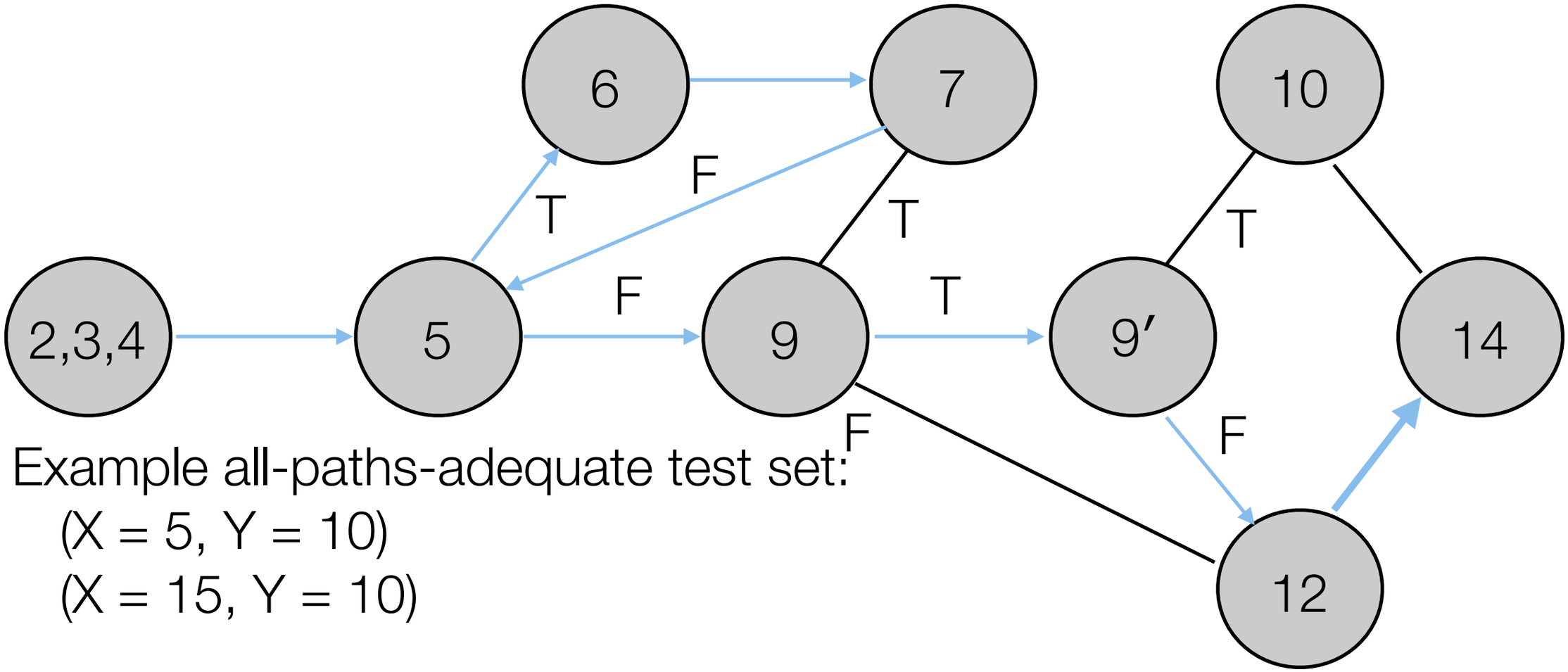
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



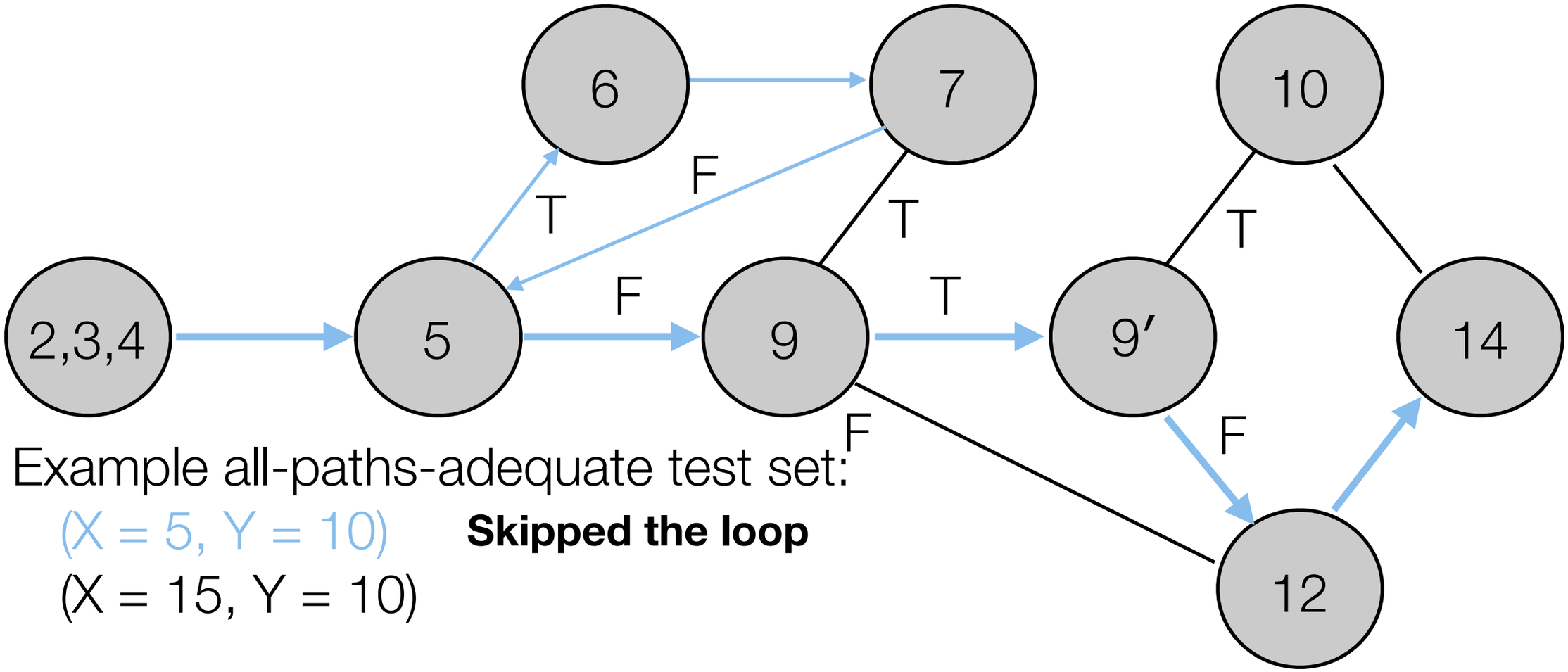
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



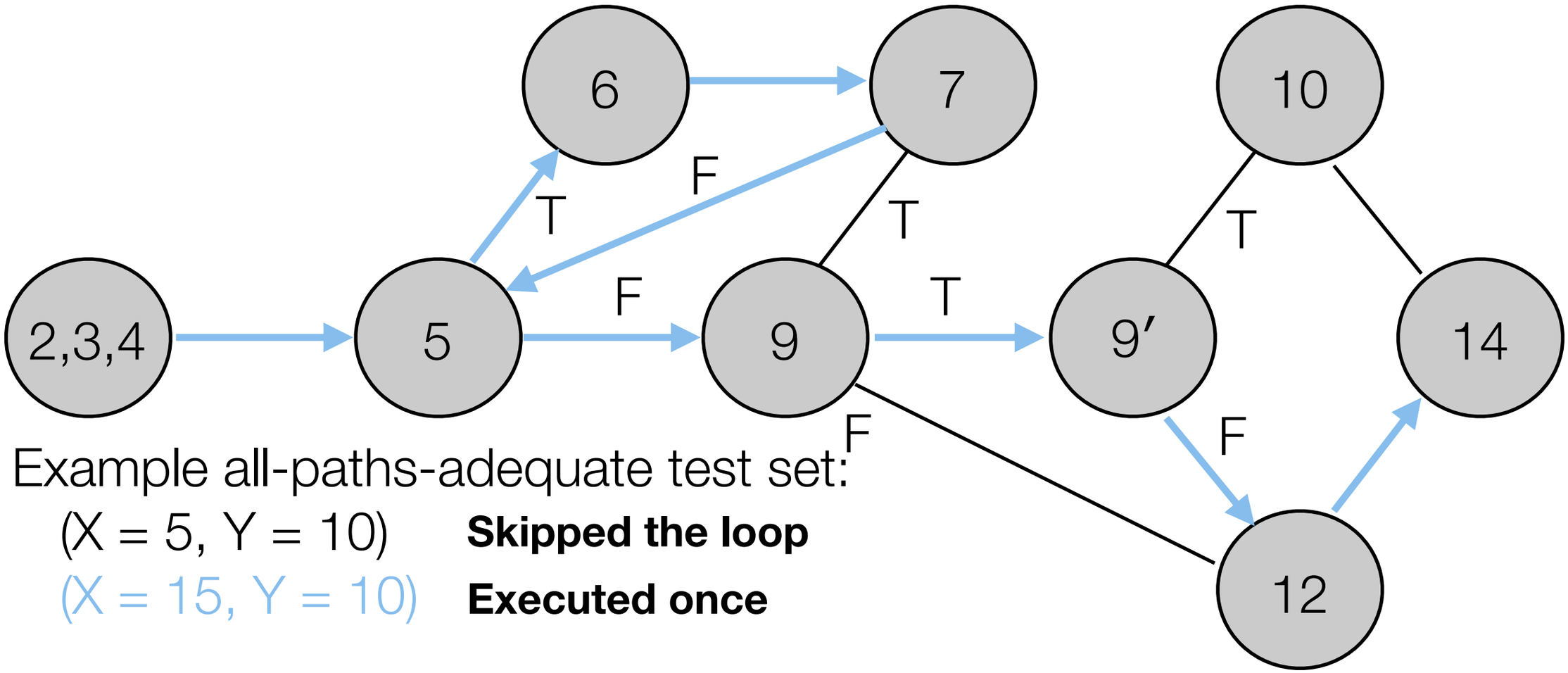
Example all-paths-adequate test set:

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

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



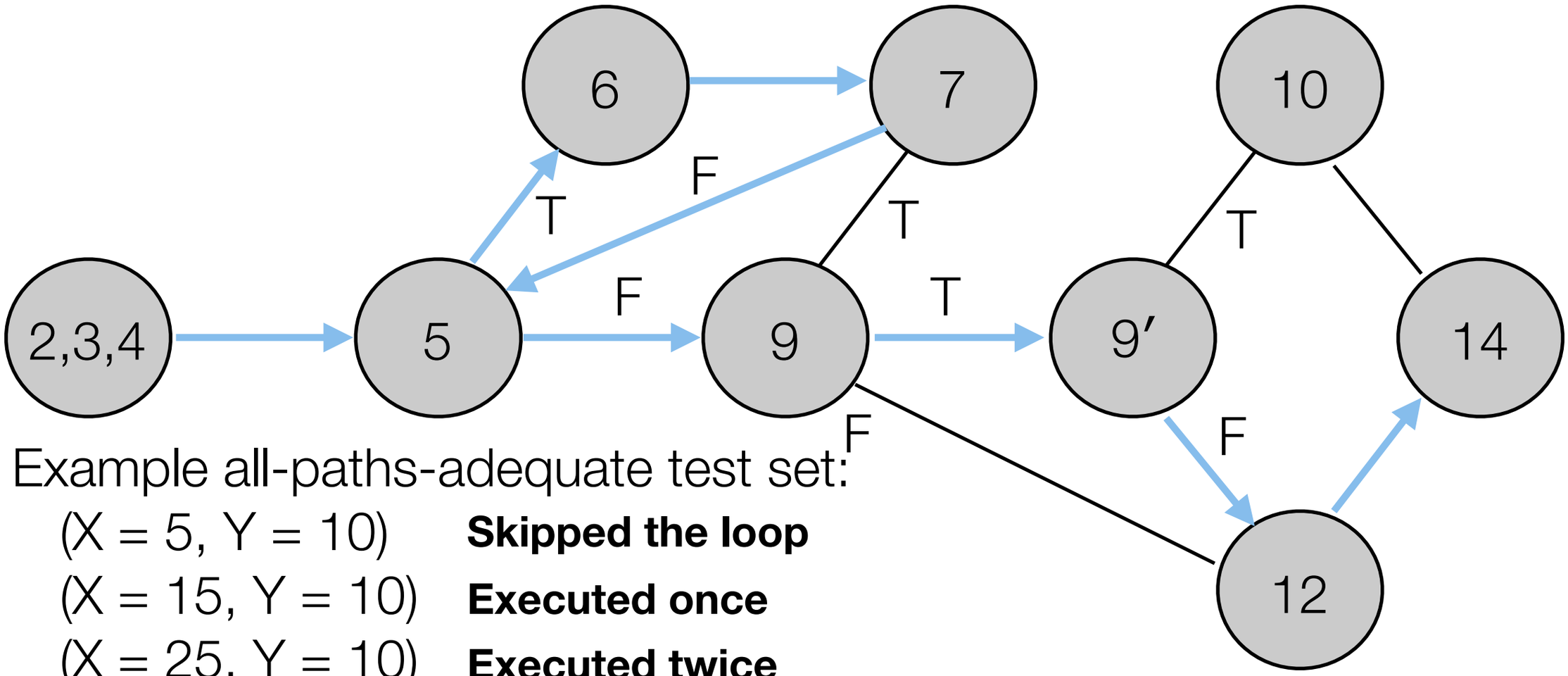
Example all-paths-adequate test set:

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

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

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



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**

And so on… you would also want permutations that exit the loop early

© Kenneth M. Anderson, 2012

Code Coverage Tools

- 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
    - These systems 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

© Kenneth M. Anderson, 2012

Testing Automation (I)

- 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!!)

© Kenneth M. Anderson, 2012

Testing Automation (II)

- 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
- We'll see examples of this later in the semester

© Kenneth M. Anderson, 2012

Continuous Integration

- Since test automation is so critical, systems known as continuous integration frameworks have emerged
- 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

© Kenneth M. Anderson, 2012

Summary

- 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 exists to integrate testing into the version control and build management processes of a development organization

© Kenneth M. Anderson, 2012

Coming Up Next:

- Lecture 6: Agile Methods and Agile Teams
- Lecture 7: Division of Labor and Design Approaches in Concurrency

© Kenneth M. Anderson, 2012

Introduction to Software Testing

CSCI 5828: Foundations of Software Engineering
Lecture 05 — 01/31/2012

© Kenneth M. Anderson, 2012

Goals

- Provide introduction to fundamental concepts of software testing
    - Terminology
    - Testing of Systems
        - unit tests, integration tests, system tests, acceptance tests
    - Testing of Code
        - Black Box
        - Gray Box
        - White Box
        - Code Coverage

© Kenneth M. Anderson, 2012

Testing

- Testing is a **critical element** of software development life cycles
  - called **software quality control** or **software quality assurance**
  - basic goals: **validation** and **verification**
    - validation: **are we building the right product?**
      - verification: **does "X" meet its specification?**
        - where "X" 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

© Kenneth M. Anderson, 2012

Terminology

- An **error** is a mistake made by an engineer
  - often a misunderstanding of a requirement or design specification
- A **fault** is a manifestation of that error in the code
  - what we often call “a bug”
- A **failure** is an incorrect output/behavior that is caused by executing a fault
  - The failure may occur immediately (crash!) or much, much later in the execution
- **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?

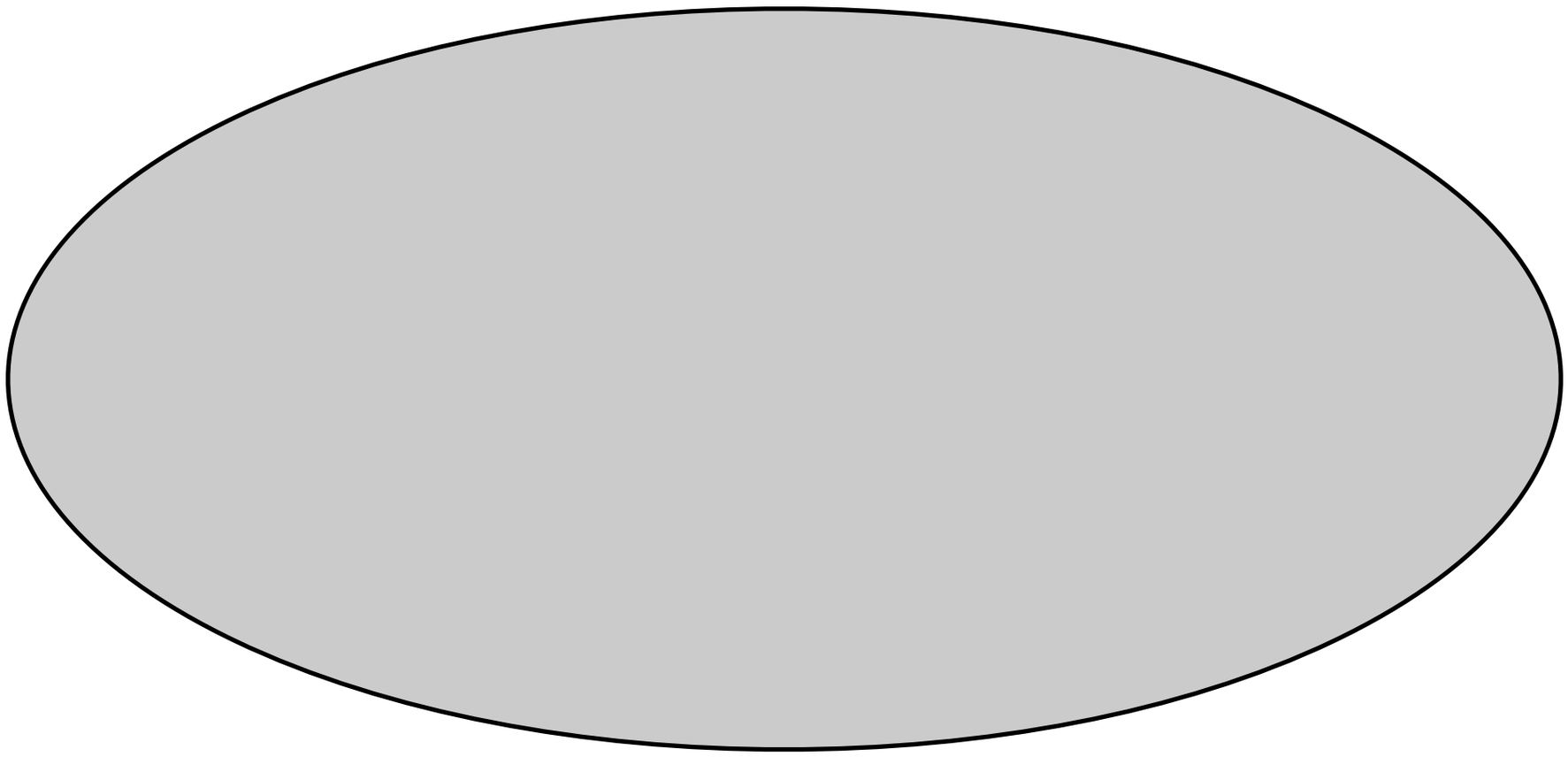
© Kenneth M. Anderson, 2012

No!

- 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 resulting in no visible failures!
      - this is particularly insidious
- However, if we do a good job in creating a test set that
  - covers all functional capabilities of a system
  - and 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

© Kenneth M. Anderson, 2012

Looking for Faults

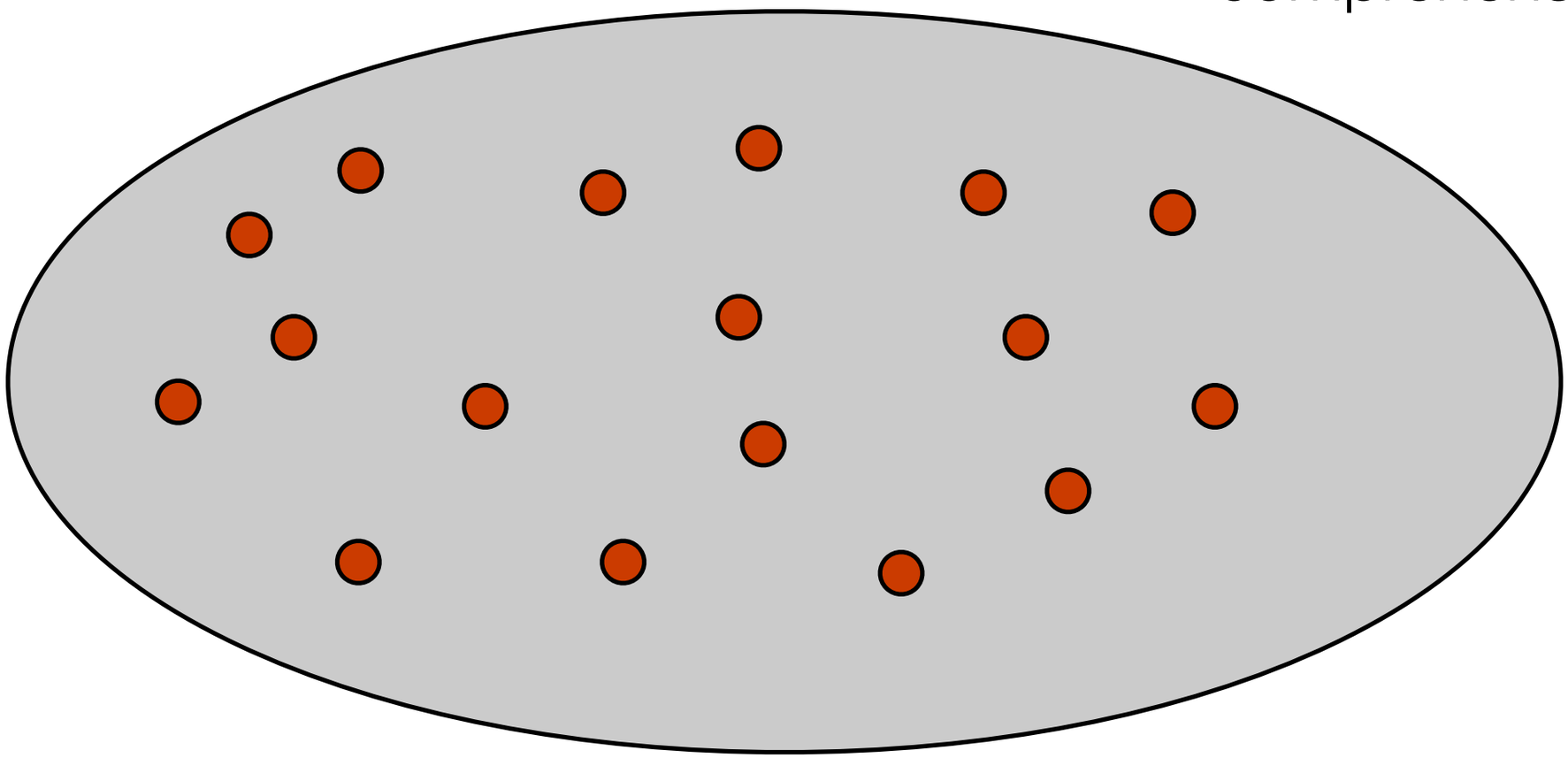


All possible states/behaviors of a system

© Kenneth M. Anderson, 2012

Looking for Faults

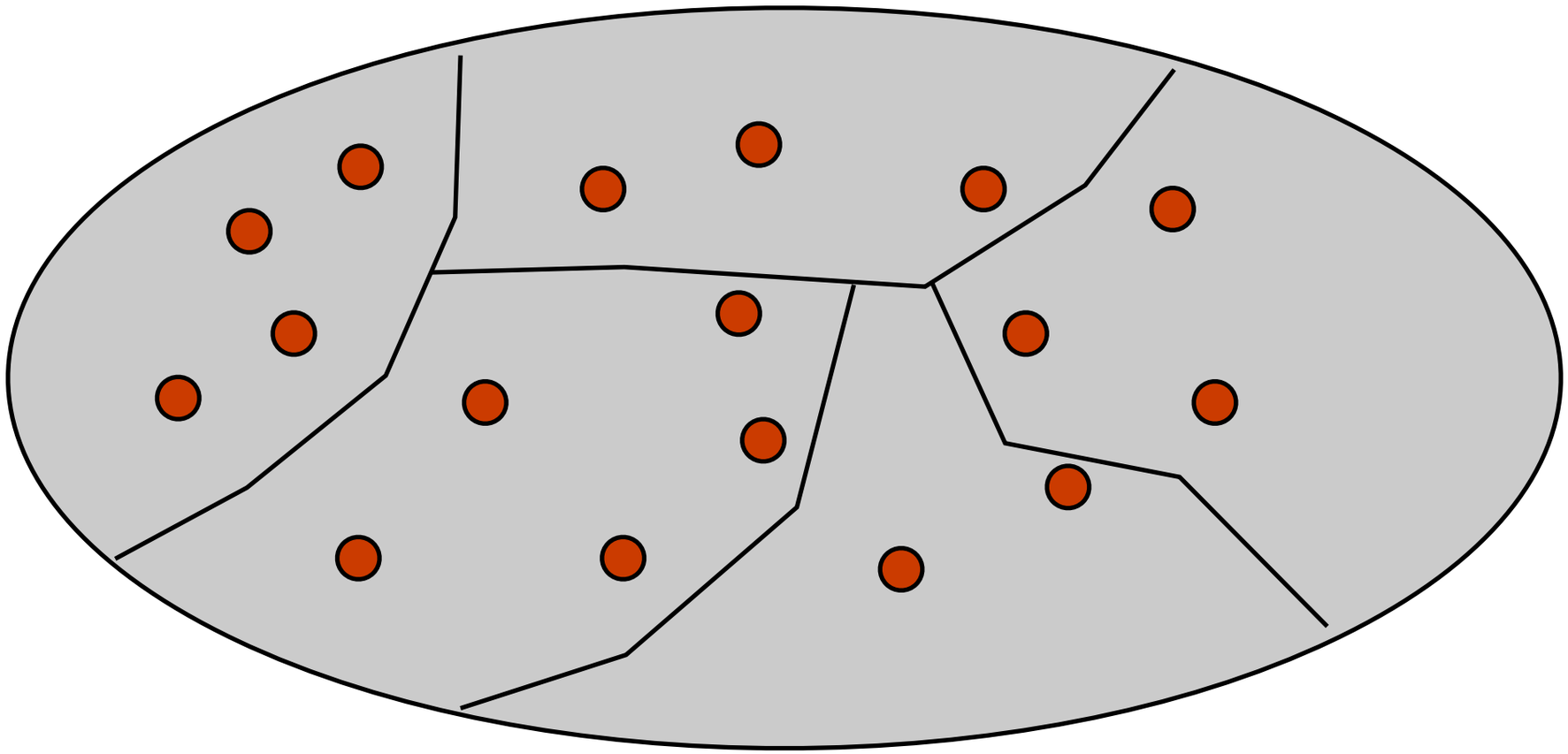
As you can see, its not very comprehensive



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

© Kenneth M. Anderson, 2012

One way forward? Fold



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

© Kenneth M. Anderson, 2012

What does that mean?

- 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

© Kenneth M. Anderson, 2012

Completeness

- 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?

© Kenneth M. Anderson, 2012

Continuous Testing

- Testing is a continuous process that should be performed at every stage of a software development process
    - During requirements gathering, for instance, we must continually query the user, “Did we get this right?”
    - Facilitated by an emphasis on iteration throughout a life cycle
- at the end of each iteration
    - we check our results to see if what we built is meeting our requirements (specification)

© Kenneth M. Anderson, 2012

Testing the System (I)

- **Unit Tests**
  - Tests that cover low-level aspects of a system
    - For each module, does each operation perform as expected
    - For method **foo()**, we'd like to see another method **testFoo()**
- **Integration Tests**
  - Tests that check that modules work together in combination
  - Most projects on schedule until they hit this point (MMM, Brooks)
    - 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 (Mars Polar Lander)

© Kenneth M. Anderson, 2012

Testing the System (II)

- **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

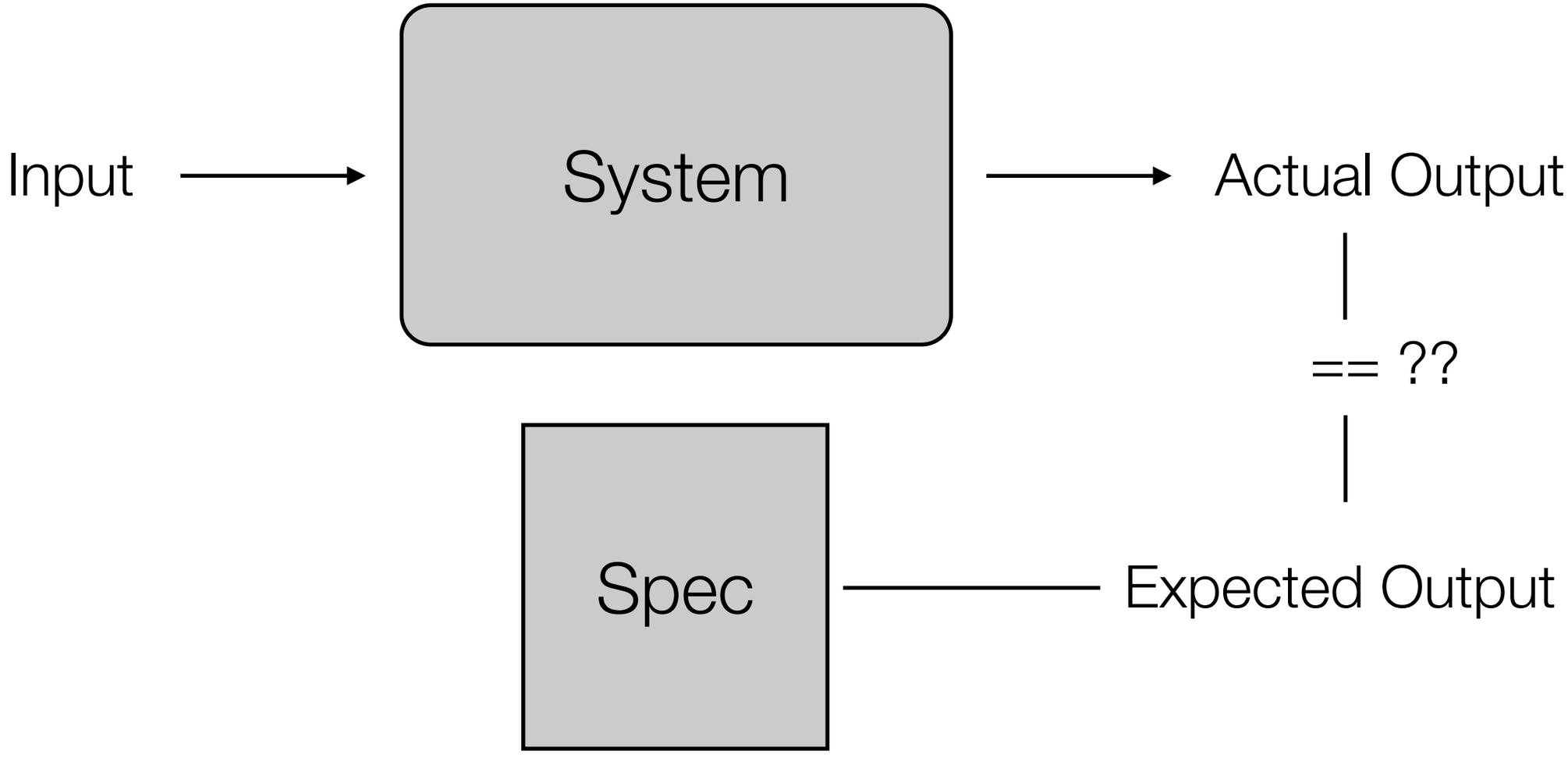
© Kenneth M. Anderson, 2012

Multi-Level Testing

- 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, …

© Kenneth M. Anderson, 2012

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!

© Kenneth M. Anderson, 2012

Results

- 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

© Kenneth M. Anderson, 2012

Black Box Categories

- 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

© Kenneth M. Anderson, 2012

Grey Box Testing

- 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

© Kenneth M. Anderson, 2012

White Box Testing

- 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?

© Kenneth M. Anderson, 2012

Code Coverage (I)

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

© Kenneth M. Anderson, 2012

Code Coverage (II)

- 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)

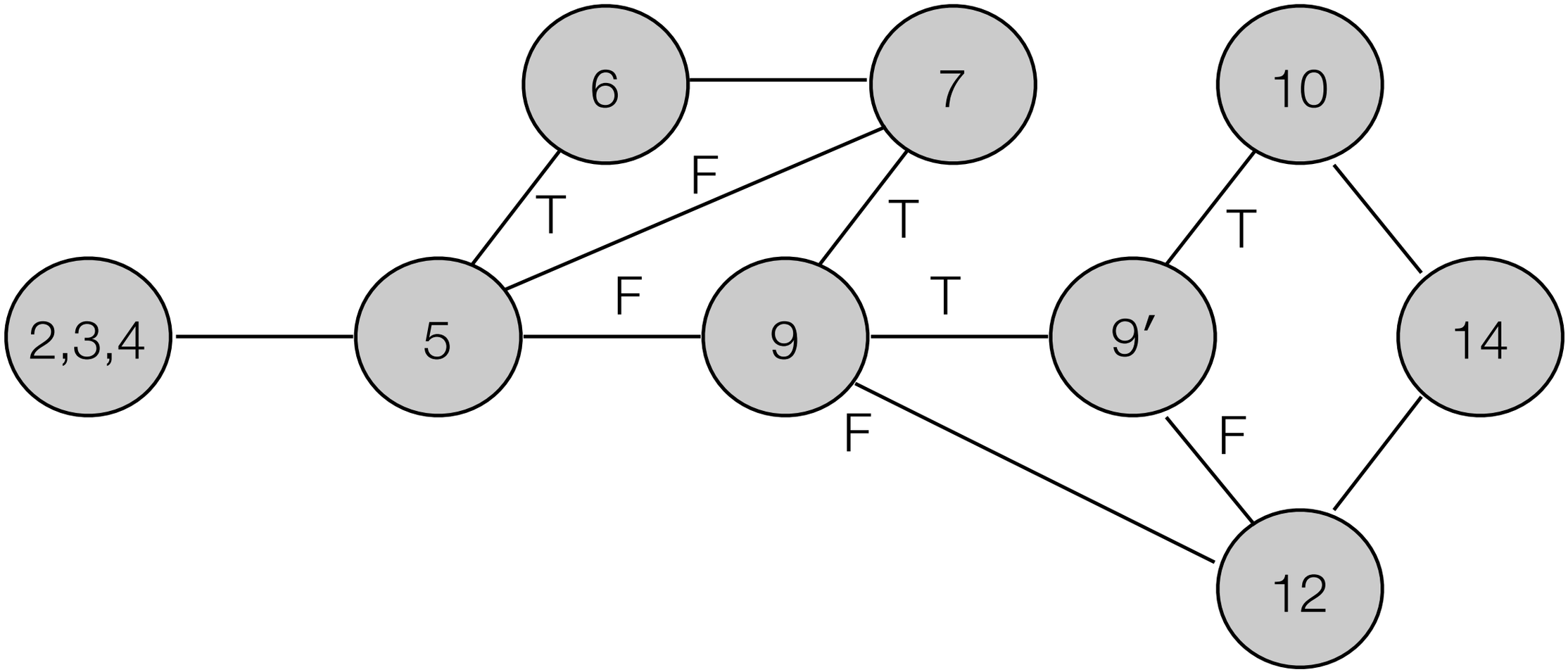
© Kenneth M. Anderson, 2012

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;
```

© Kenneth M. Anderson, 2012

P’s Control Flow Graph (CFG)



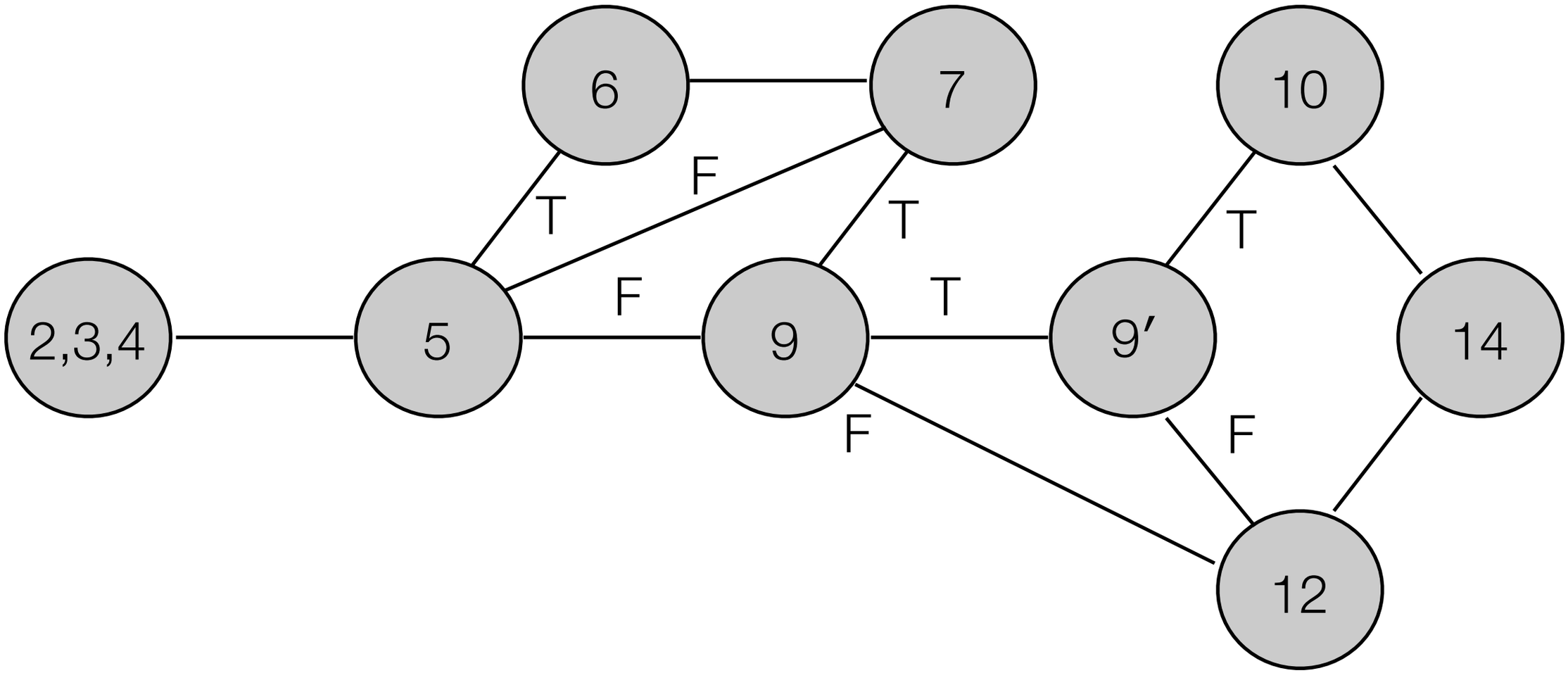
© Kenneth M. Anderson, 2012

White-box Testing Criteria

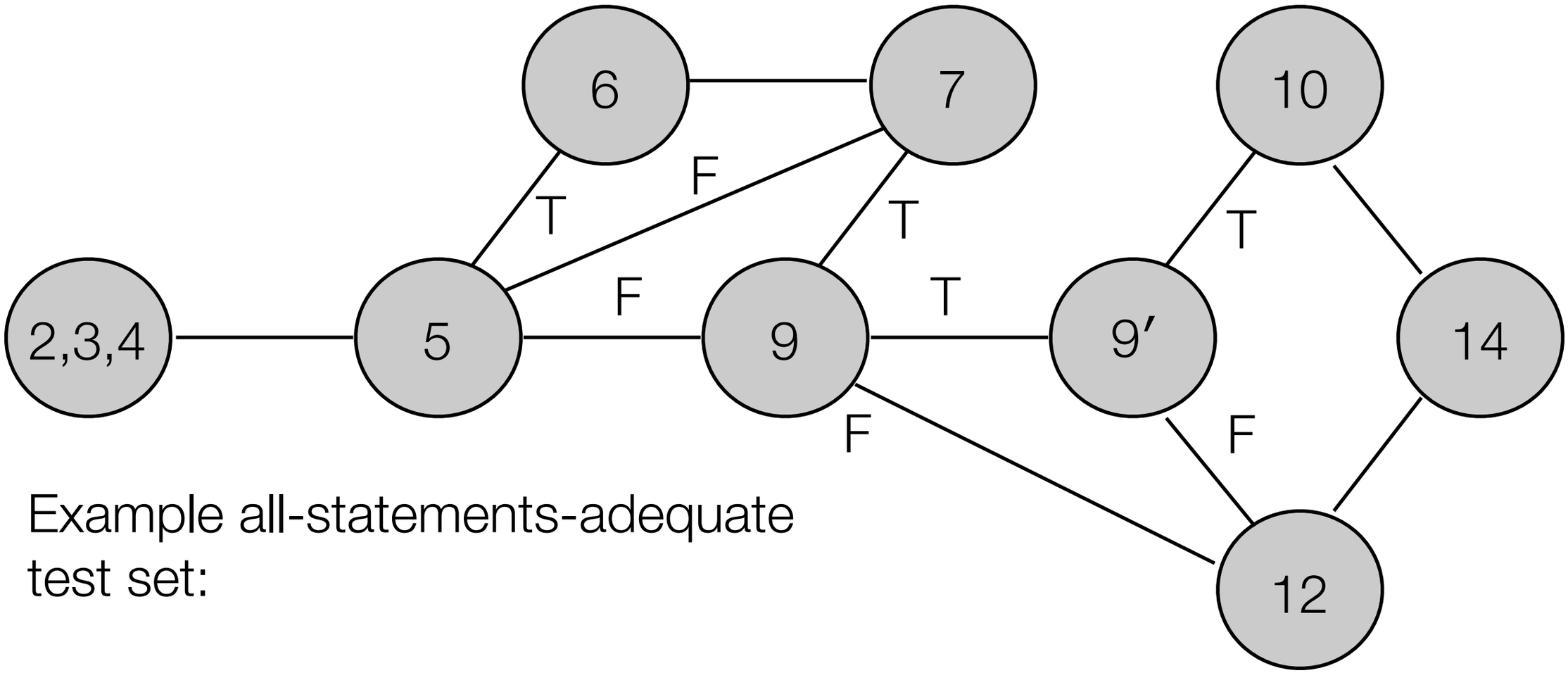
- Statement Coverage
  - Create a test set T such that
    - by executing P for each t in T
      - each elementary statement of P is executed at least once

© Kenneth M. Anderson, 2012

All-Statements Coverage of P

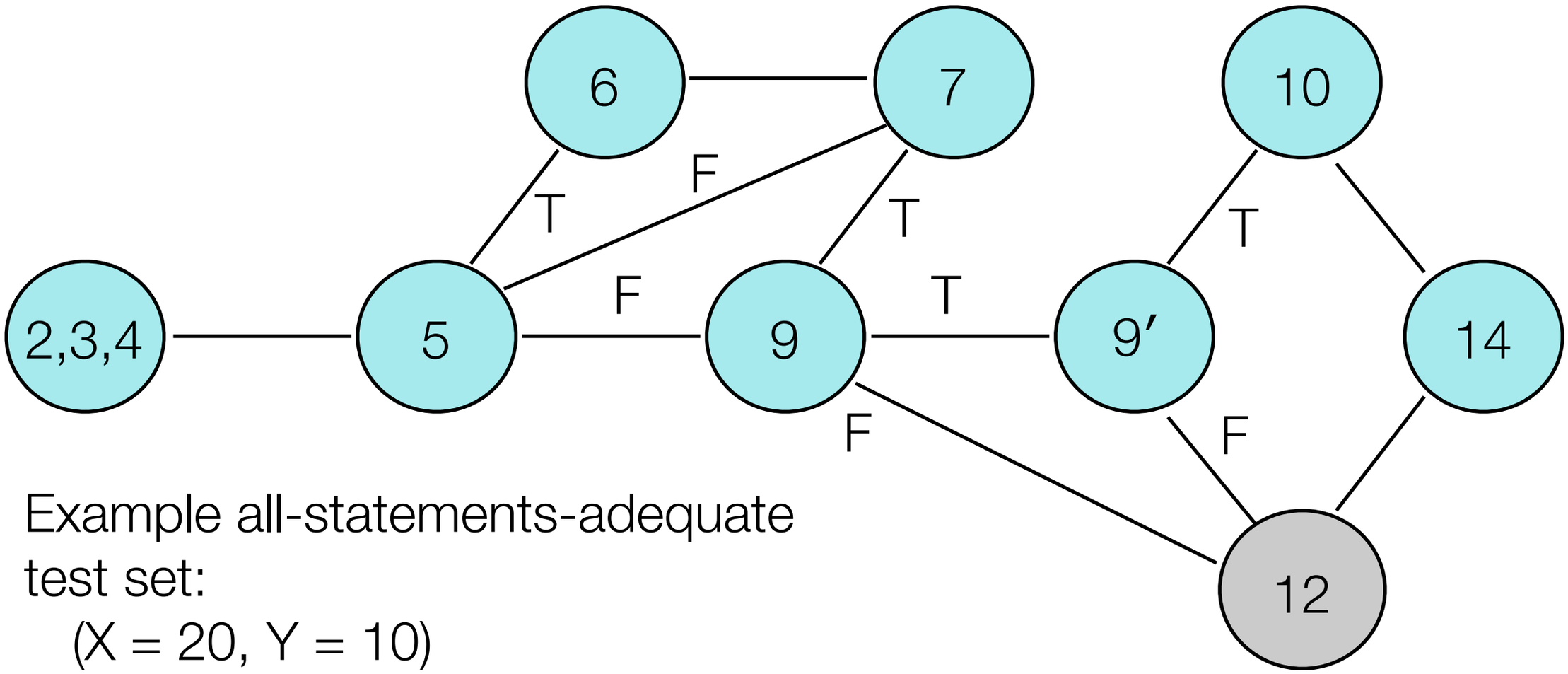


All-Statements Coverage of P



Example all-statements-adequate test set:

All-Statements Coverage of P

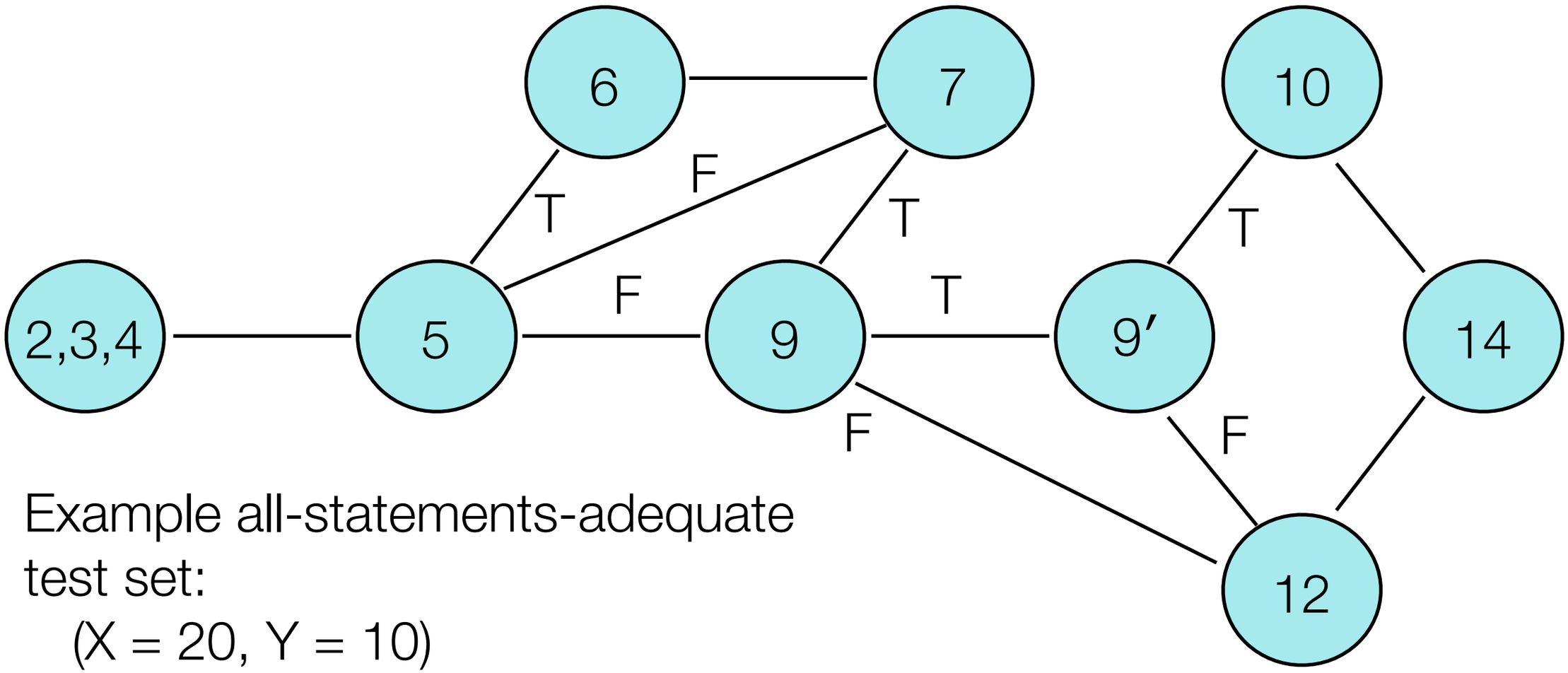


Example all-statements-adequate test set:

(X = 20, Y = 10)

© Kenneth M. Anderson, 2012

All-Statements Coverage of P



Example all-statements-adequate test set:

(X = 20, Y = 10)

(X = 20, Y = 30)

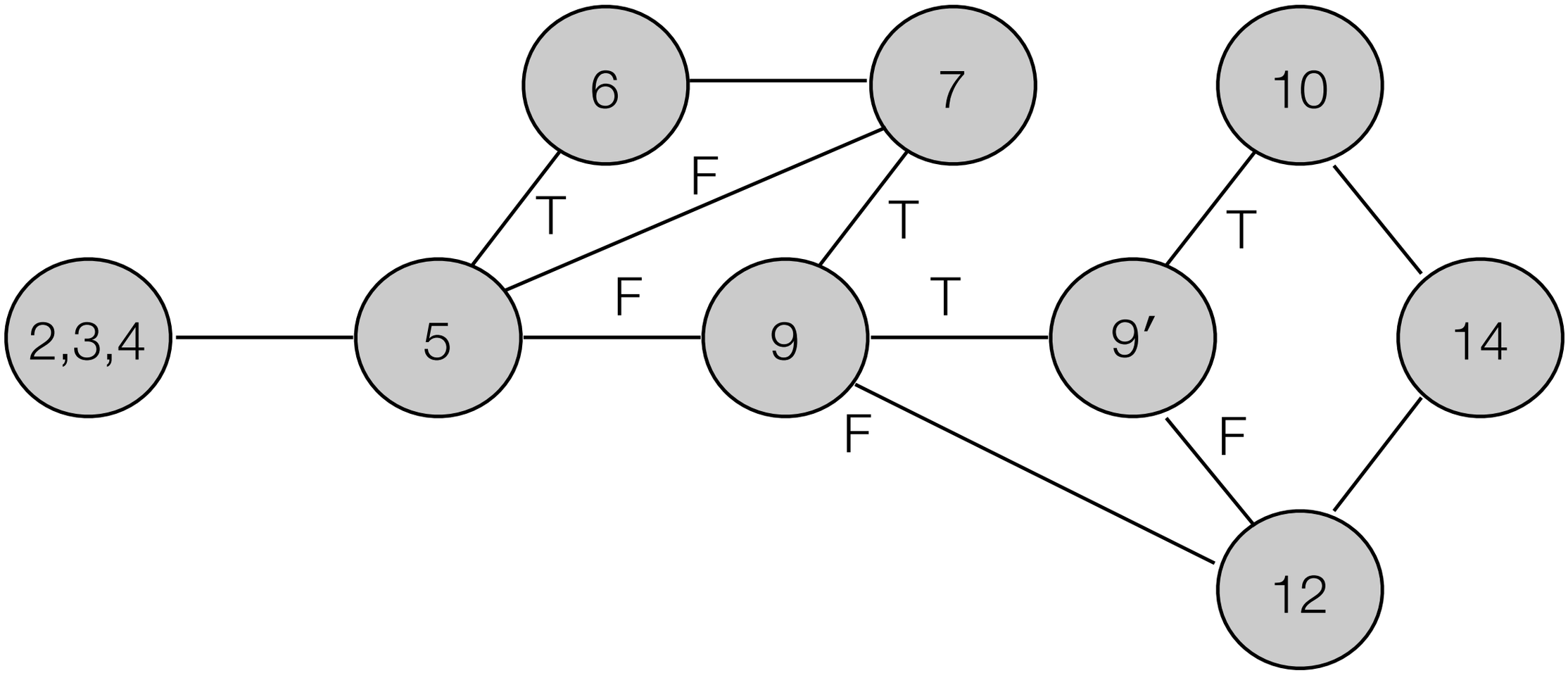
© Kenneth M. Anderson, 2012

White-box Testing Criteria

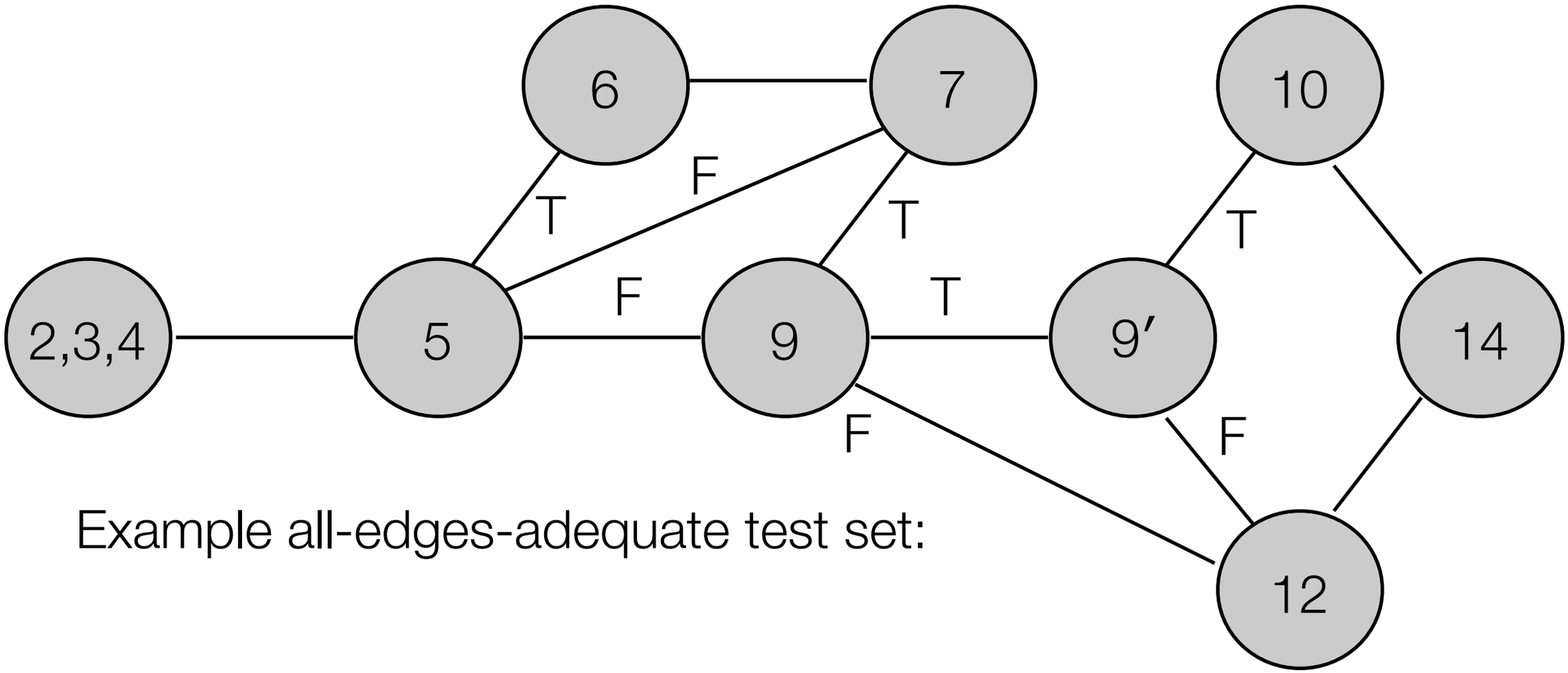
- Edge Coverage
  - Select a test set T such that
    - by executing P for each t in T
      - each edge of P's control flow graph is traversed at least once

© Kenneth M. Anderson, 2012

All-Edges Coverage of P



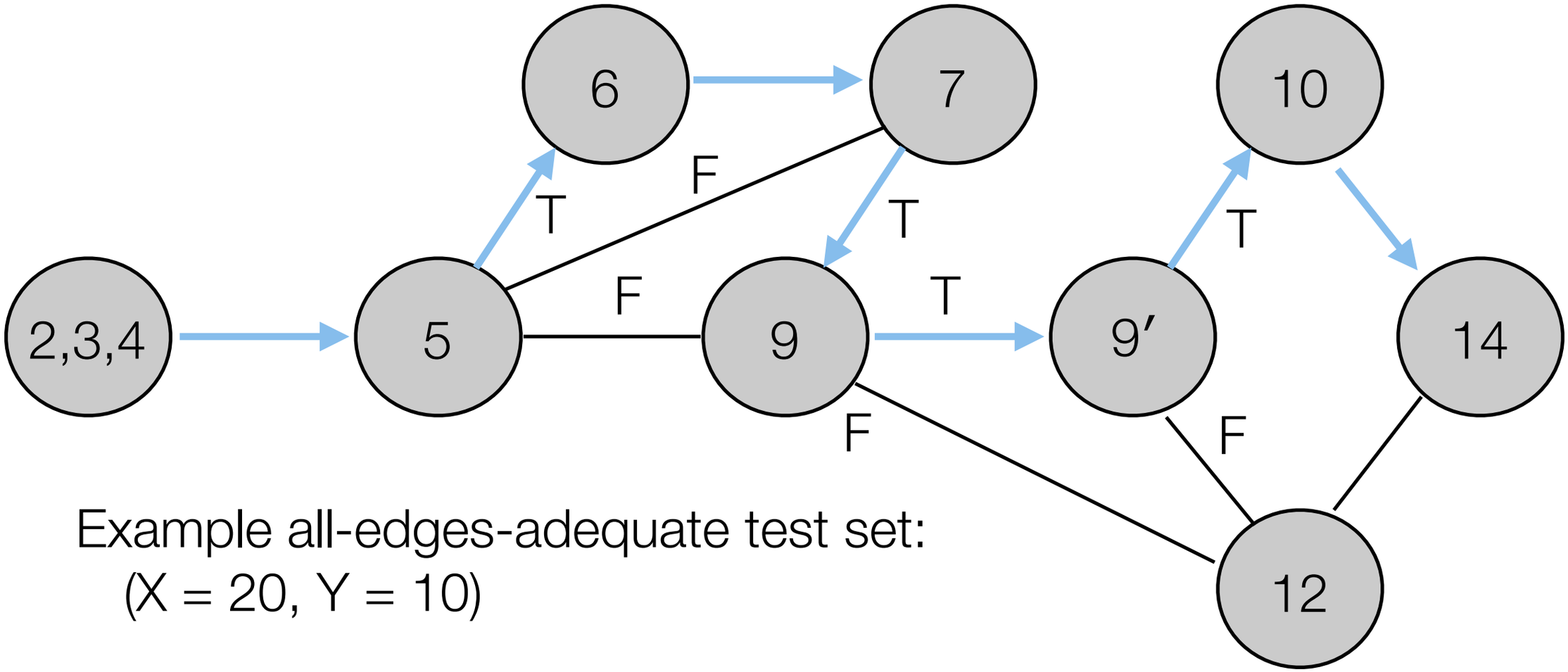
All-Edges Coverage of P



Example all-edges-adequate test set:

© Kenneth M. Anderson, 2012

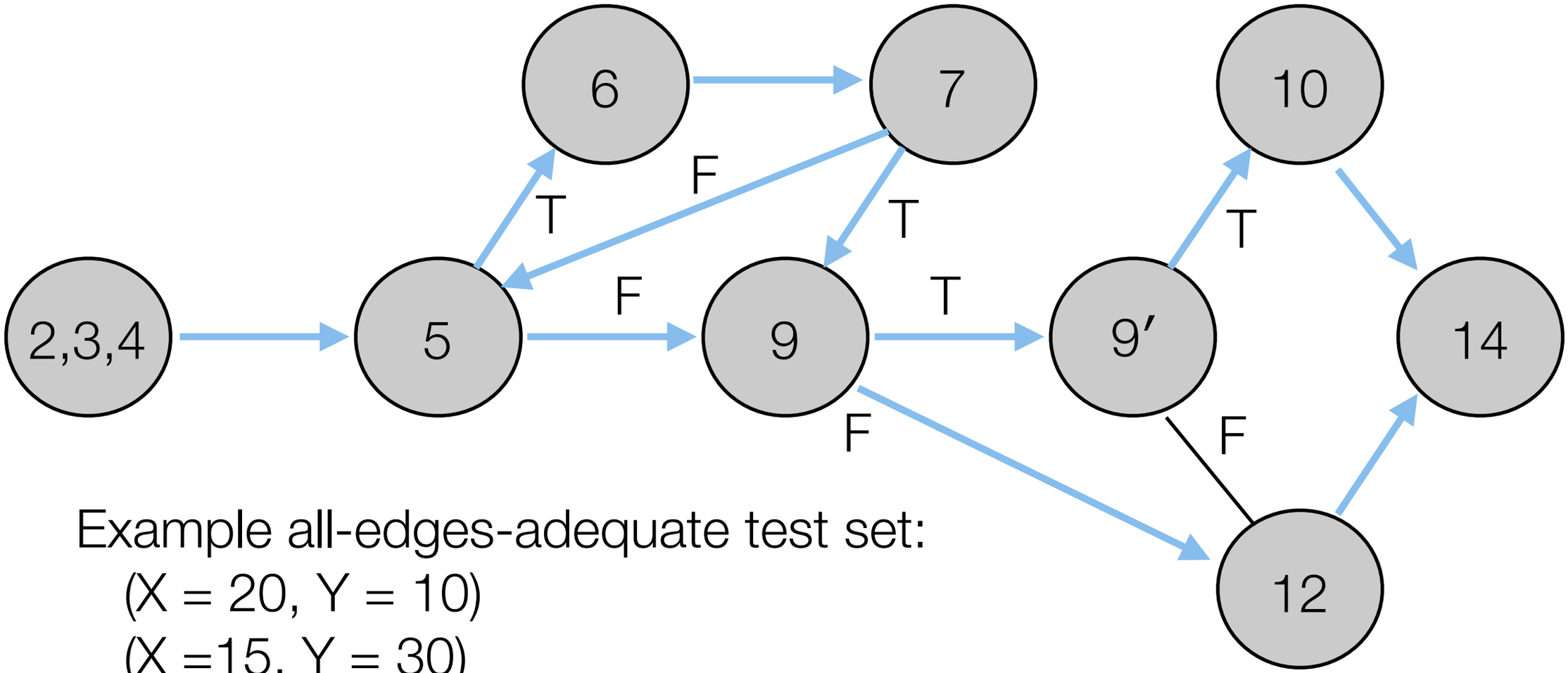
All-Edges Coverage of P



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

© Kenneth M. Anderson, 2012

All-Edges Coverage of P



Example all-edges-adequate test set:

(X = 20, Y = 10)

(X =15, Y = 30)

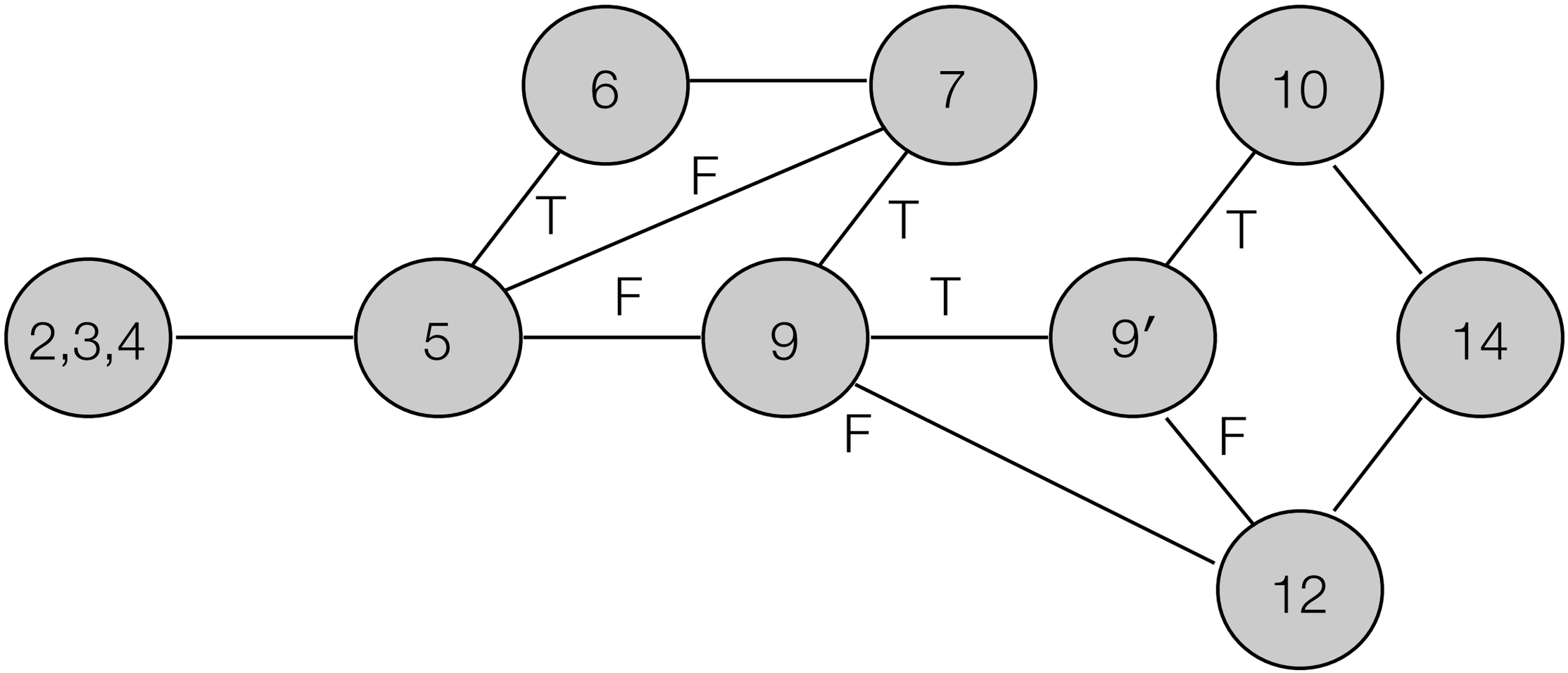
© Kenneth M. Anderson, 2012

White-box Testing Criteria

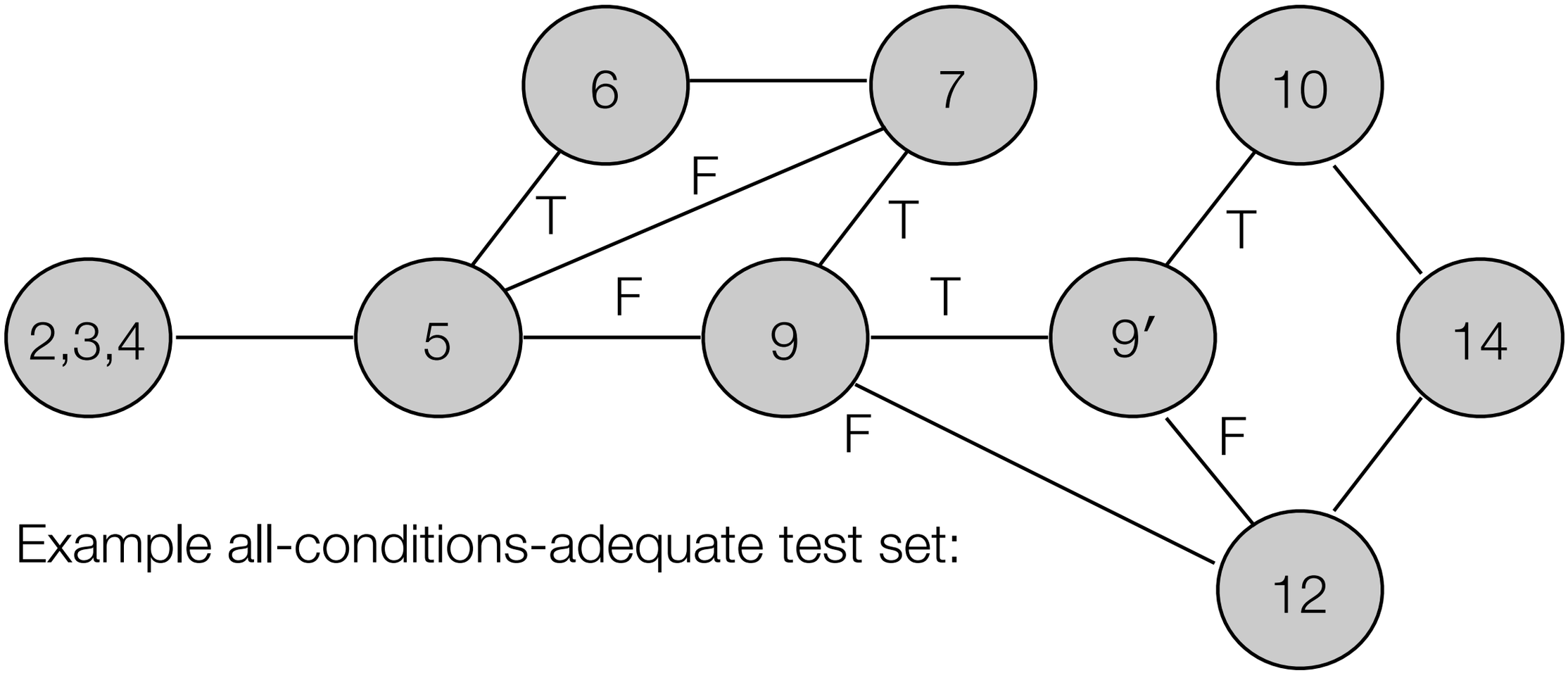
- Condition Coverage
  - Select a test set T such that
    - by executing P for each t in T
      - each edge of P's control flow graph is traversed at least once
      - and all possible values of the constituents of compound conditions are exercised at least once

© Kenneth M. Anderson, 2012

All-Conditions Coverage of P



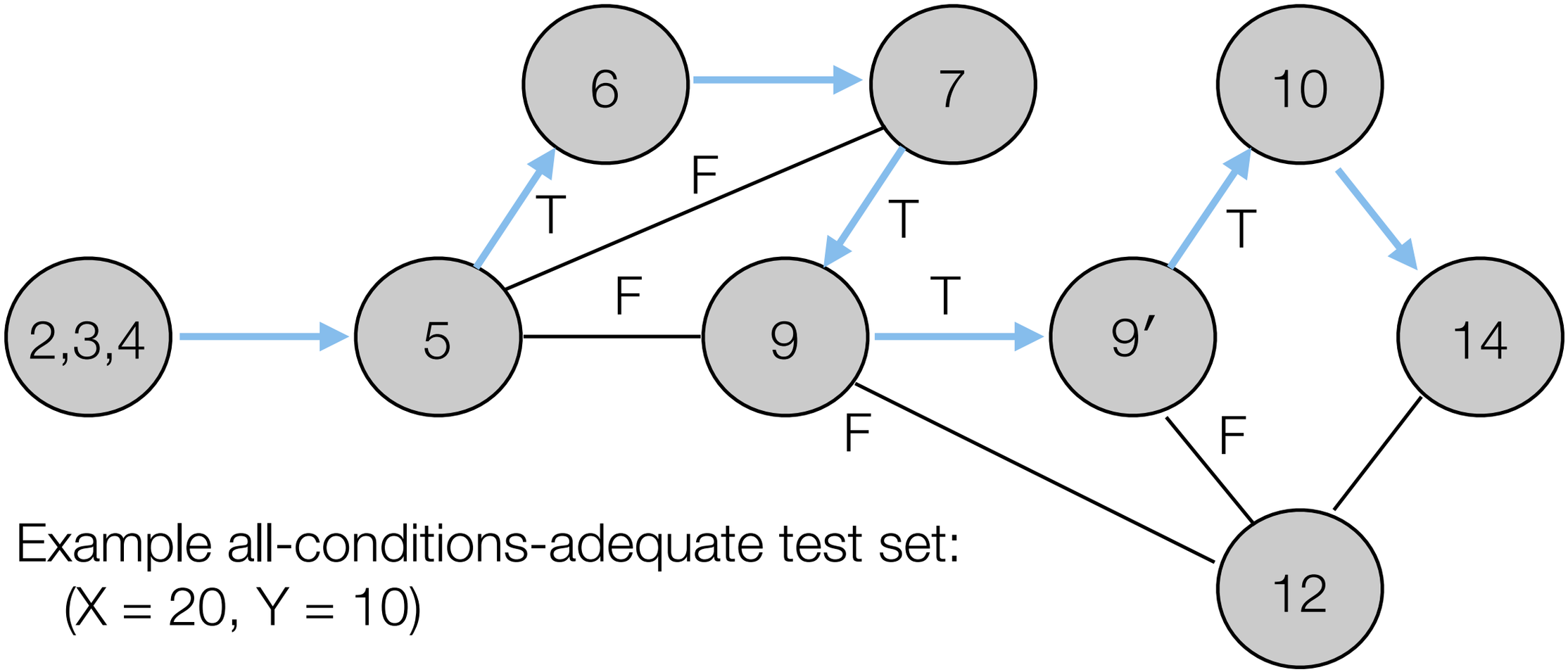
All-Conditions Coverage of P



Example all-conditions-adequate test set:

© Kenneth M. Anderson, 2012

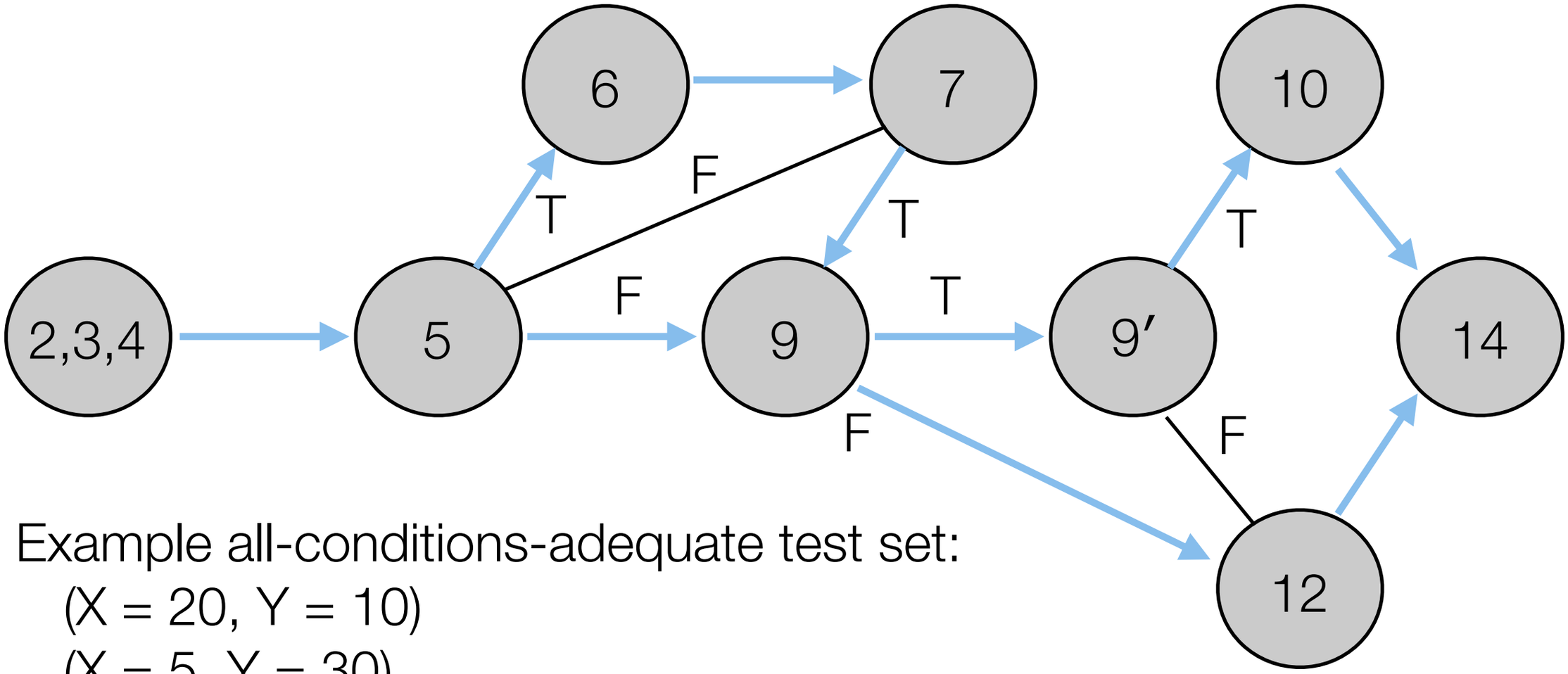
All-Conditions Coverage of P



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

© Kenneth M. Anderson, 2012

All-Conditions Coverage of P



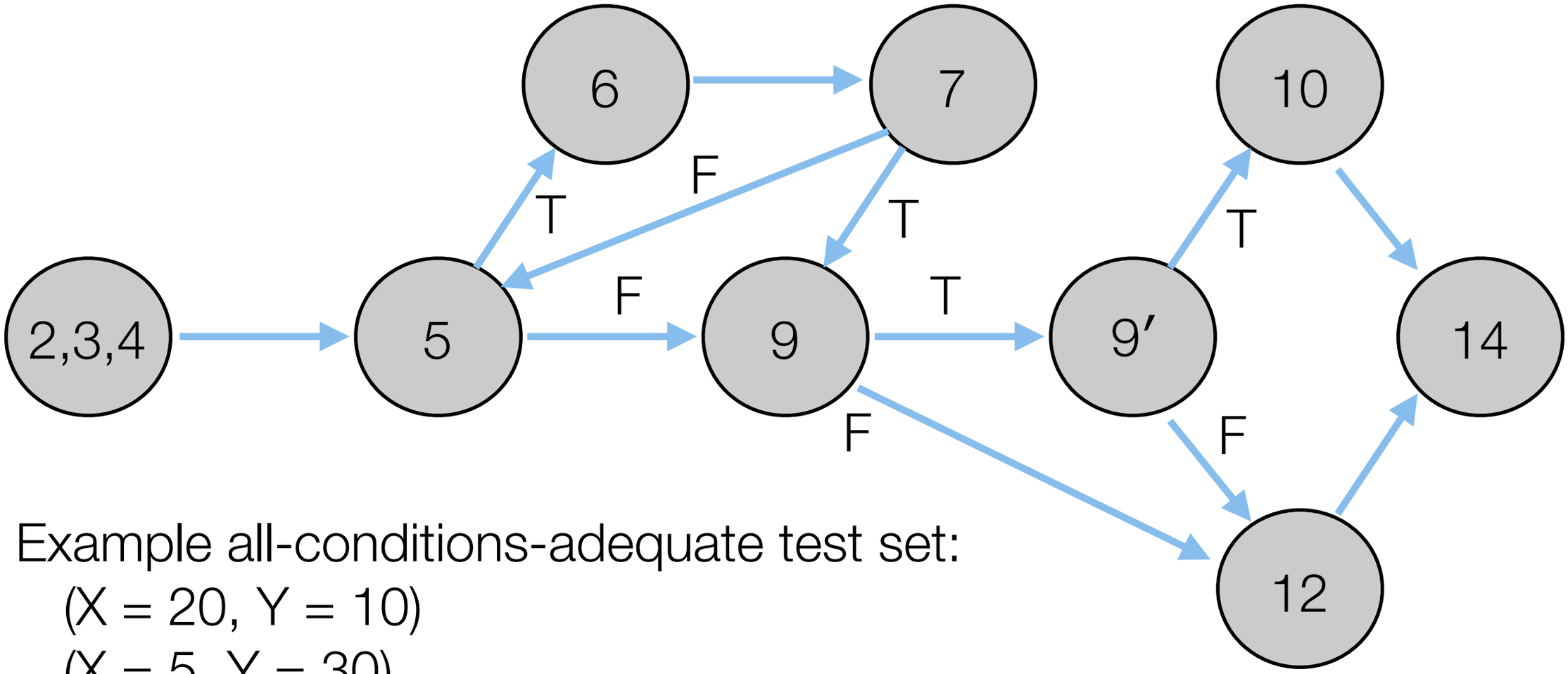
Example all-conditions-adequate test set:

(X = 20, Y = 10)

(X = 5, Y = 30)

© Kenneth M. Anderson, 2012

All-Conditions Coverage of P



Example all-conditions-adequate test set:

(X = 20, Y = 10)

(X = 5, Y = 30)

(X = 21, Y = 10)

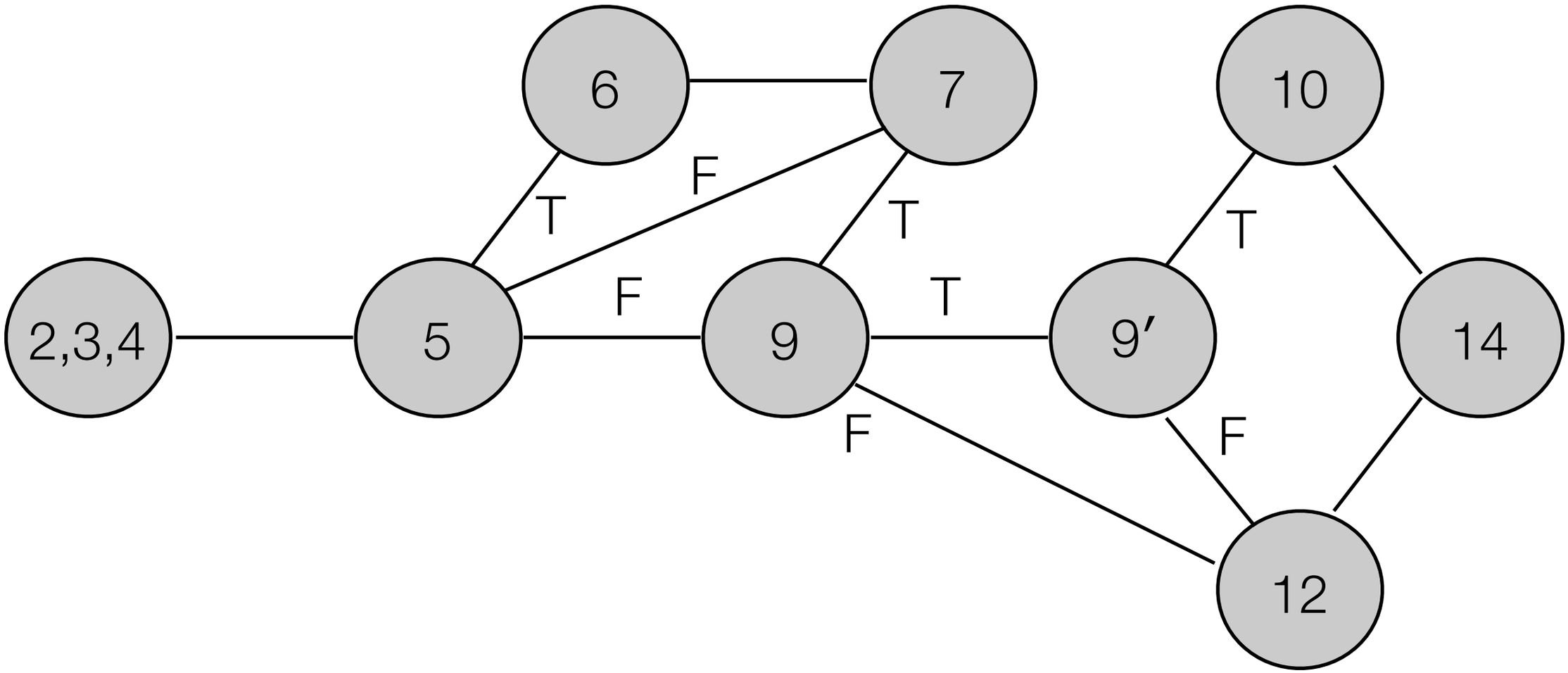
© Kenneth M. Anderson, 2012

White-box Testing Criteria

- 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

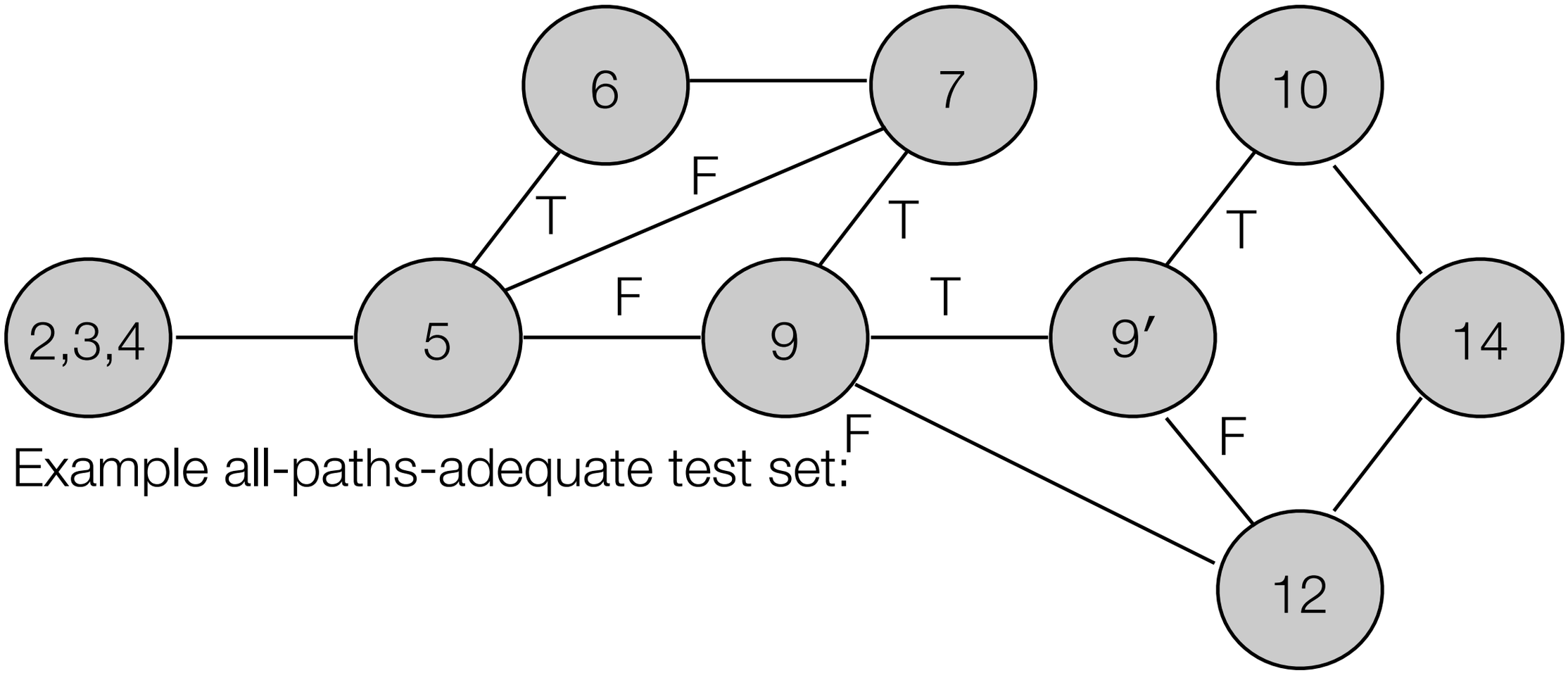
© Kenneth M. Anderson, 2012

All-Paths Coverage of P



© Kenneth M. Anderson, 2012

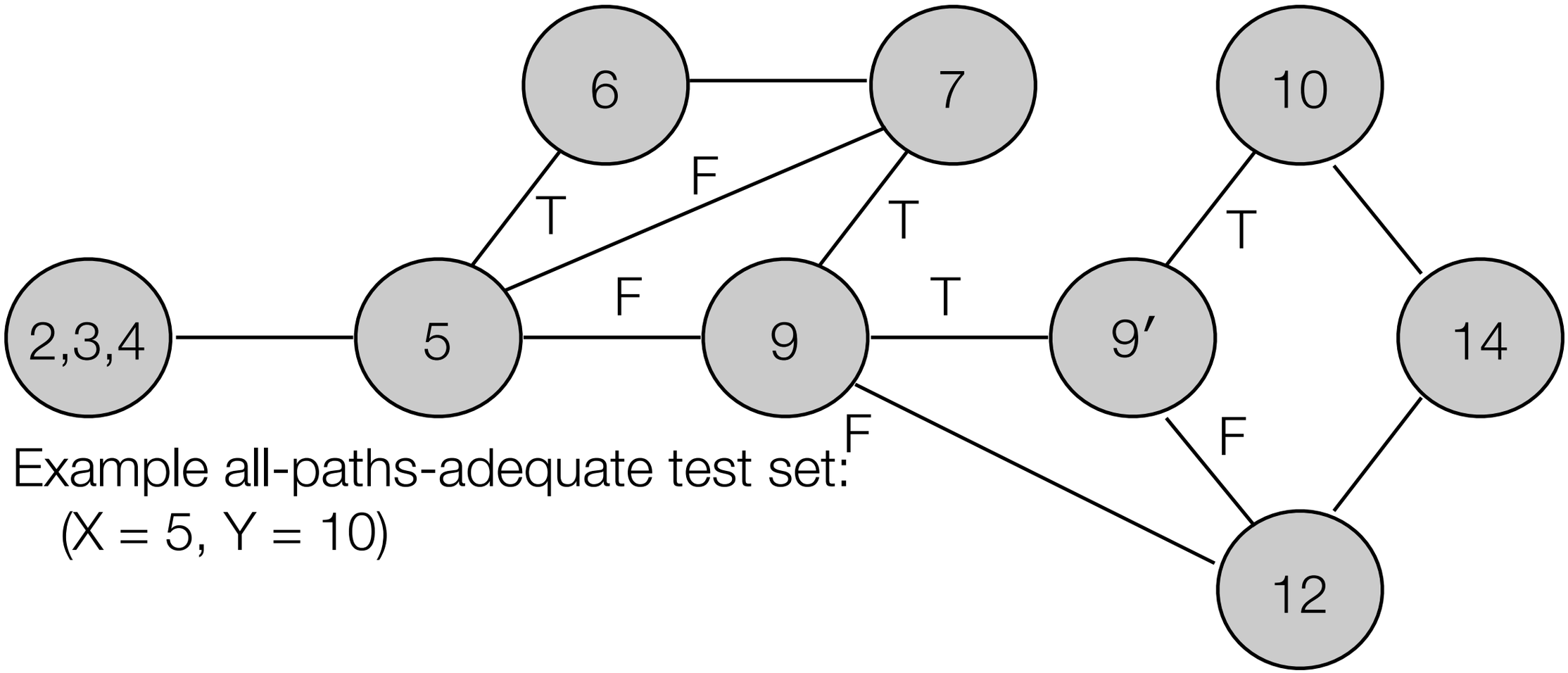
All-Paths Coverage of P



Example all-paths-adequate test set:

© Kenneth M. Anderson, 2012

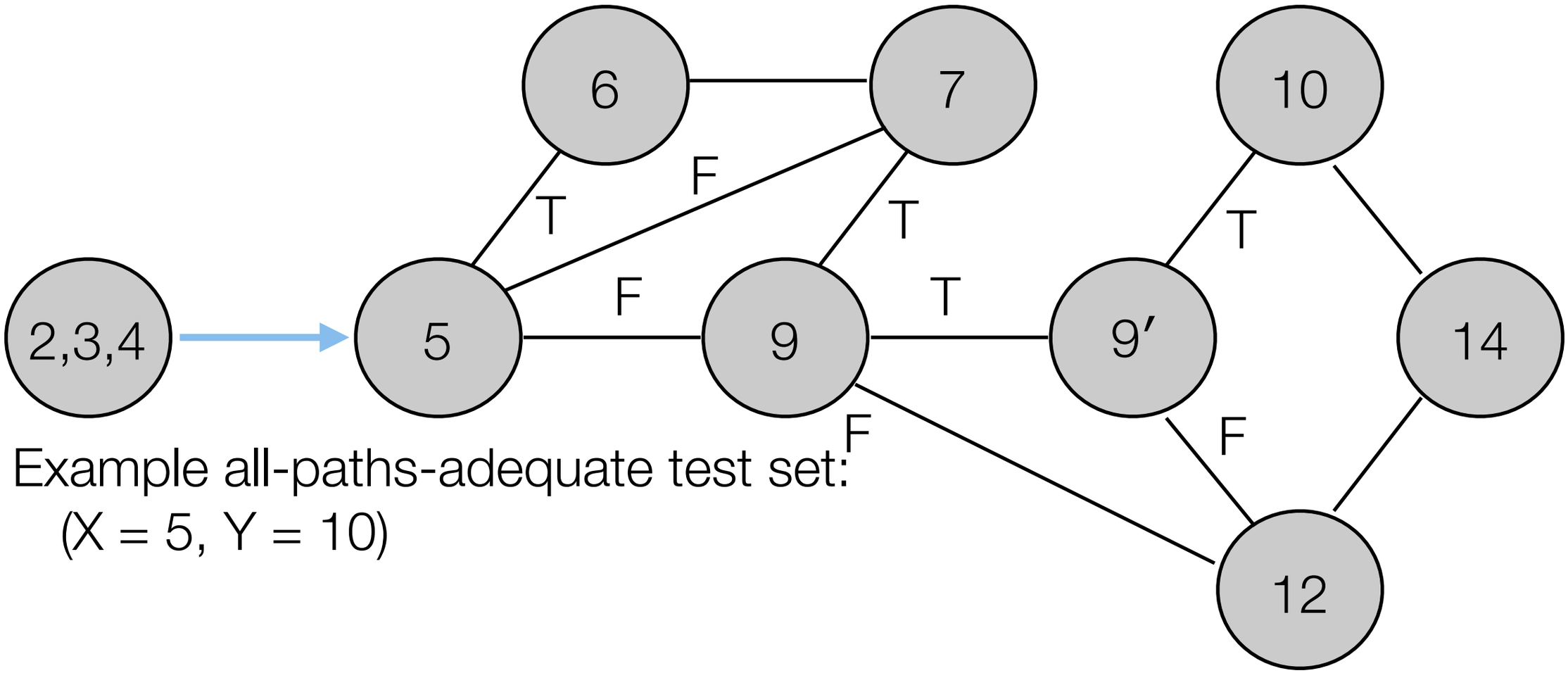
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

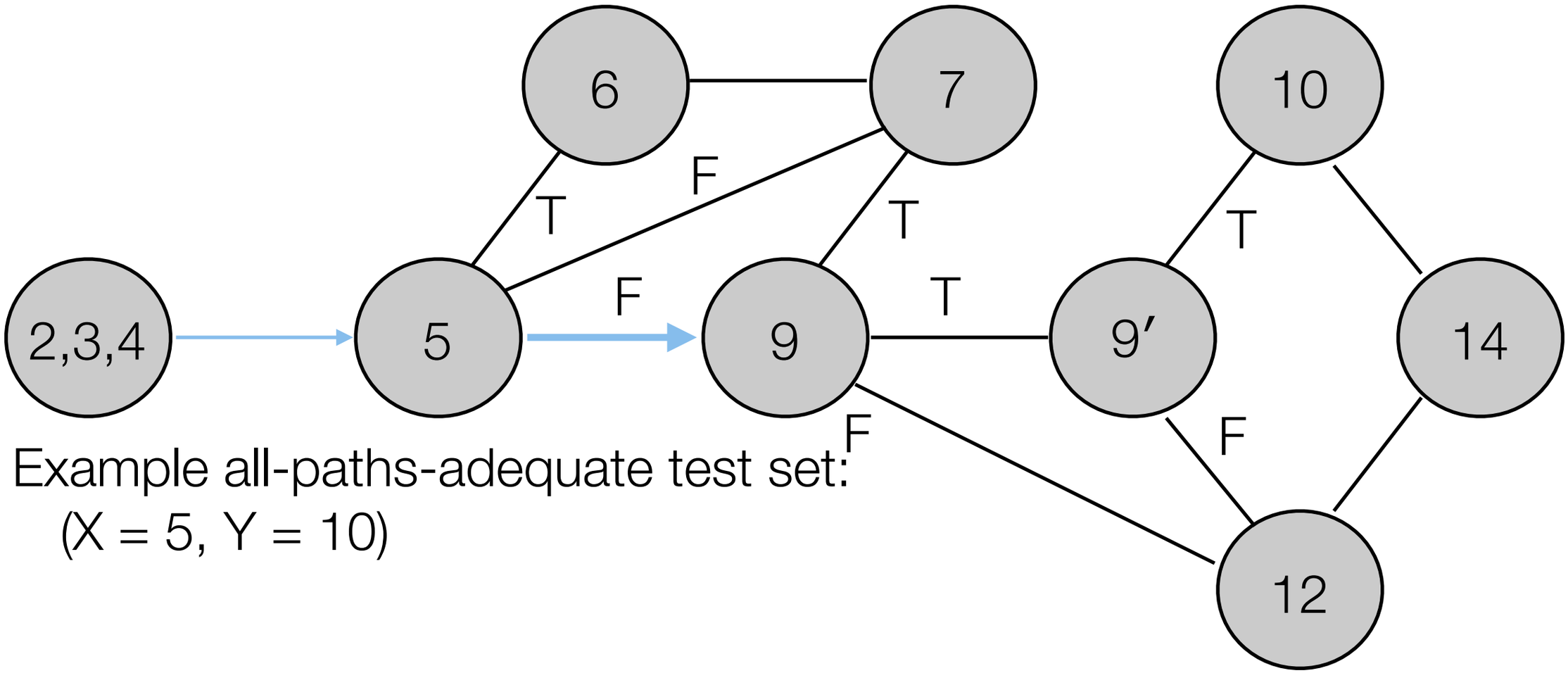
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

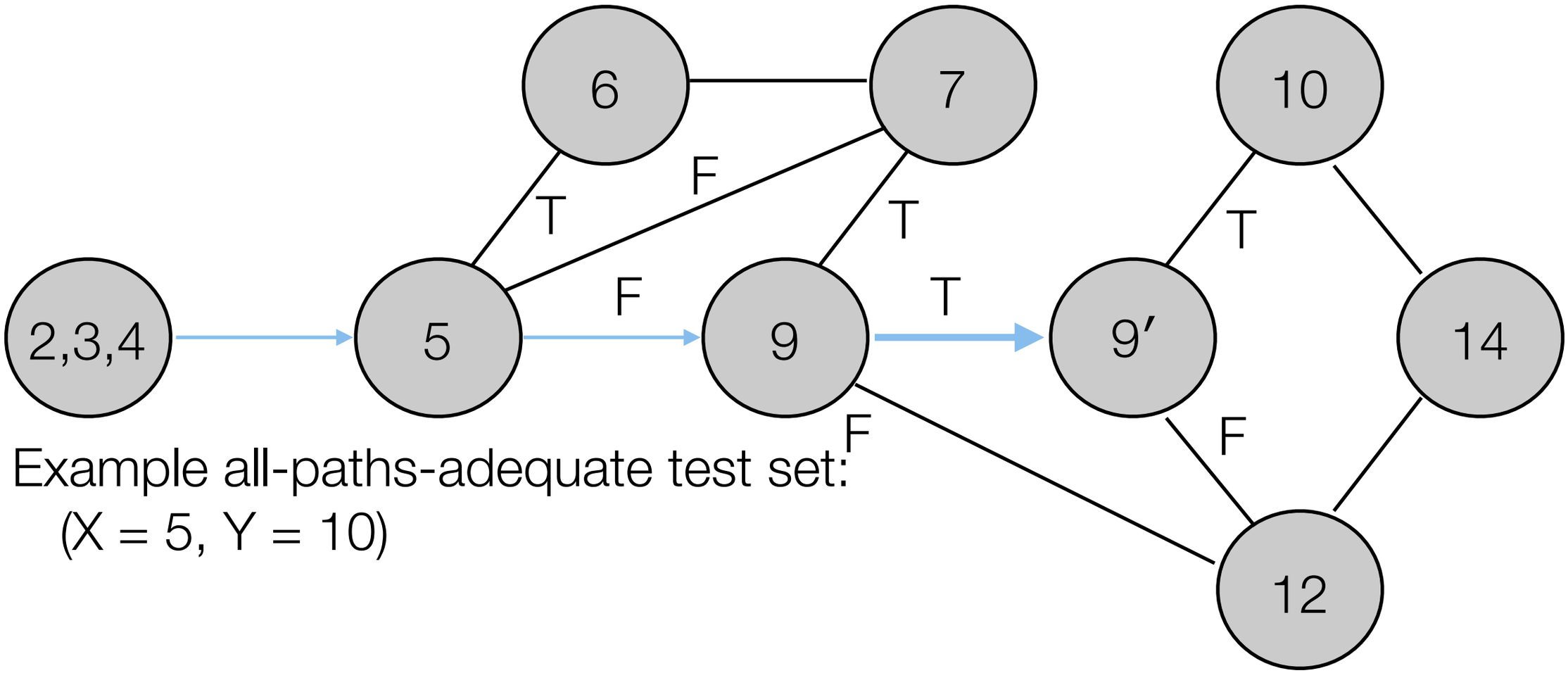
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

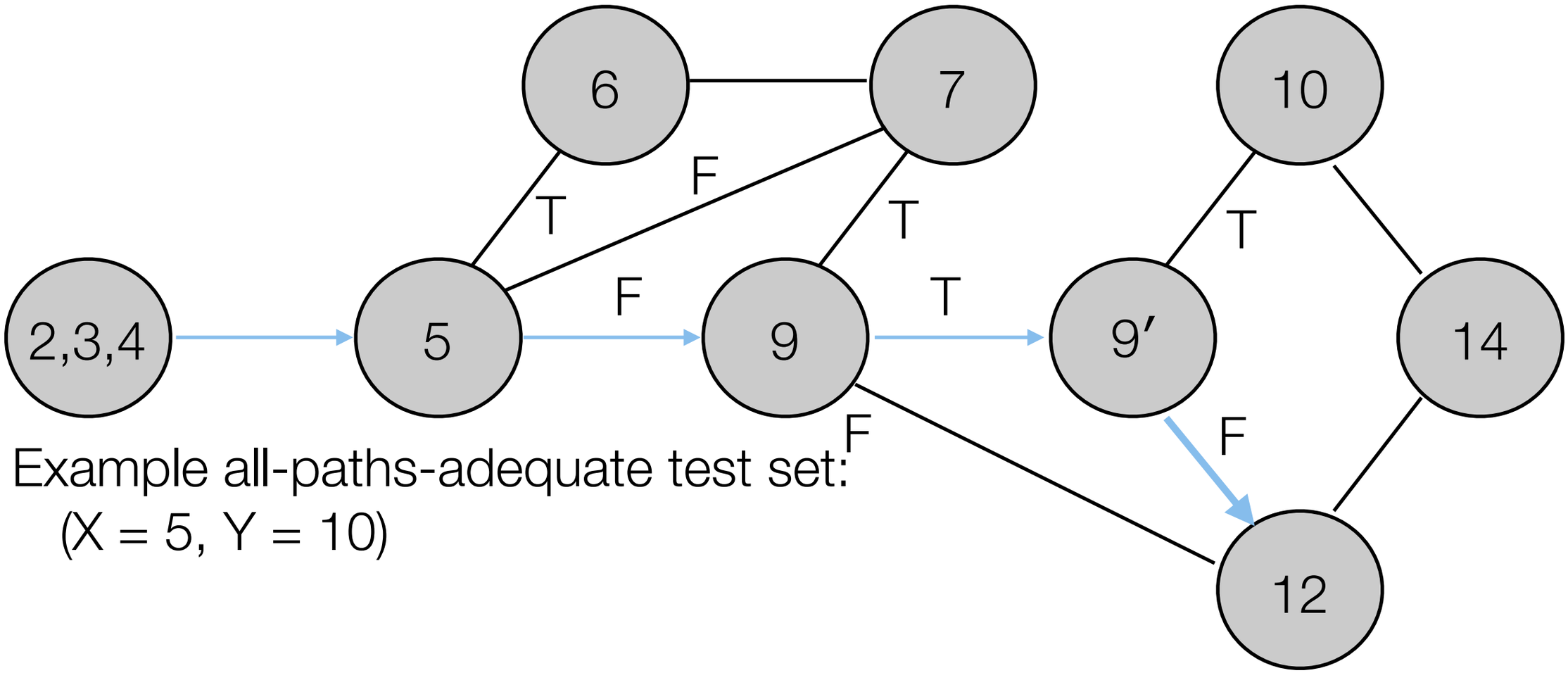
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

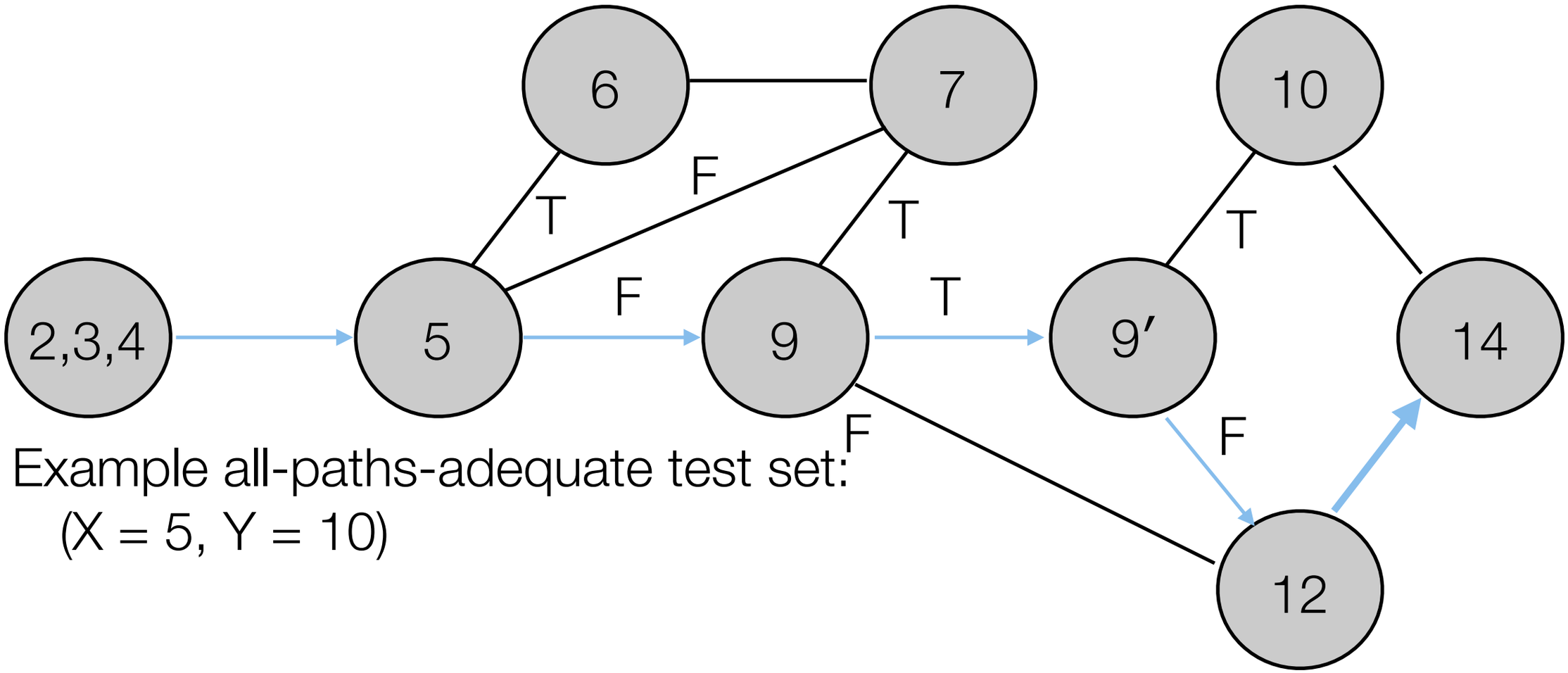
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

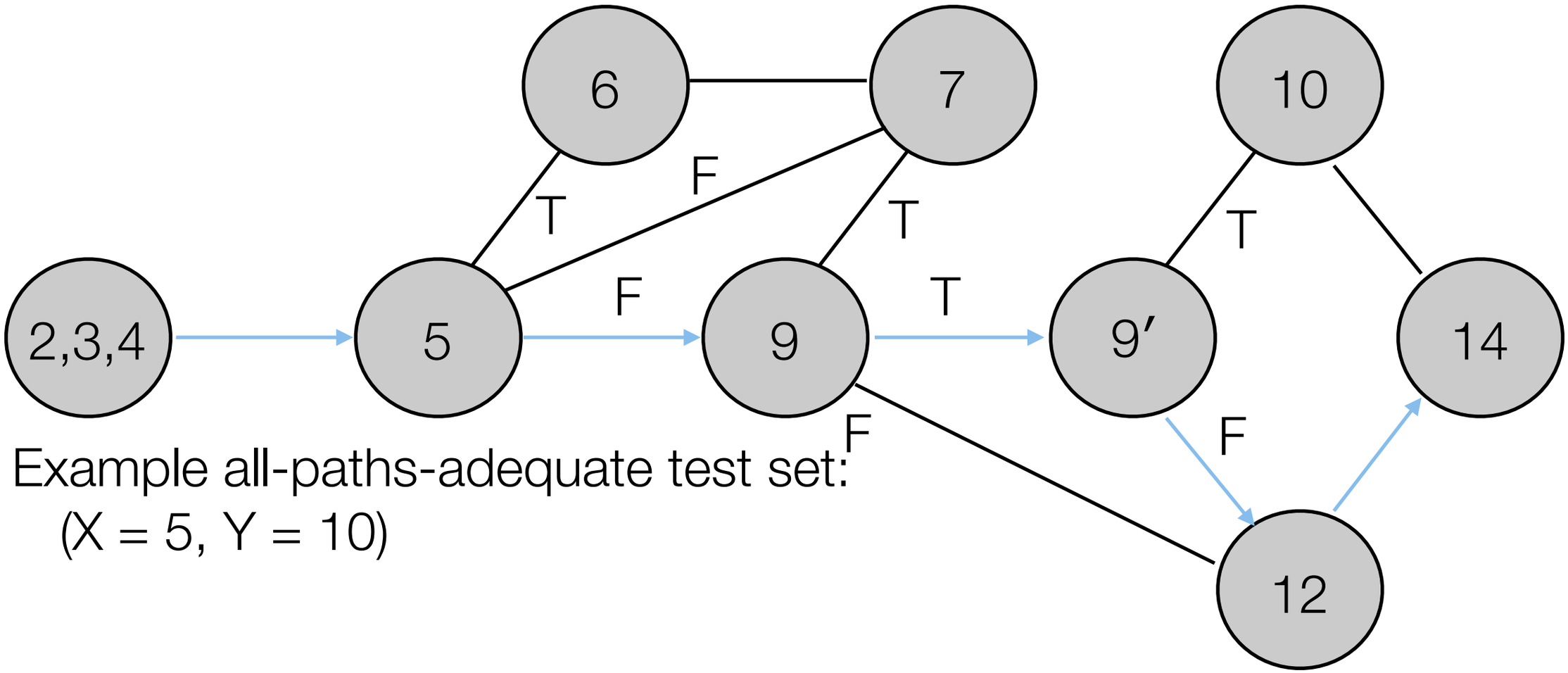
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

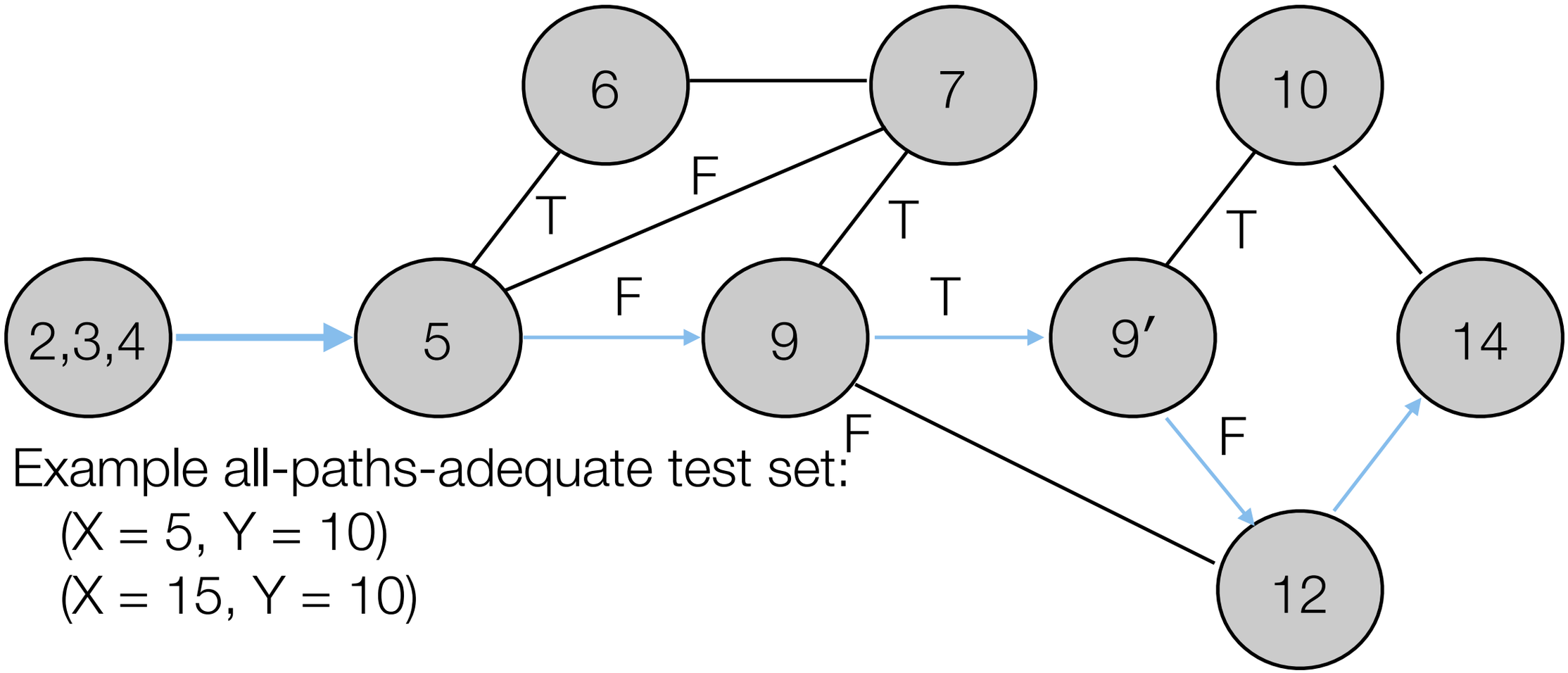
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



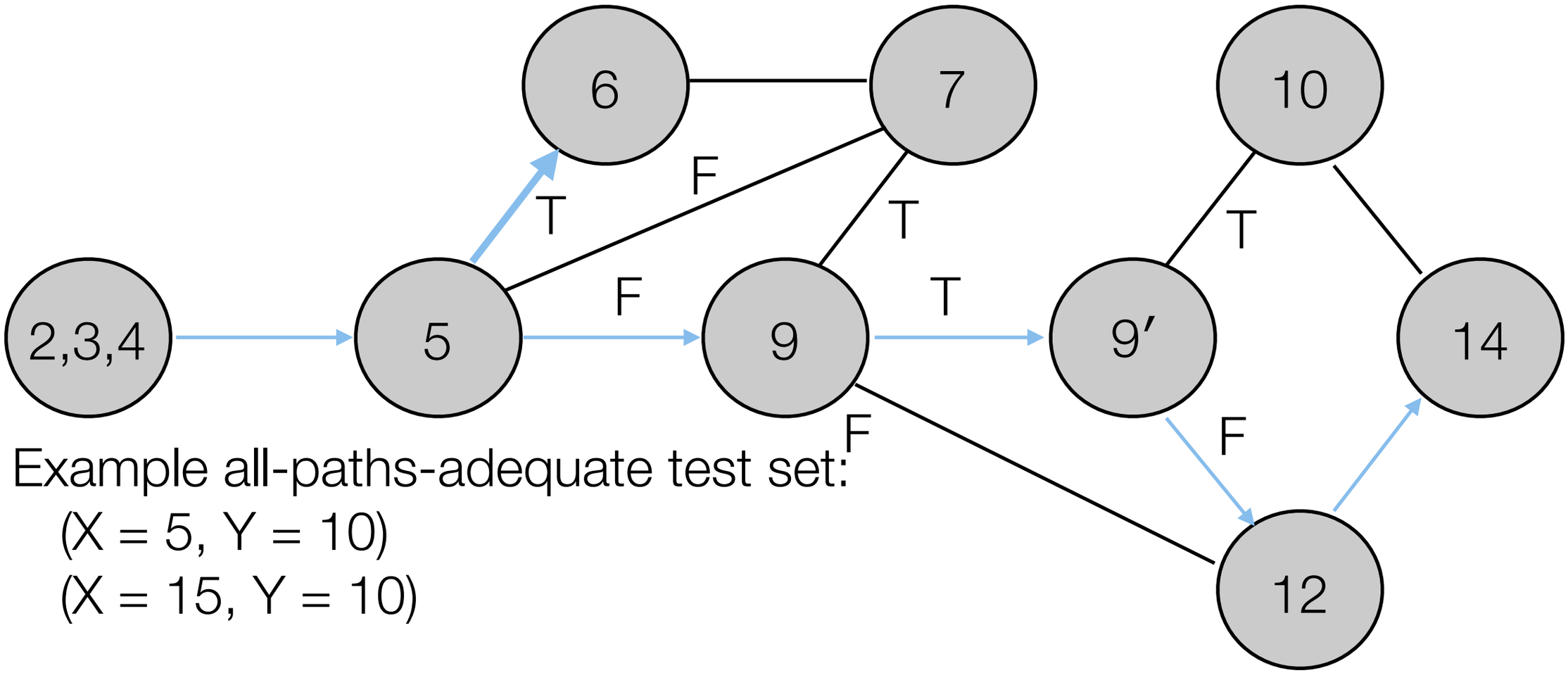
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



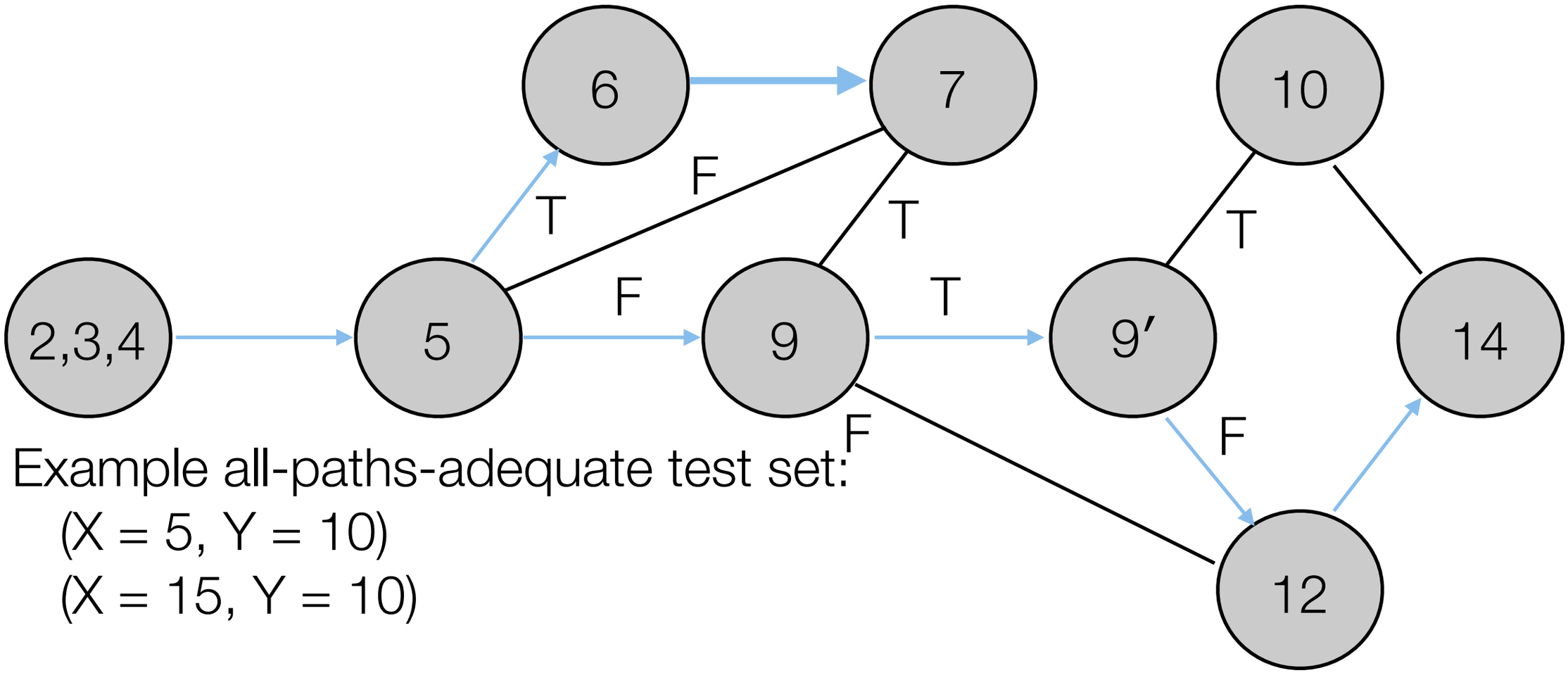
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



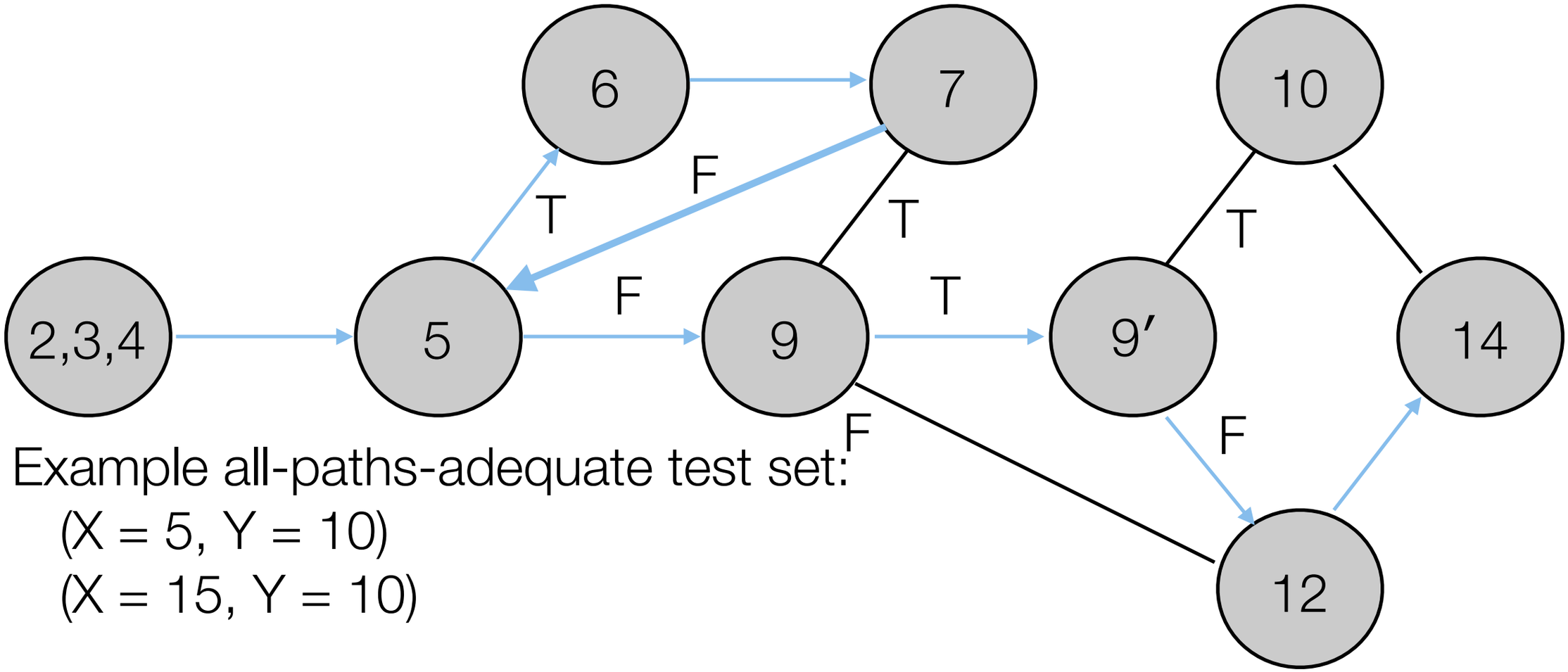
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



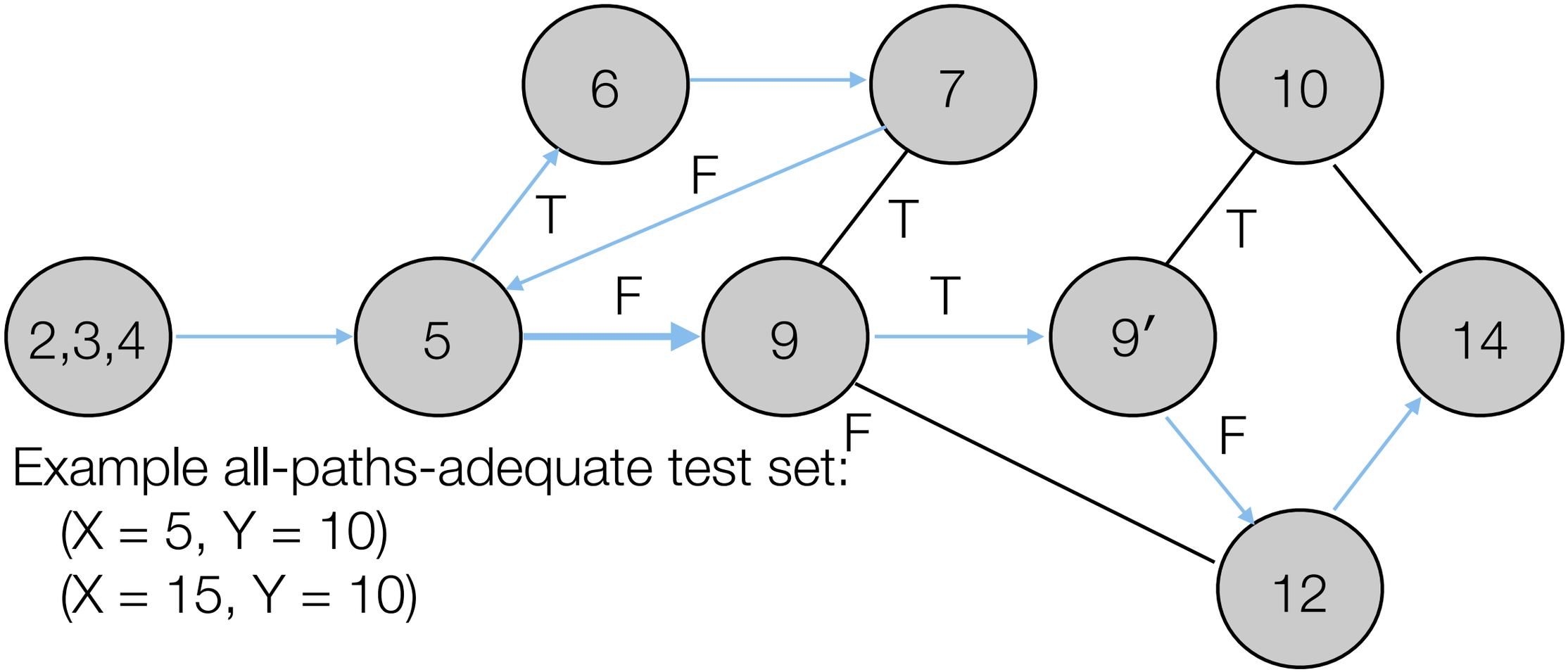
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



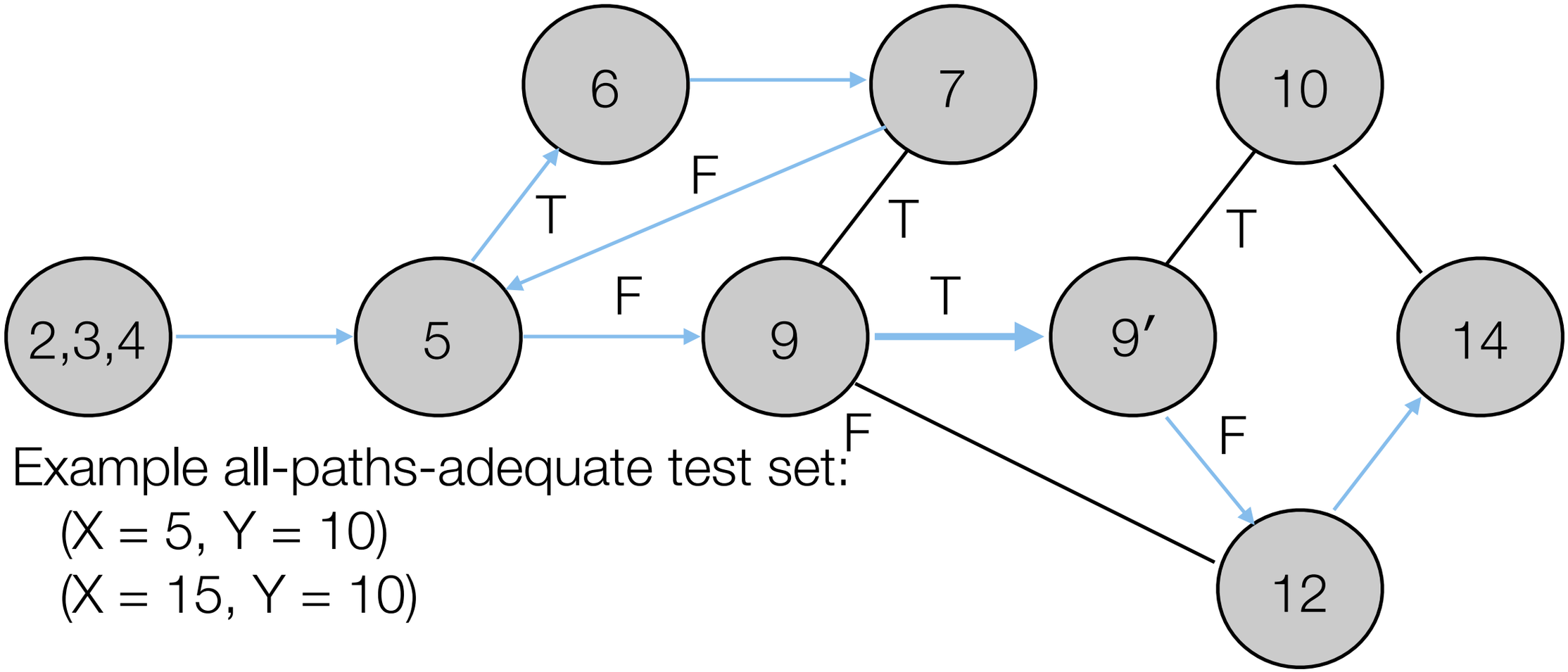
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



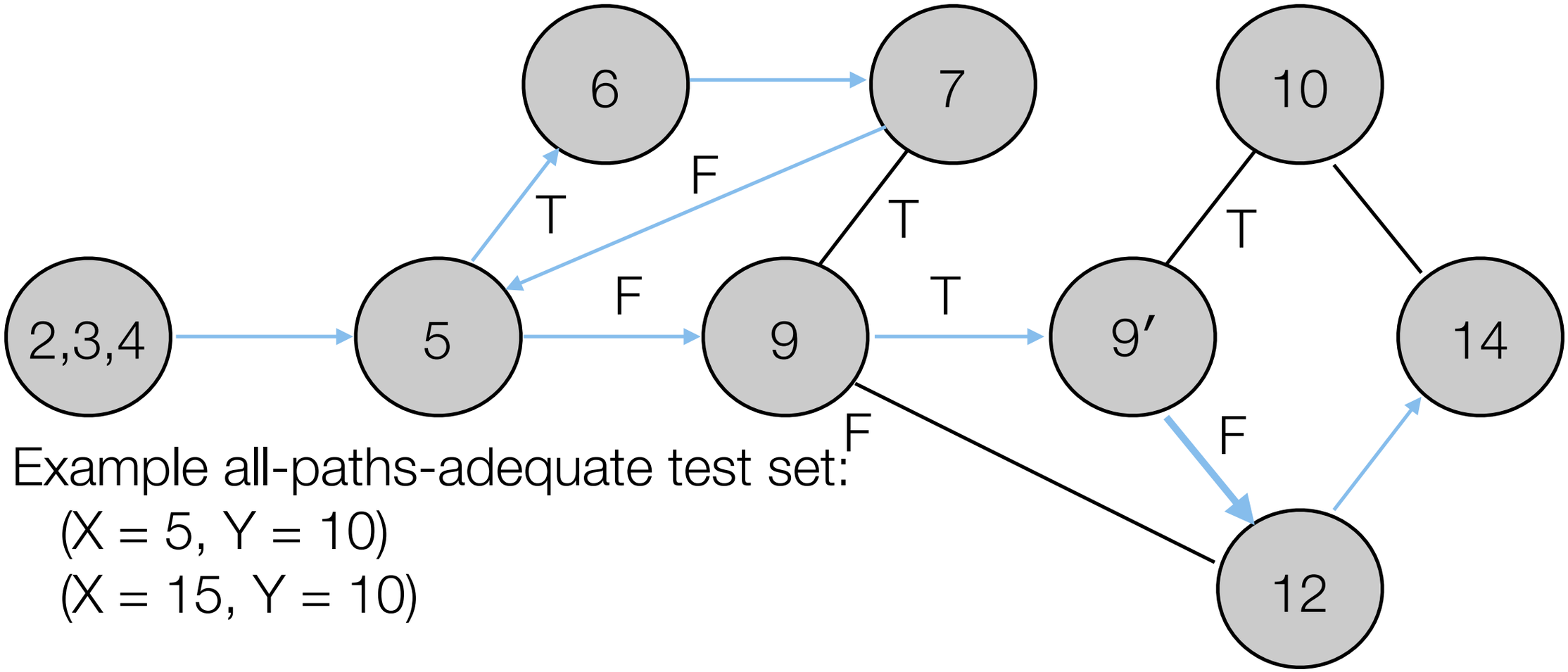
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



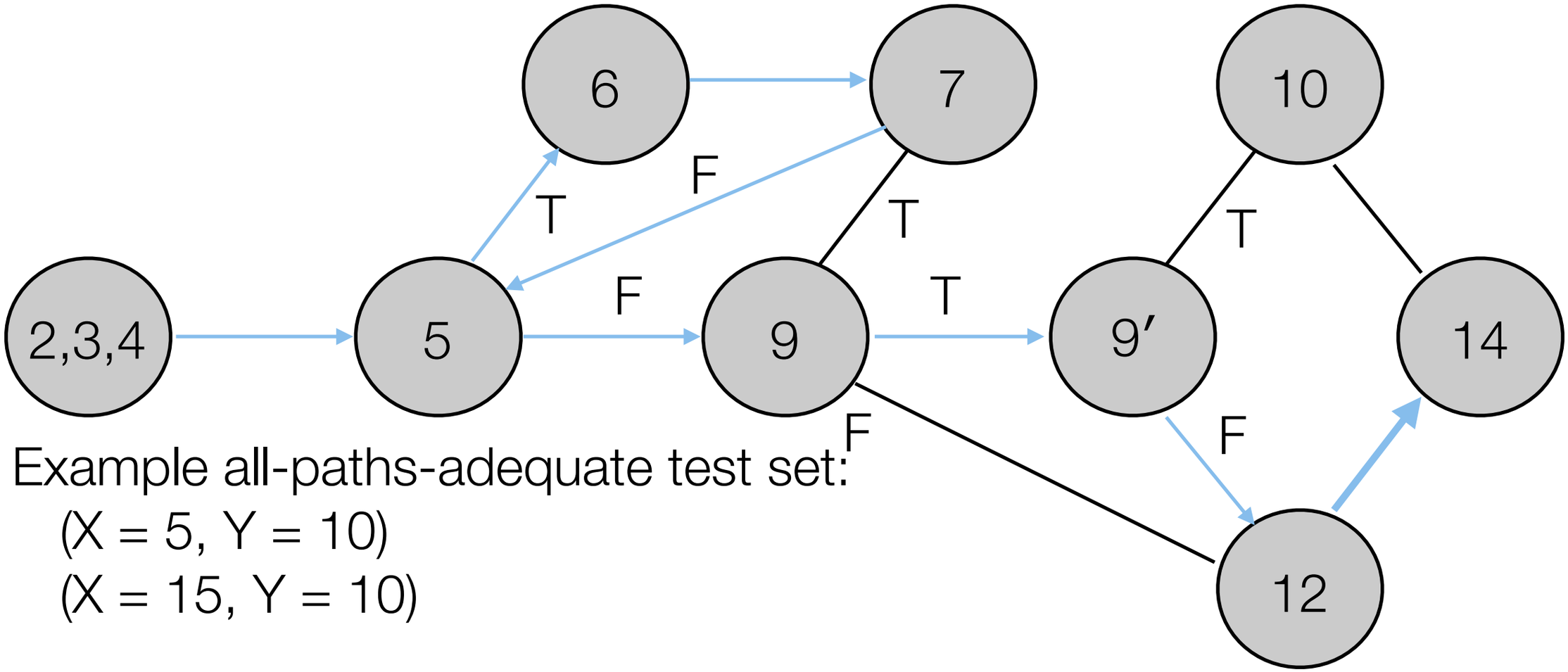
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



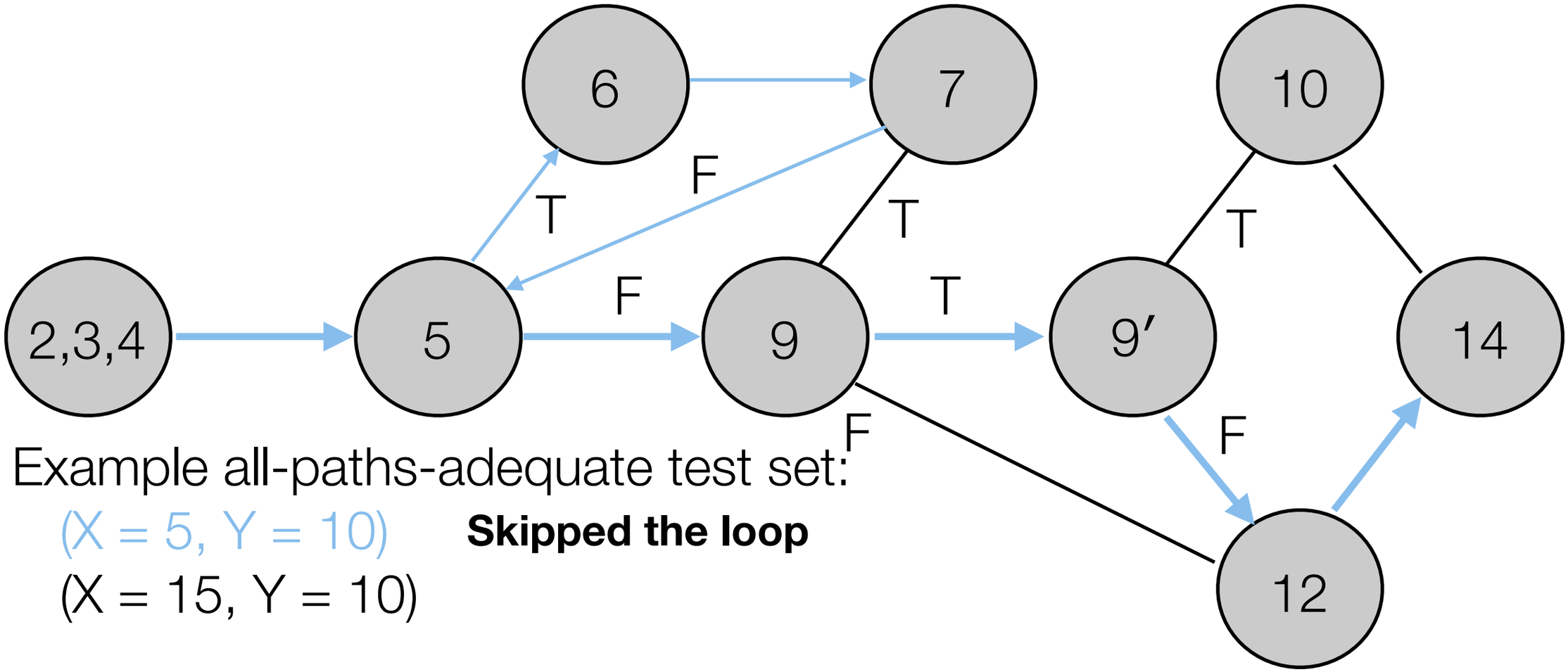
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



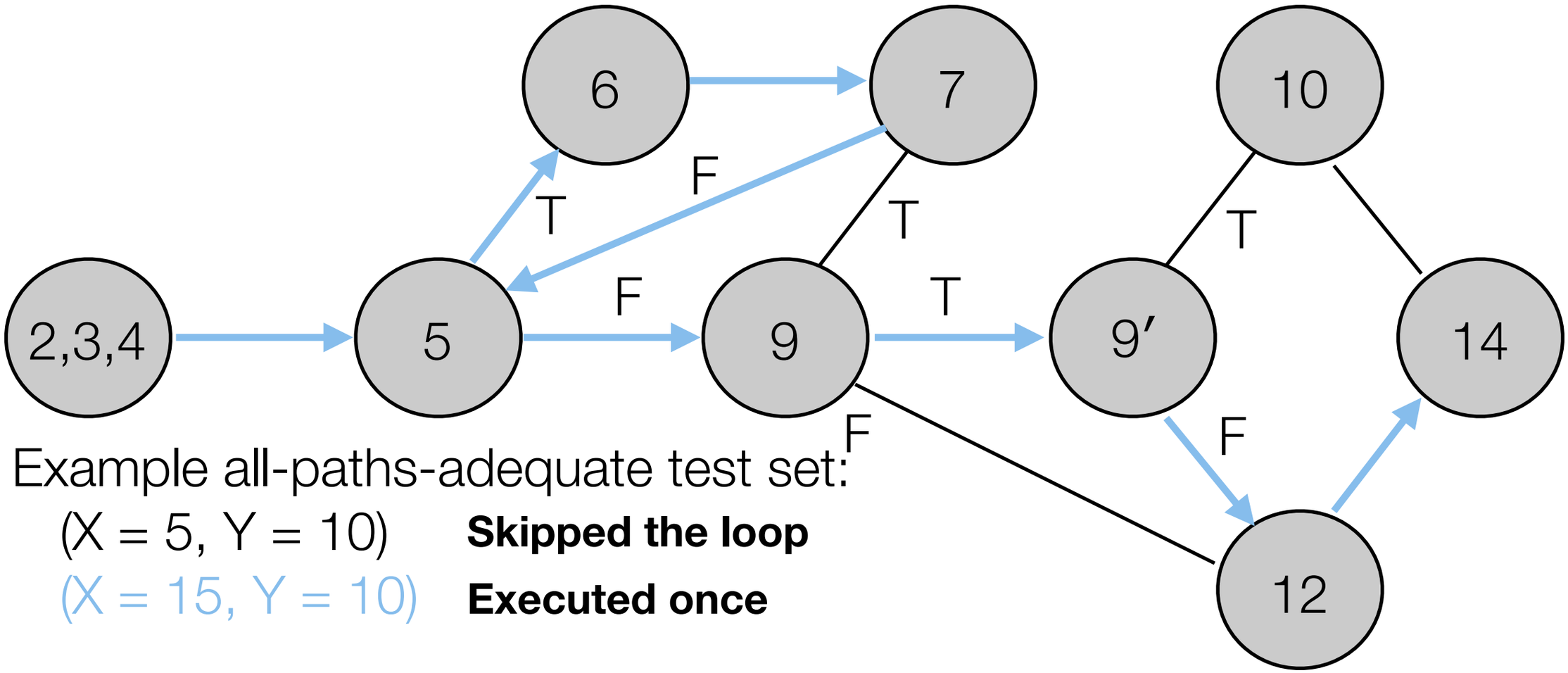
Example all-paths-adequate test set:

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

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



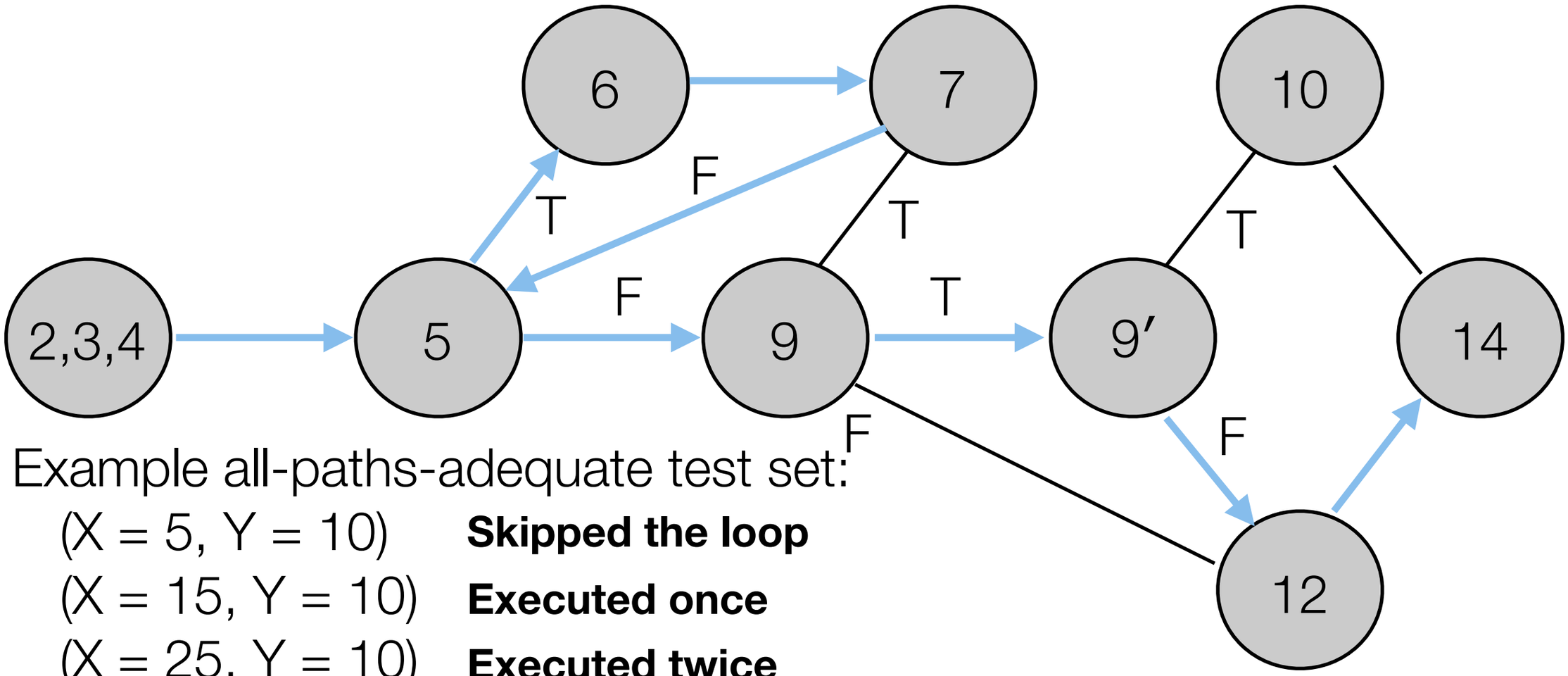
Example all-paths-adequate test set:

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

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

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



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**

And so on… you would also want permutations that exit the loop early

© Kenneth M. Anderson, 2012

Code Coverage Tools

- 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
    - These systems 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

© Kenneth M. Anderson, 2012

Testing Automation (I)

- 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!!)

© Kenneth M. Anderson, 2012

Testing Automation (II)

- 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
- We'll see examples of this later in the semester

© Kenneth M. Anderson, 2012

Continuous Integration

- Since test automation is so critical, systems known as continuous integration frameworks have emerged
- 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

© Kenneth M. Anderson, 2012

Summary

- 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 exists to integrate testing into the version control and build management processes of a development organization

© Kenneth M. Anderson, 2012

Coming Up Next:

- Lecture 6: Agile Methods and Agile Teams
- Lecture 7: Division of Labor and Design Approaches in Concurrency

© Kenneth M. Anderson, 2012

Introduction to Software Testing

CSCI 5828: Foundations of Software Engineering
Lecture 05 — 01/31/2012

© Kenneth M. Anderson, 2012

Goals

- Provide introduction to fundamental concepts of software testing
    - Terminology
    - Testing of Systems
        - unit tests, integration tests, system tests, acceptance tests
    - Testing of Code
        - Black Box
        - Gray Box
        - White Box
        - Code Coverage

© Kenneth M. Anderson, 2012

Testing

- Testing is a **critical element** of software development life cycles
  - called **software quality control** or **software quality assurance**
  - basic goals: **validation** and **verification**
    - validation: **are we building the right product?**
      - verification: **does "X" meet its specification?**
        - where "X" 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

© Kenneth M. Anderson, 2012

Terminology

- An **error** is a mistake made by an engineer
  - often a misunderstanding of a requirement or design specification
- A **fault** is a manifestation of that error in the code
  - what we often call “a bug”
- A **failure** is an incorrect output/behavior that is caused by executing a fault
  - The failure may occur immediately (crash!) or much, much later in the execution
- **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?

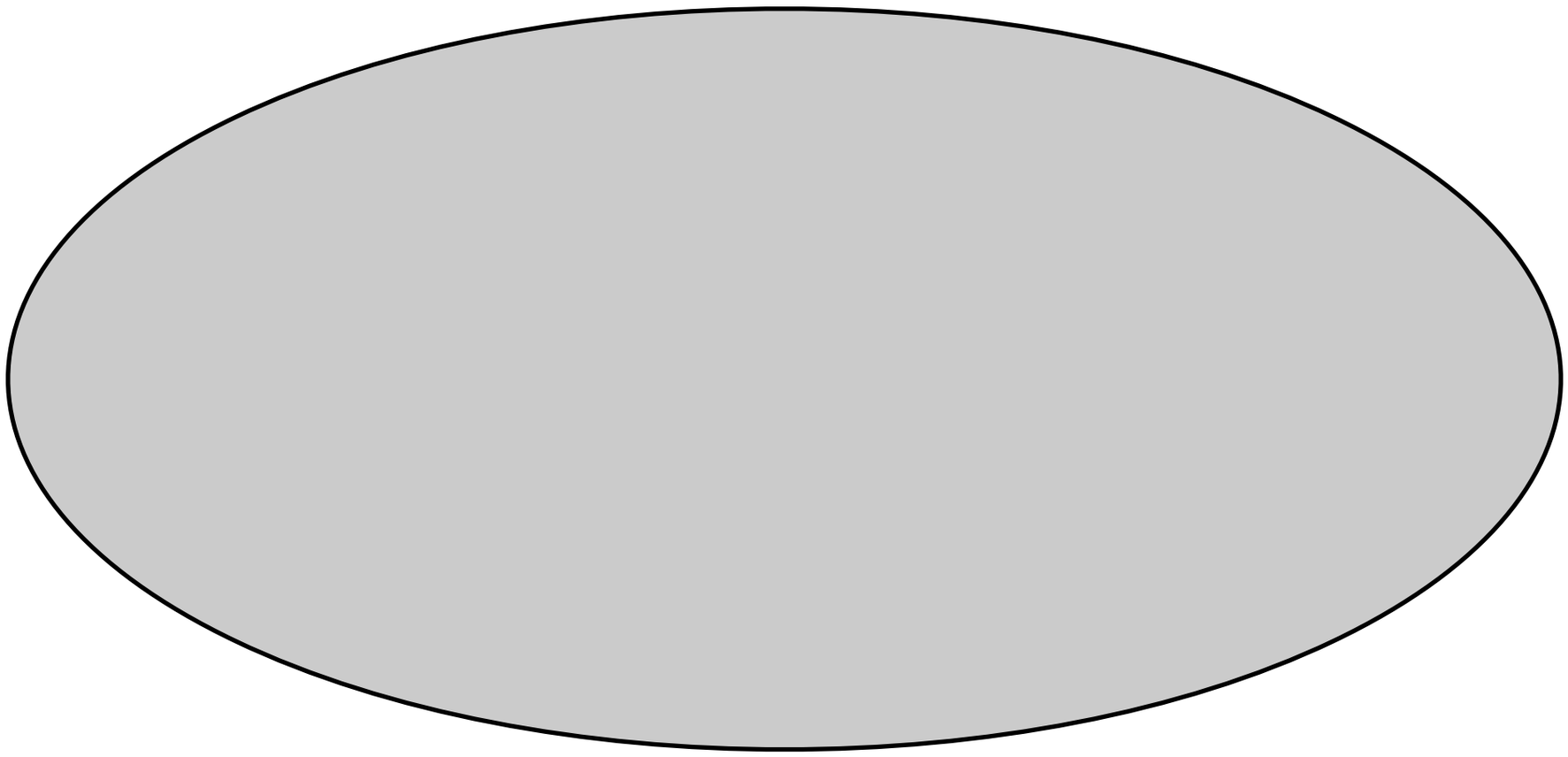
© Kenneth M. Anderson, 2012

No!

- 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 resulting in no visible failures!
      - this is particularly insidious
- However, if we do a good job in creating a test set that
  - covers all functional capabilities of a system
  - and 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

© Kenneth M. Anderson, 2012

Looking for Faults

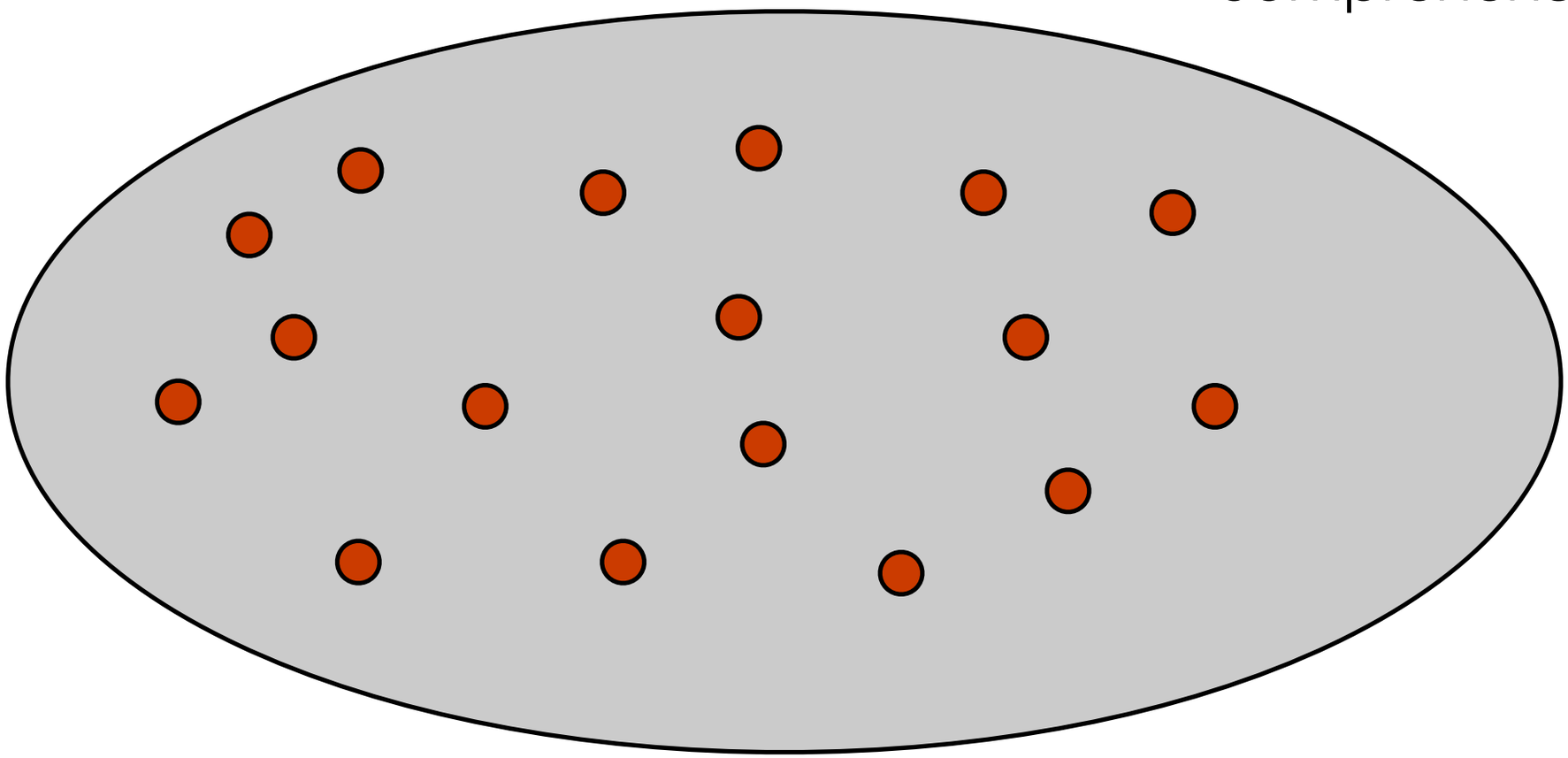


All possible states/behaviors of a system

© Kenneth M. Anderson, 2012

Looking for Faults

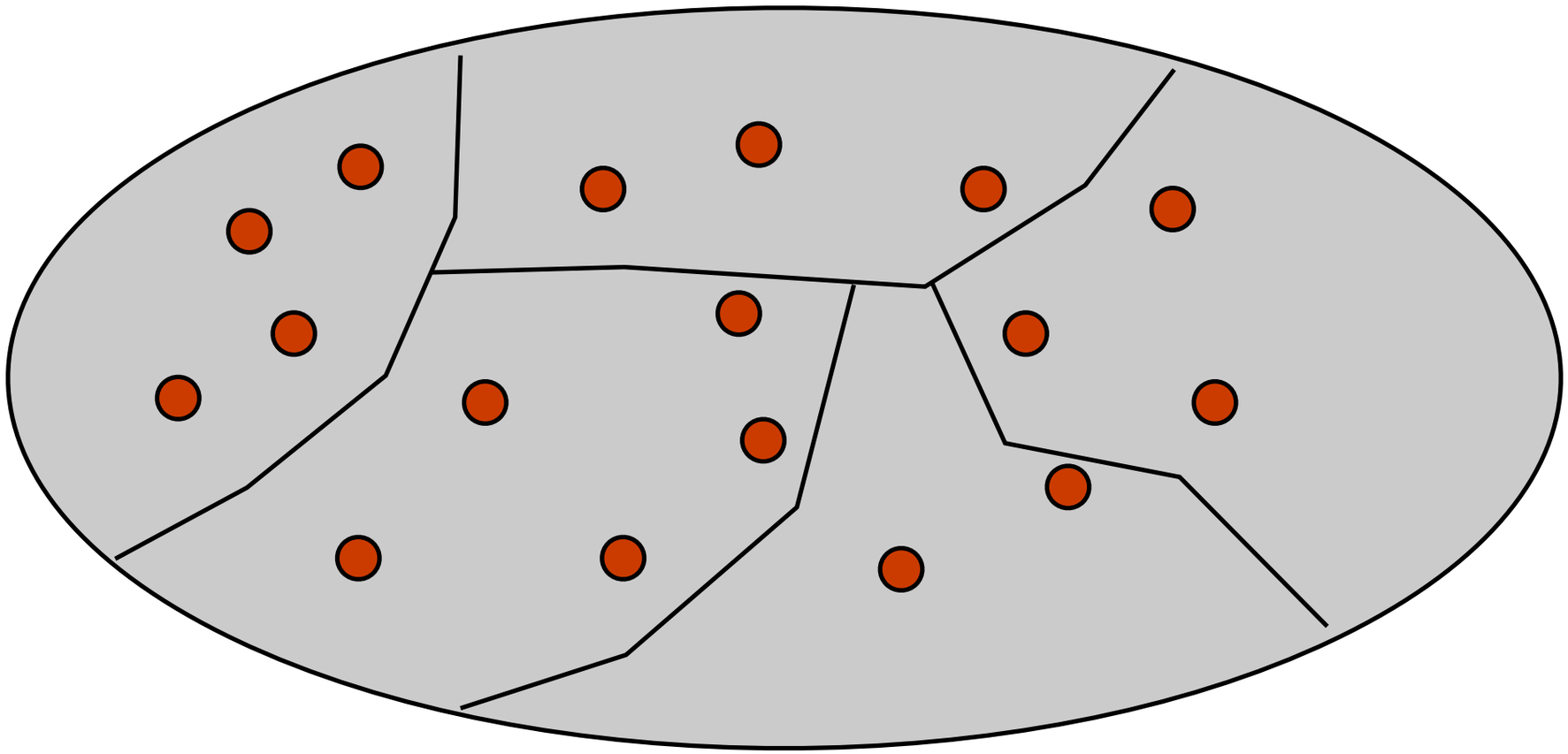
As you can see, its not very comprehensive



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

© Kenneth M. Anderson, 2012

One way forward? Fold



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

© Kenneth M. Anderson, 2012

What does that mean?

- 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

© Kenneth M. Anderson, 2012

Completeness

- 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?

© Kenneth M. Anderson, 2012

Continuous Testing

- Testing is a continuous process that should be performed at every stage of a software development process
    - During requirements gathering, for instance, we must continually query the user, “Did we get this right?”
    - Facilitated by an emphasis on iteration throughout a life cycle
- at the end of each iteration
    - we check our results to see if what we built is meeting our requirements (specification)

© Kenneth M. Anderson, 2012

Testing the System (I)

- **Unit Tests**
  - Tests that cover low-level aspects of a system
    - For each module, does each operation perform as expected
    - For method **foo()**, we'd like to see another method **testFoo()**
- **Integration Tests**
  - Tests that check that modules work together in combination
  - Most projects on schedule until they hit this point (MMM, Brooks)
    - 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 (Mars Polar Lander)

© Kenneth M. Anderson, 2012

Testing the System (II)

- **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

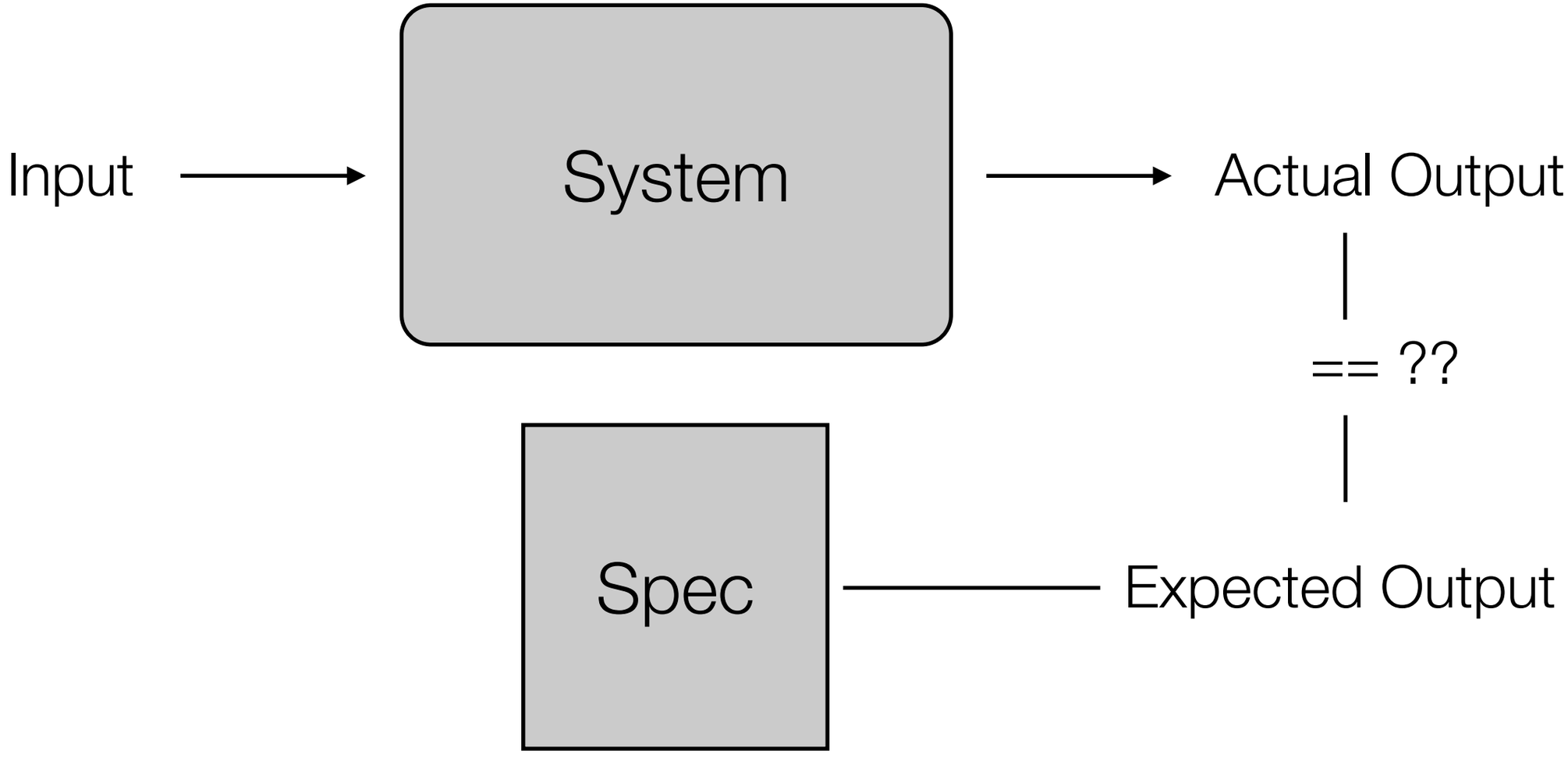
© Kenneth M. Anderson, 2012

Multi-Level Testing

- 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, …

© Kenneth M. Anderson, 2012

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!

© Kenneth M. Anderson, 2012

Results

- 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

© Kenneth M. Anderson, 2012

Black Box Categories

- 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

© Kenneth M. Anderson, 2012

Grey Box Testing

- 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

© Kenneth M. Anderson, 2012

White Box Testing

- 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?

© Kenneth M. Anderson, 2012

Code Coverage (I)

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

© Kenneth M. Anderson, 2012

Code Coverage (II)

- 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)

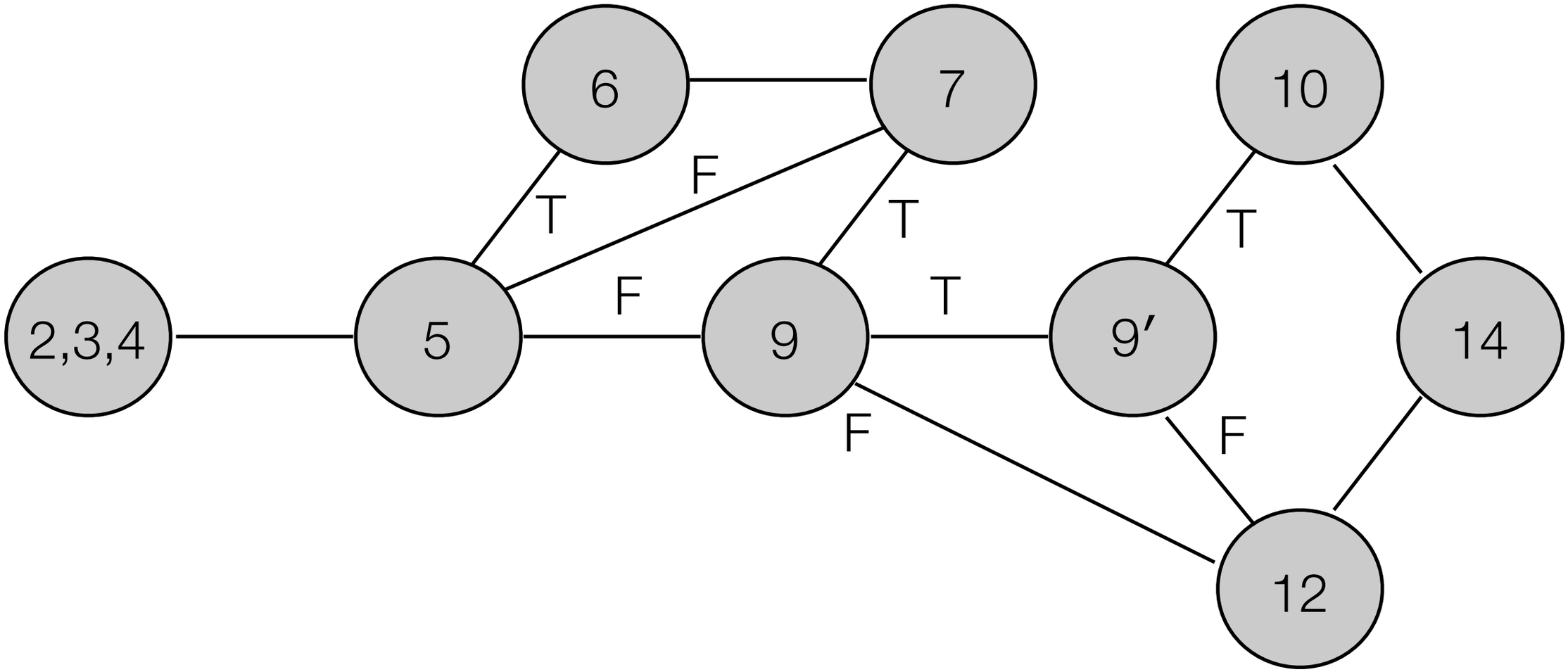
© Kenneth M. Anderson, 2012

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;
```

© Kenneth M. Anderson, 2012

P’s Control Flow Graph (CFG)



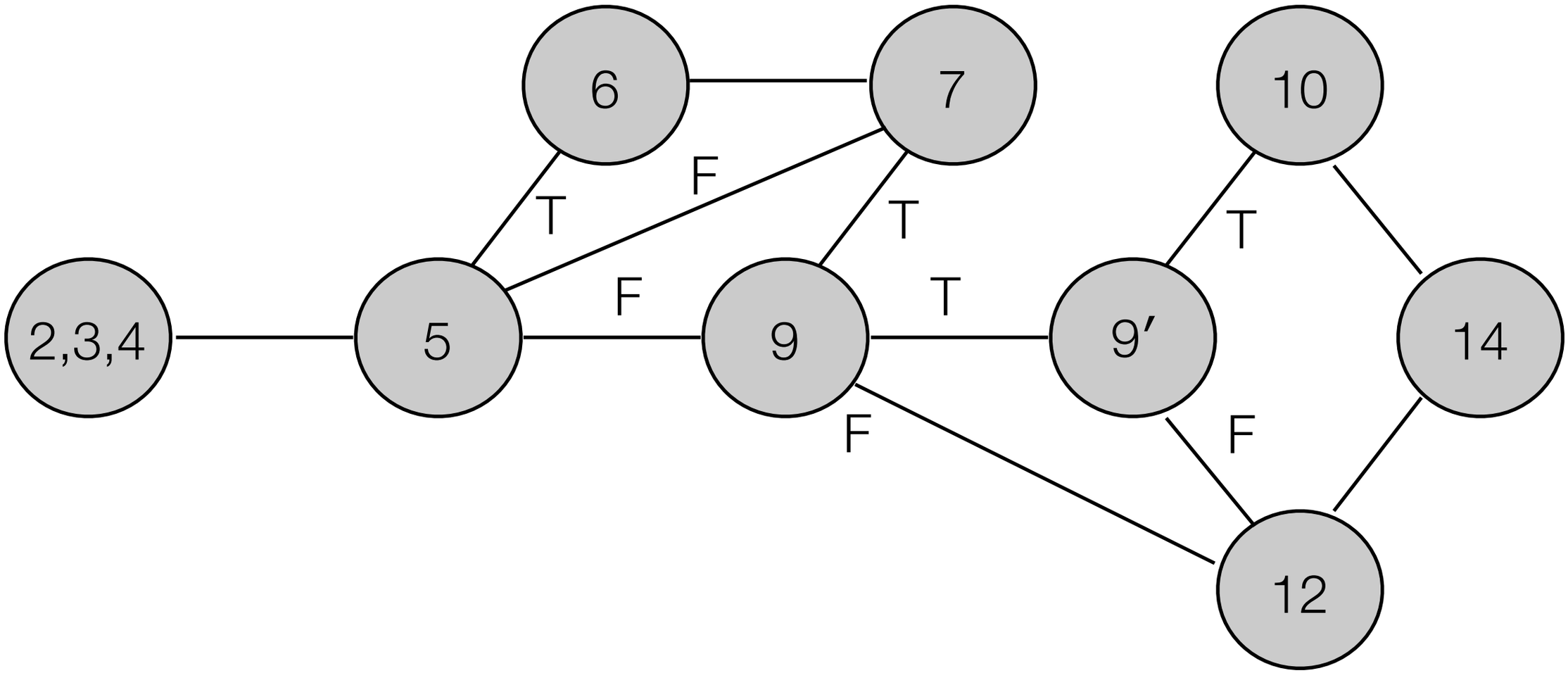
© Kenneth M. Anderson, 2012

White-box Testing Criteria

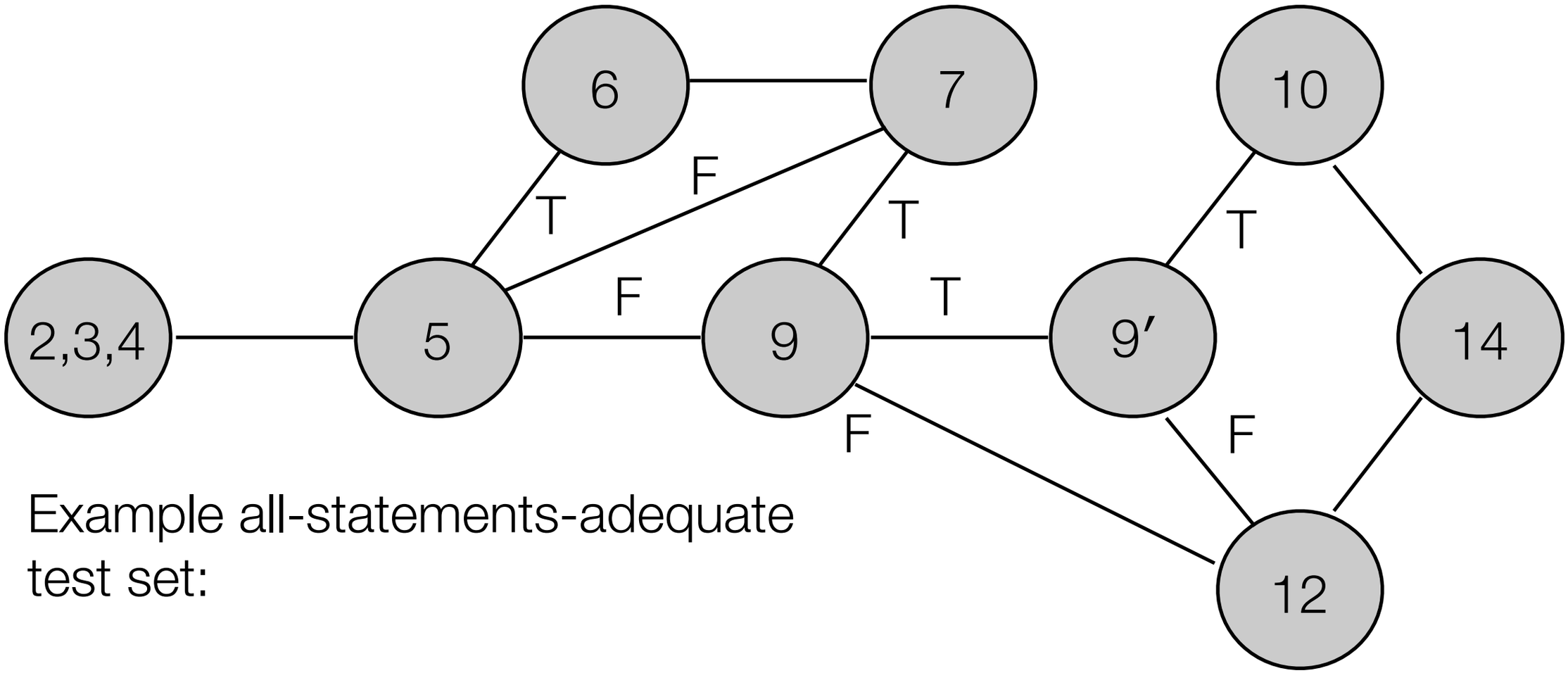
- Statement Coverage
  - Create a test set T such that
    - by executing P for each t in T
      - each elementary statement of P is executed at least once

© Kenneth M. Anderson, 2012

All-Statements Coverage of P

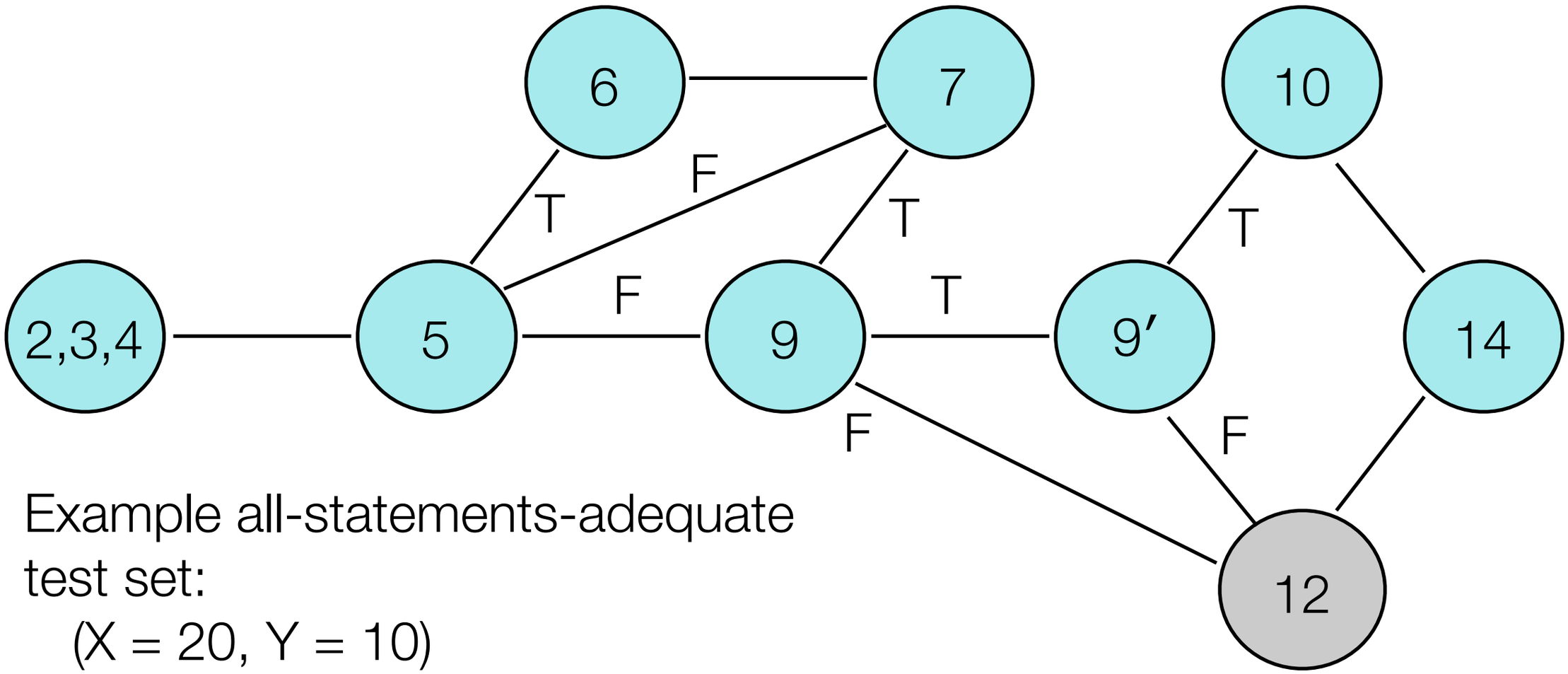


All-Statements Coverage of P



Example all-statements-adequate test set:

All-Statements Coverage of P

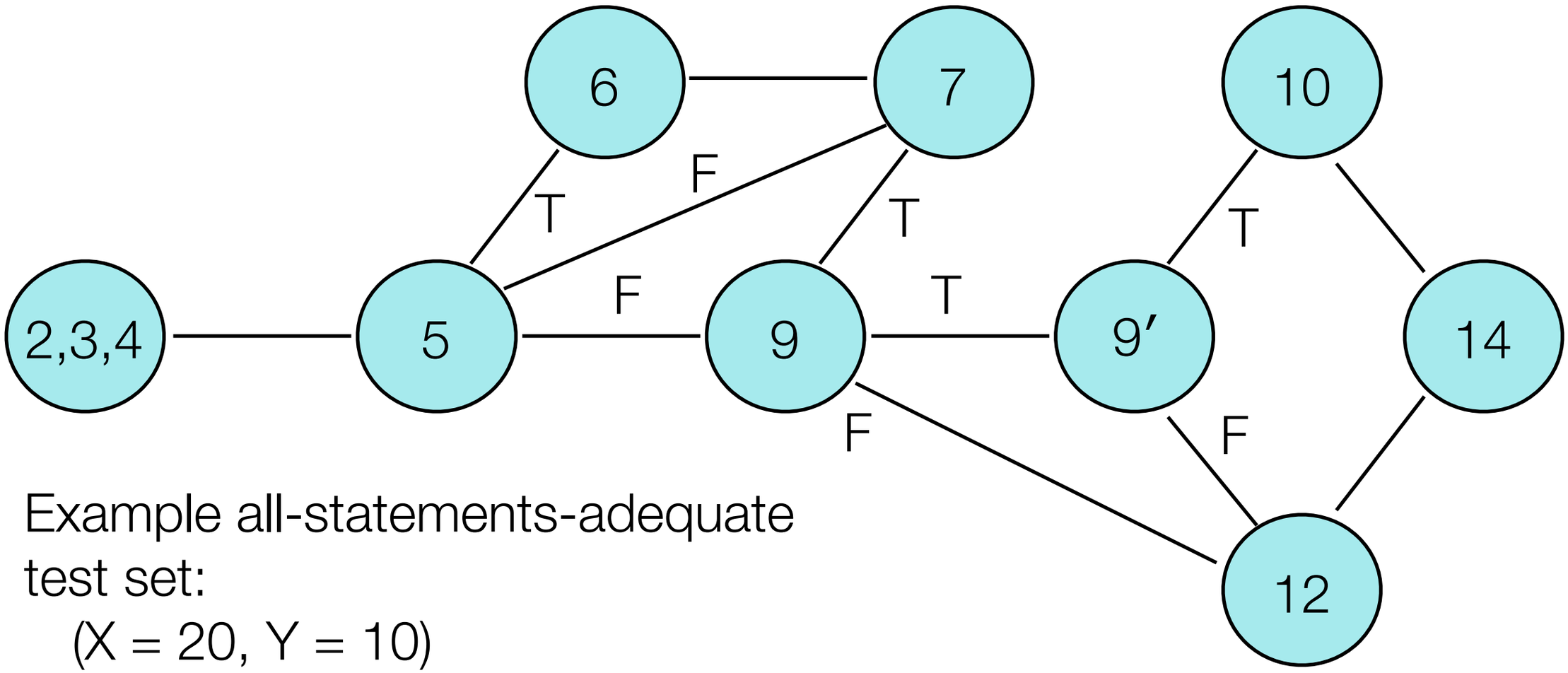


Example all-statements-adequate test set:

(X = 20, Y = 10)

© Kenneth M. Anderson, 2012

All-Statements Coverage of P



Example all-statements-adequate test set:

(X = 20, Y = 10)

(X = 20, Y = 30)

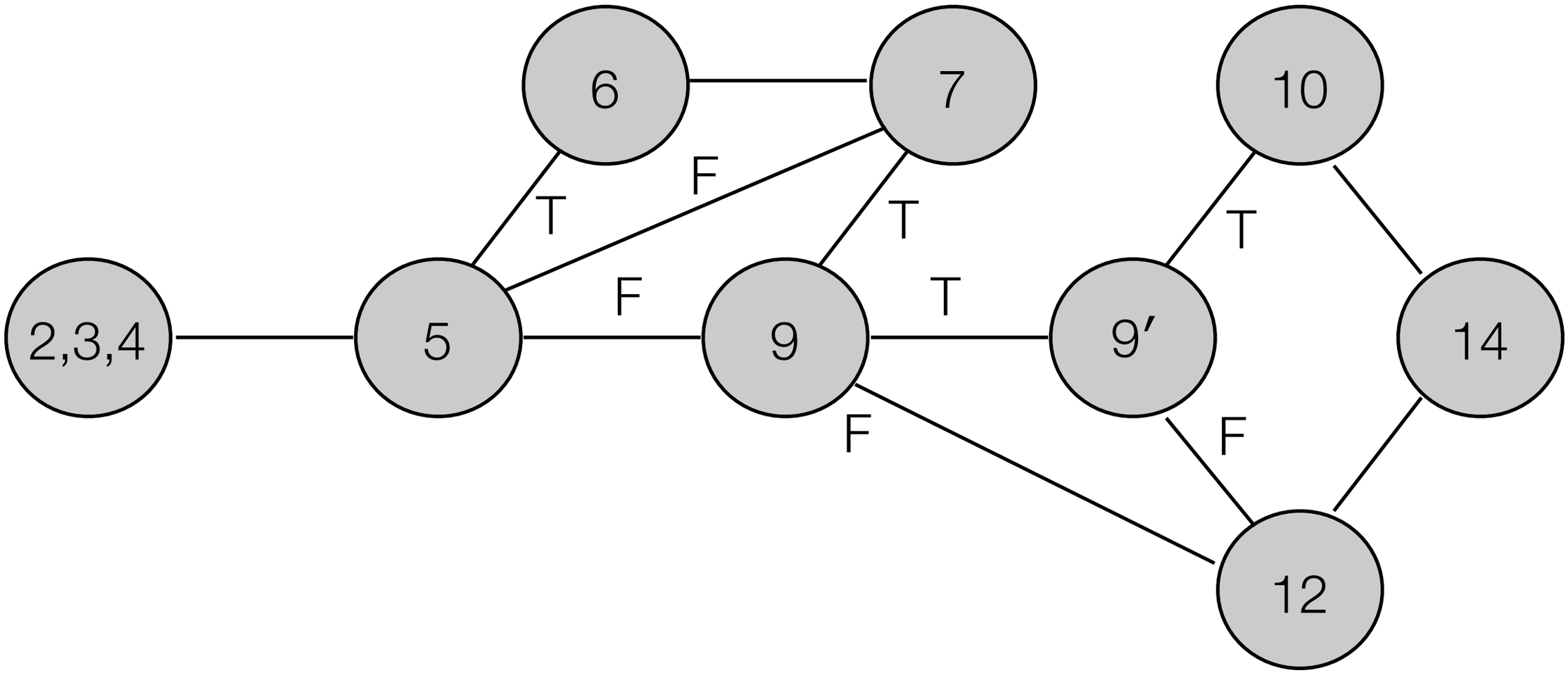
© Kenneth M. Anderson, 2012

White-box Testing Criteria

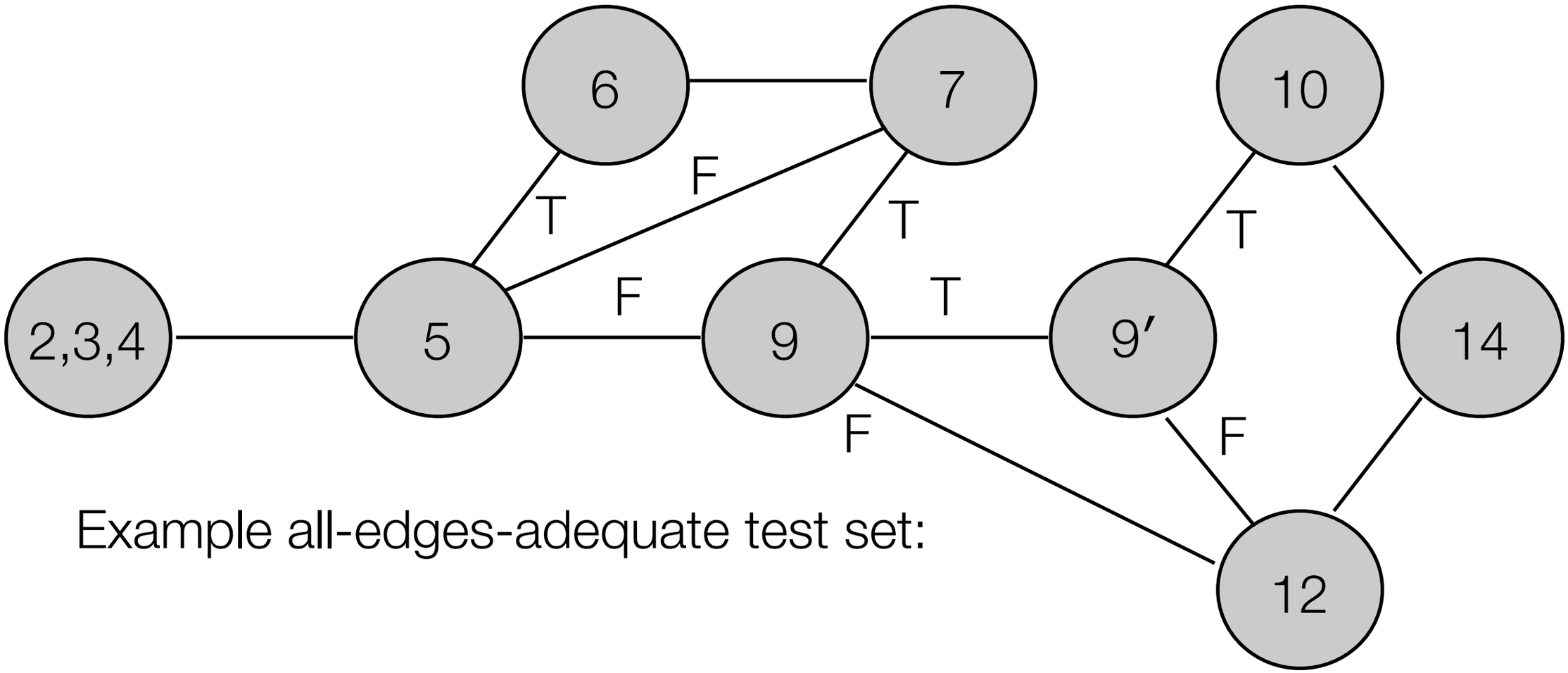
- Edge Coverage
  - Select a test set T such that
    - by executing P for each t in T
      - each edge of P's control flow graph is traversed at least once

© Kenneth M. Anderson, 2012

All-Edges Coverage of P



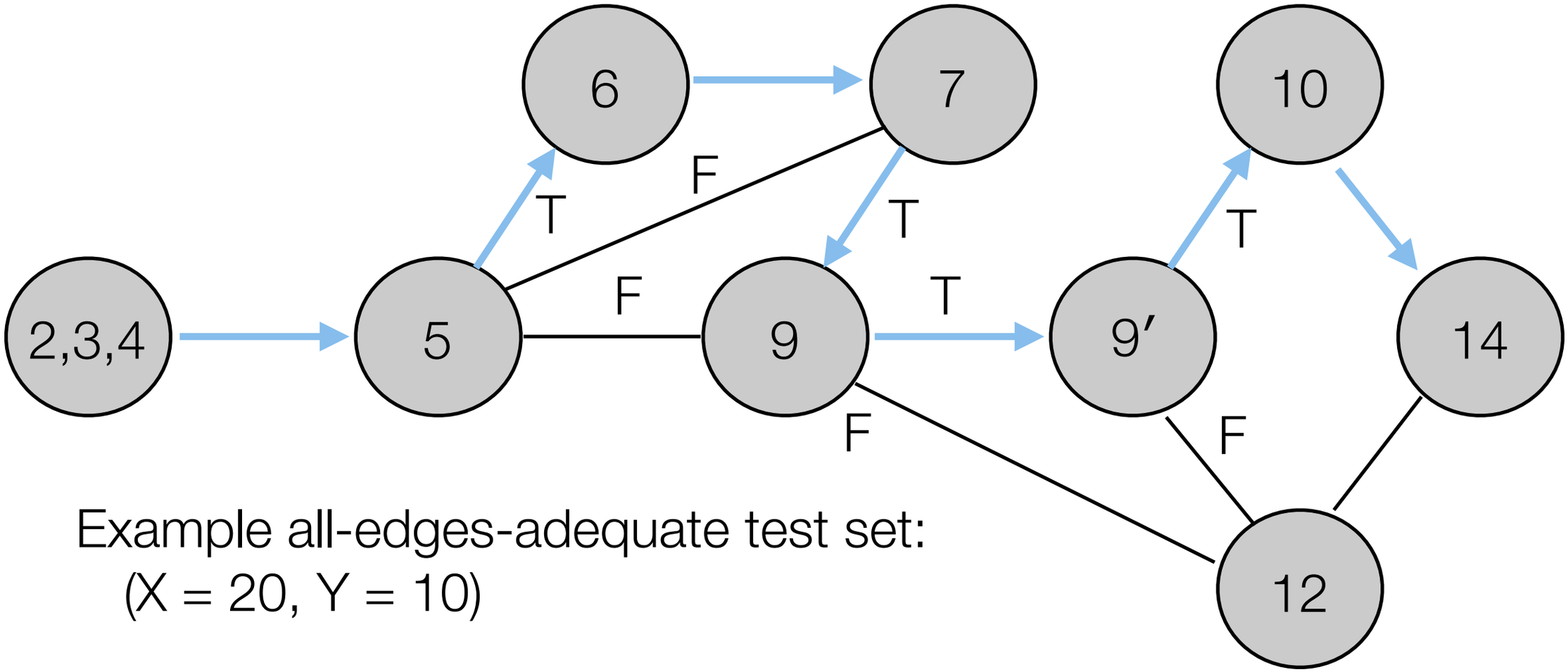
All-Edges Coverage of P



Example all-edges-adequate test set:

© Kenneth M. Anderson, 2012

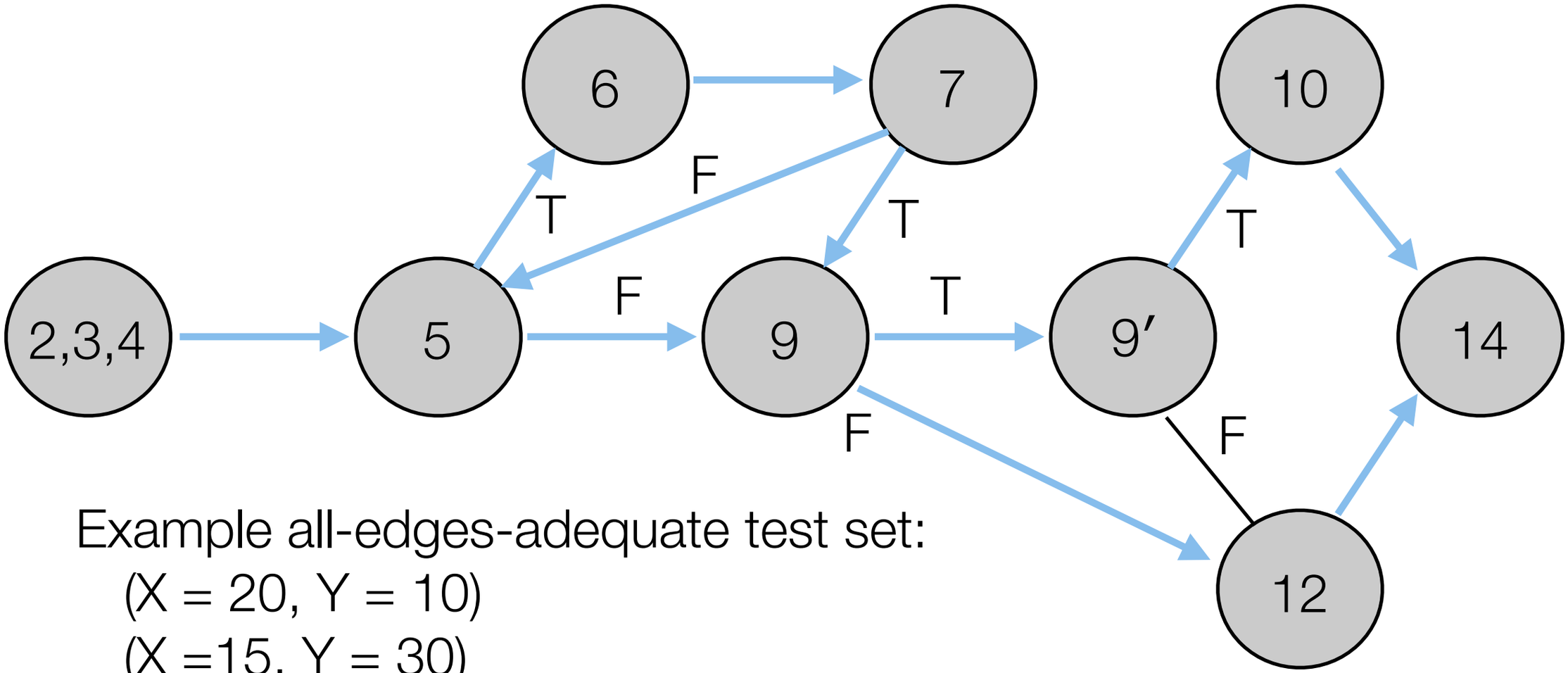
All-Edges Coverage of P



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

© Kenneth M. Anderson, 2012

All-Edges Coverage of P



Example all-edges-adequate test set:

(X = 20, Y = 10)

(X =15, Y = 30)

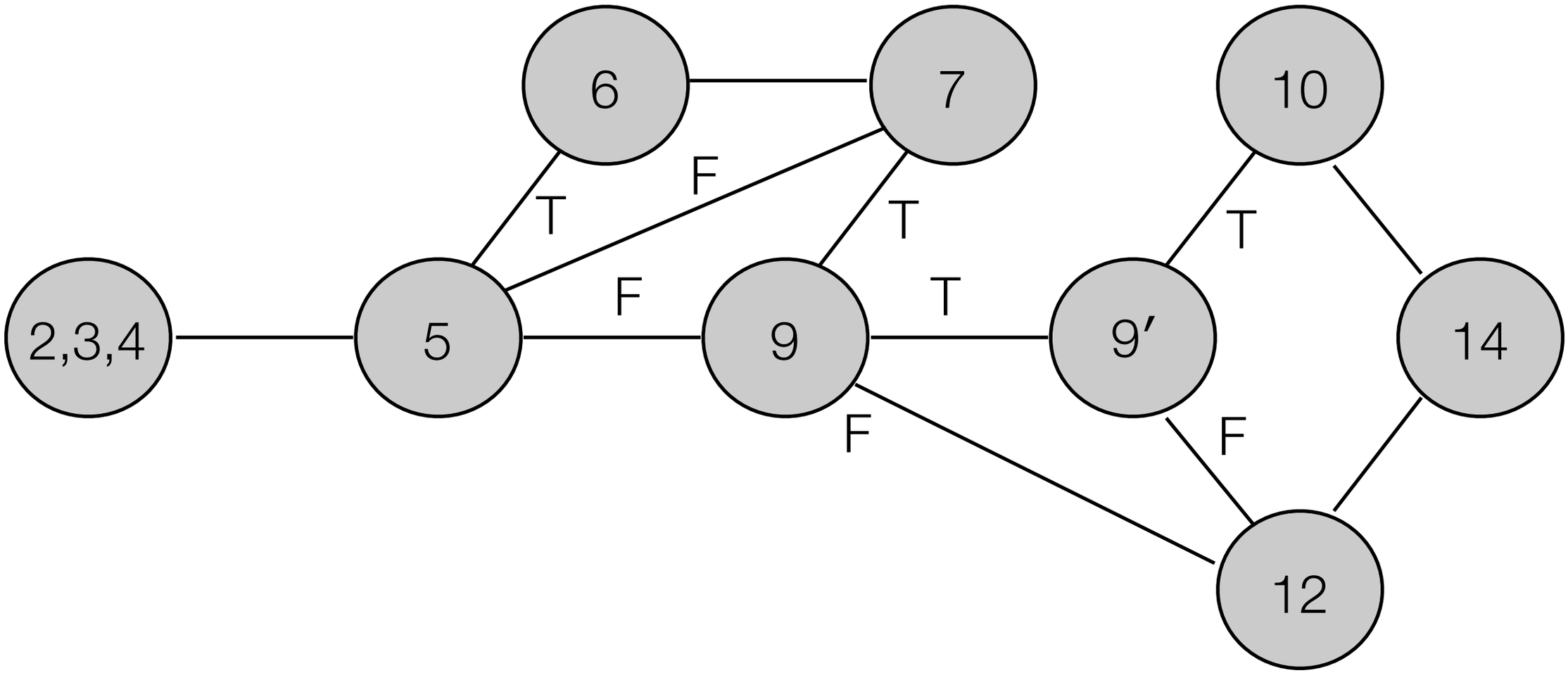
© Kenneth M. Anderson, 2012

White-box Testing Criteria

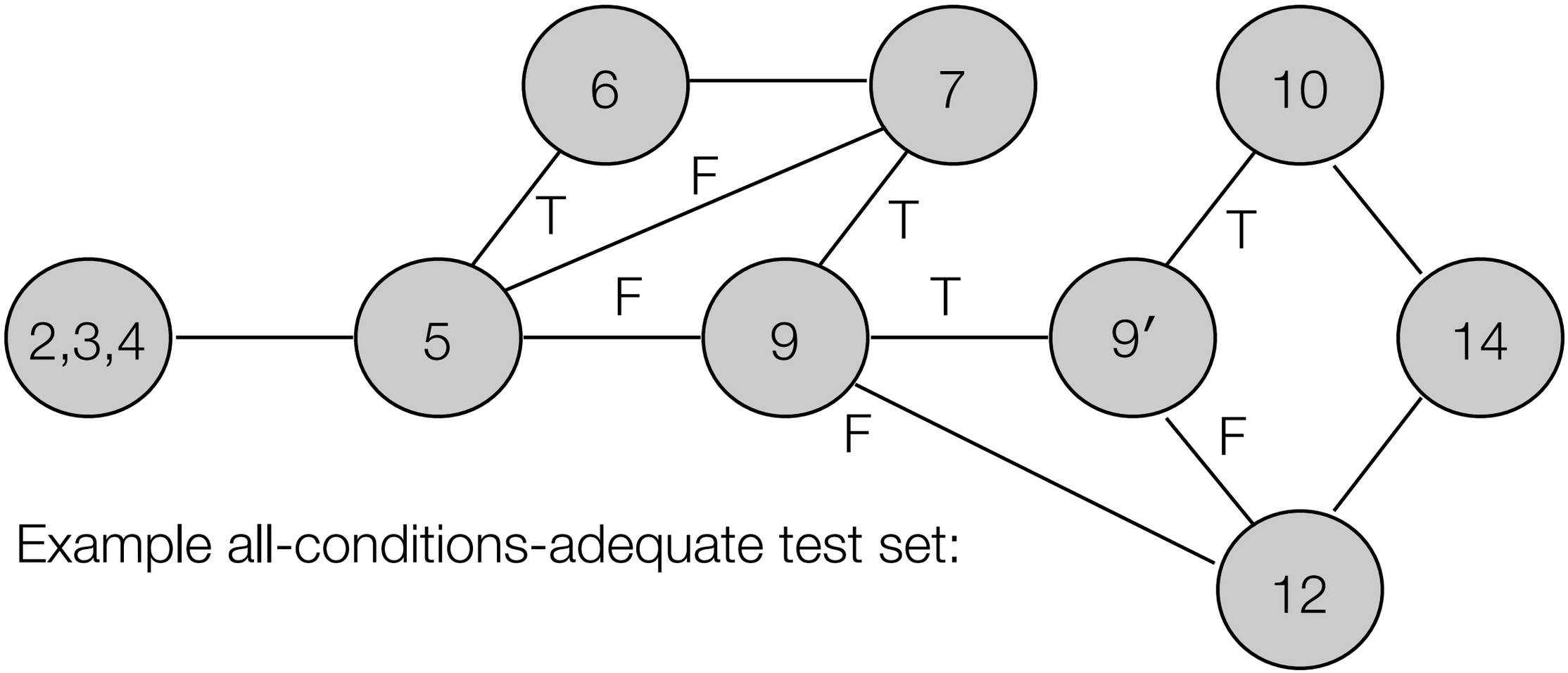
- Condition Coverage
  - Select a test set T such that
    - by executing P for each t in T
      - each edge of P's control flow graph is traversed at least once
      - and all possible values of the constituents of compound conditions are exercised at least once

© Kenneth M. Anderson, 2012

All-Conditions Coverage of P



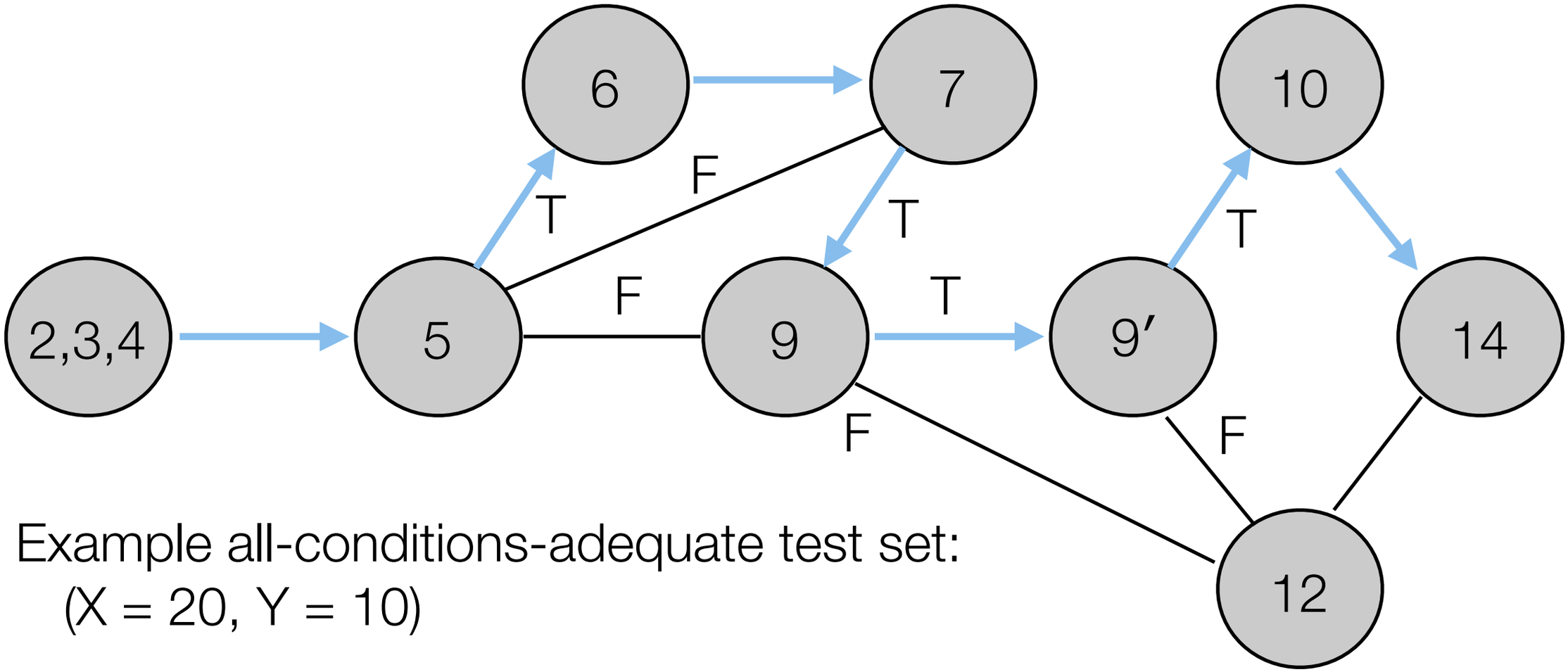
All-Conditions Coverage of P



Example all-conditions-adequate test set:

© Kenneth M. Anderson, 2012

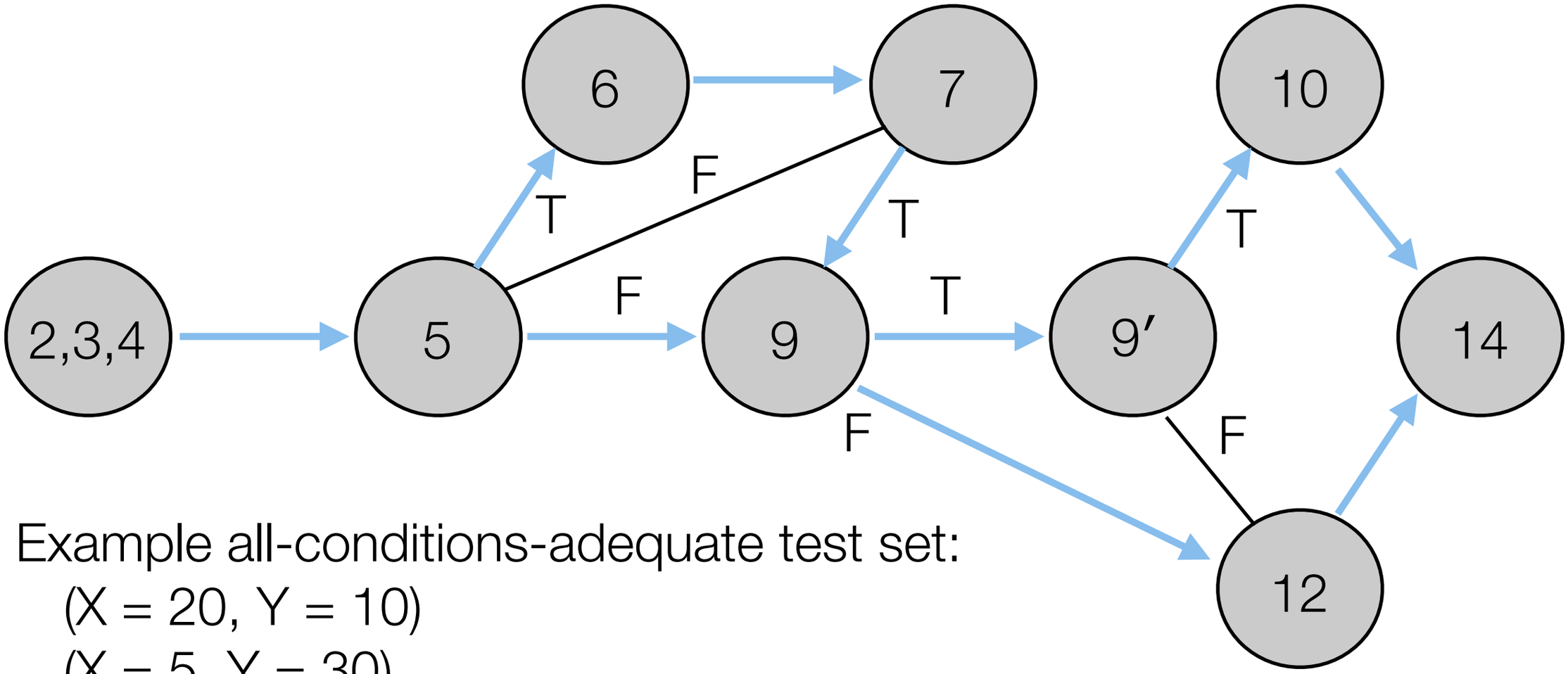
All-Conditions Coverage of P



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

© Kenneth M. Anderson, 2012

All-Conditions Coverage of P



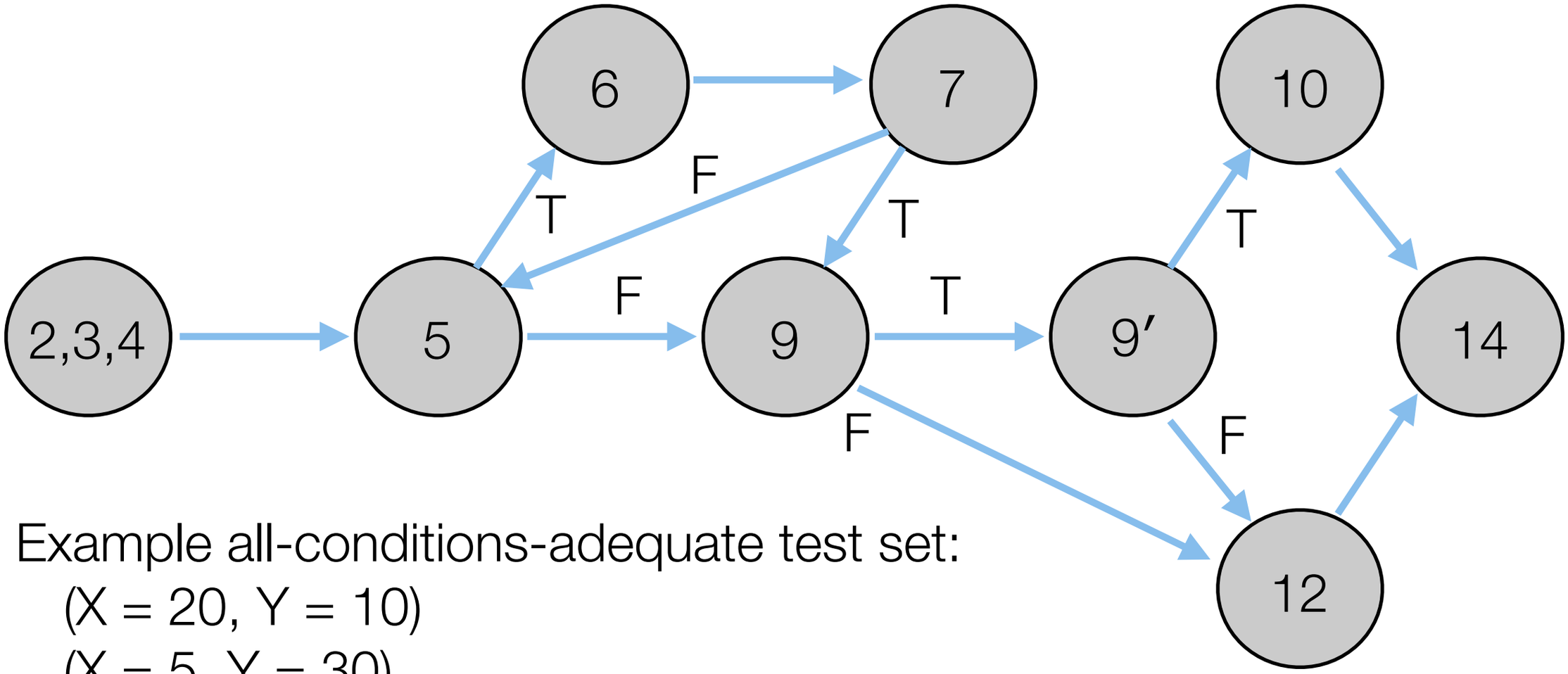
Example all-conditions-adequate test set:

(X = 20, Y = 10)

(X = 5, Y = 30)

© Kenneth M. Anderson, 2012

All-Conditions Coverage of P



Example all-conditions-adequate test set:

(X = 20, Y = 10)

(X = 5, Y = 30)

(X = 21, Y = 10)

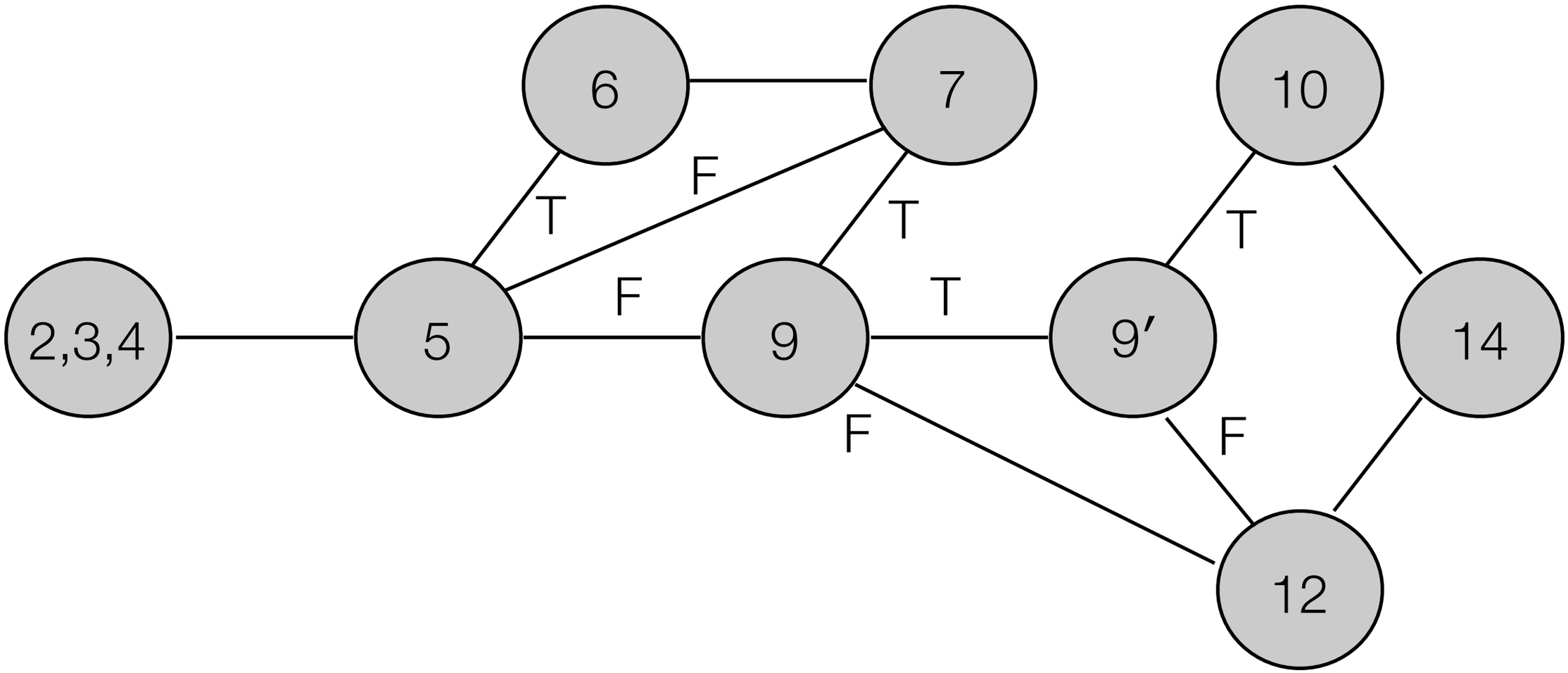
© Kenneth M. Anderson, 2012

White-box Testing Criteria

- 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

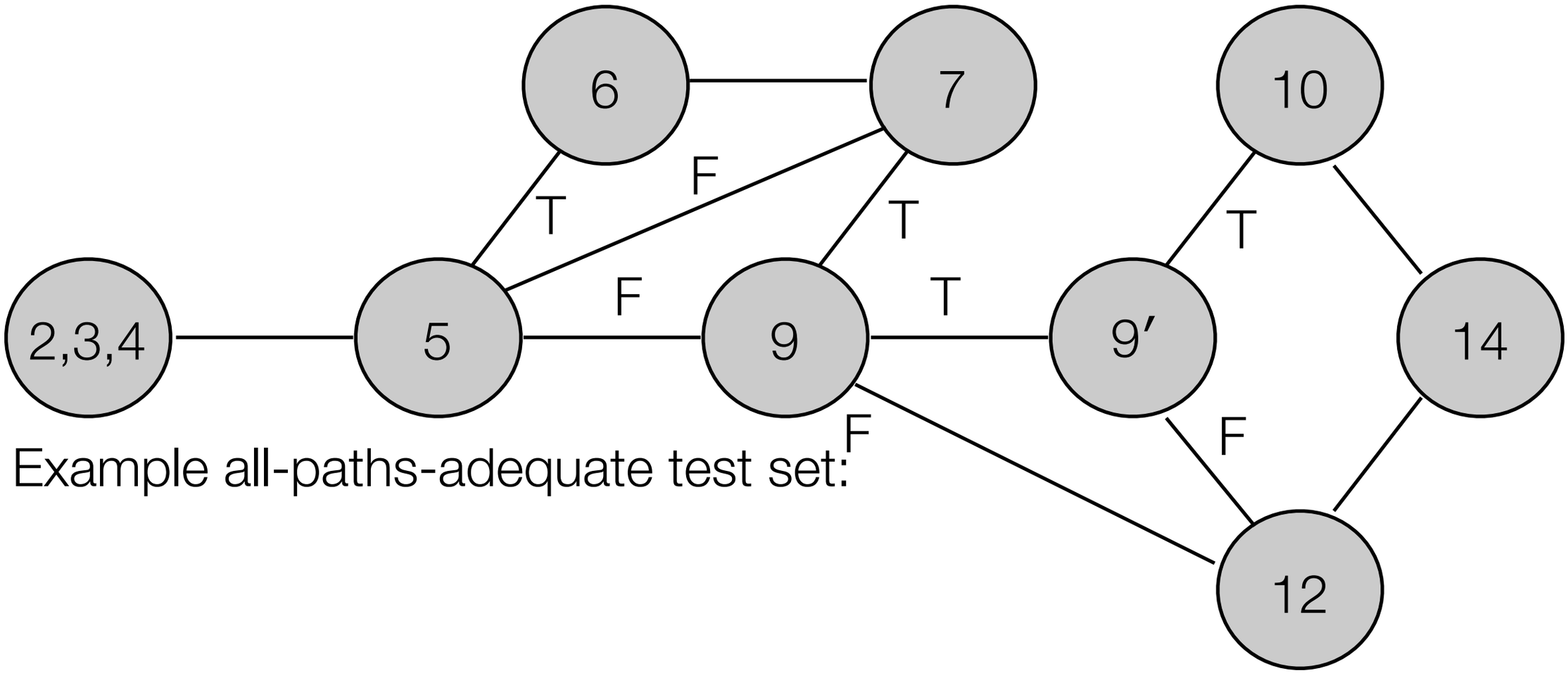
© Kenneth M. Anderson, 2012

All-Paths Coverage of P



© Kenneth M. Anderson, 2012

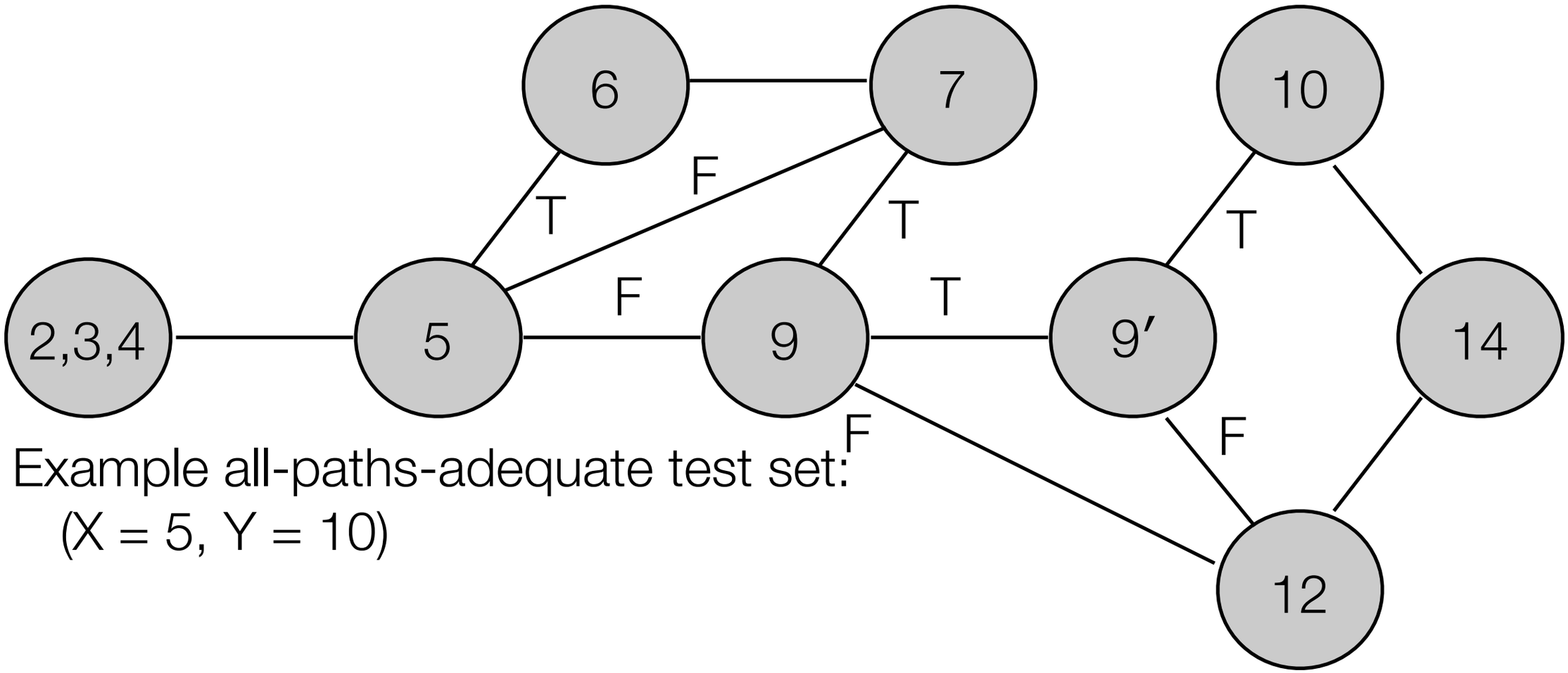
All-Paths Coverage of P



Example all-paths-adequate test set:

© Kenneth M. Anderson, 2012

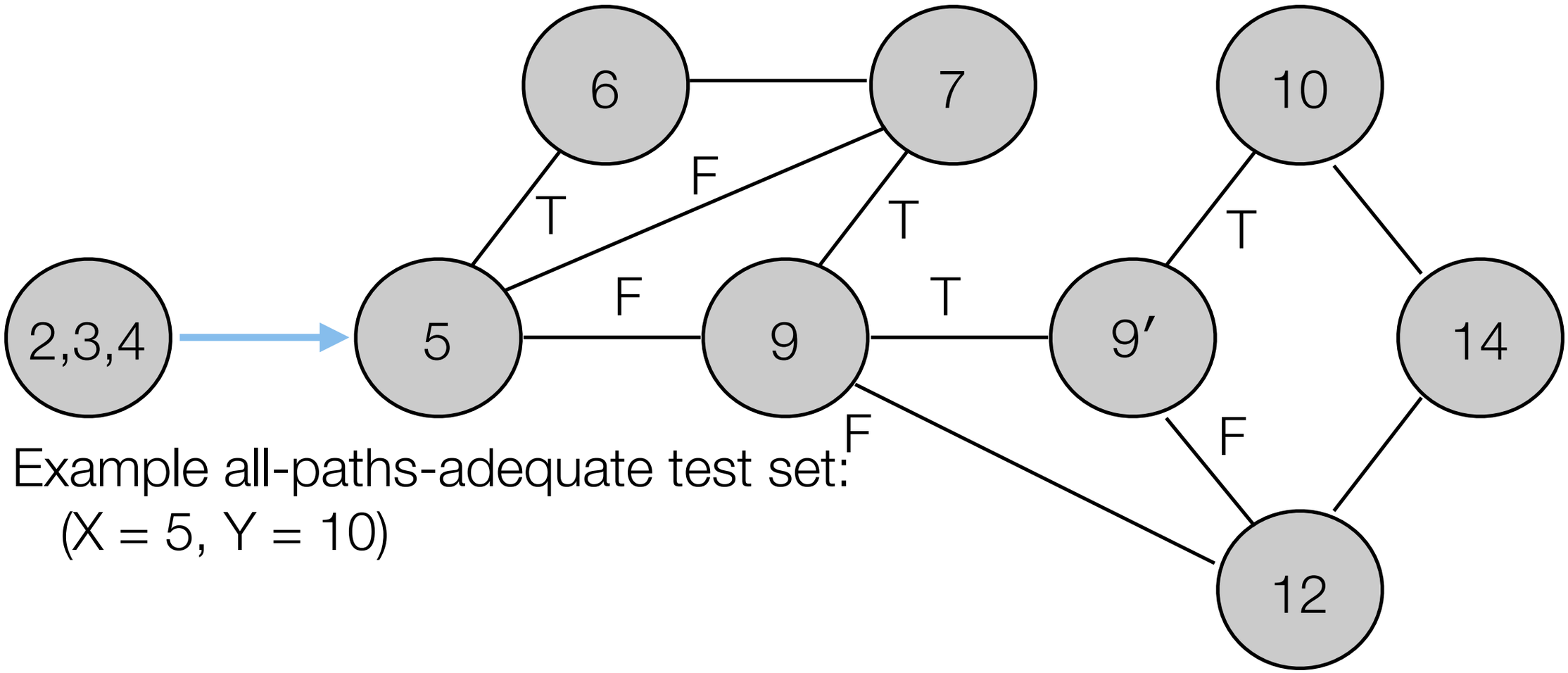
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

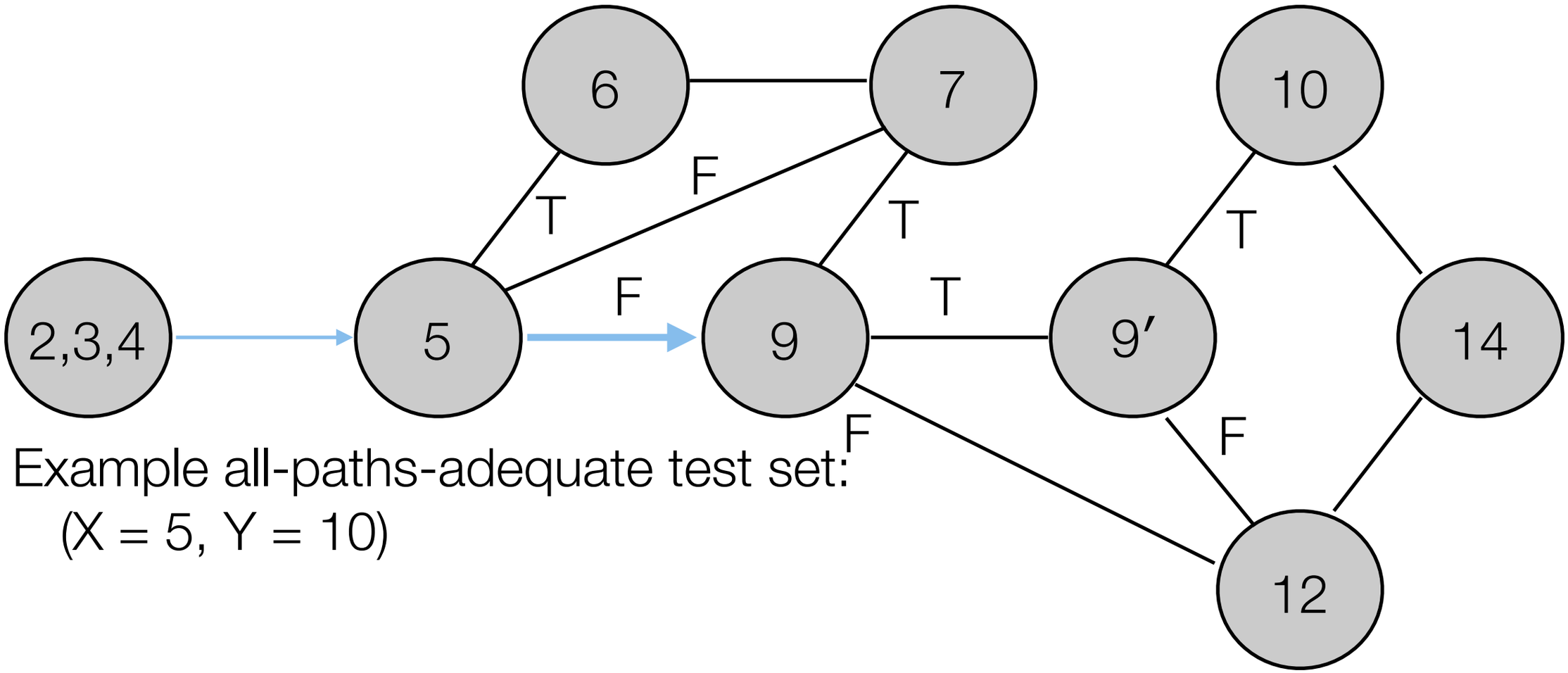
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

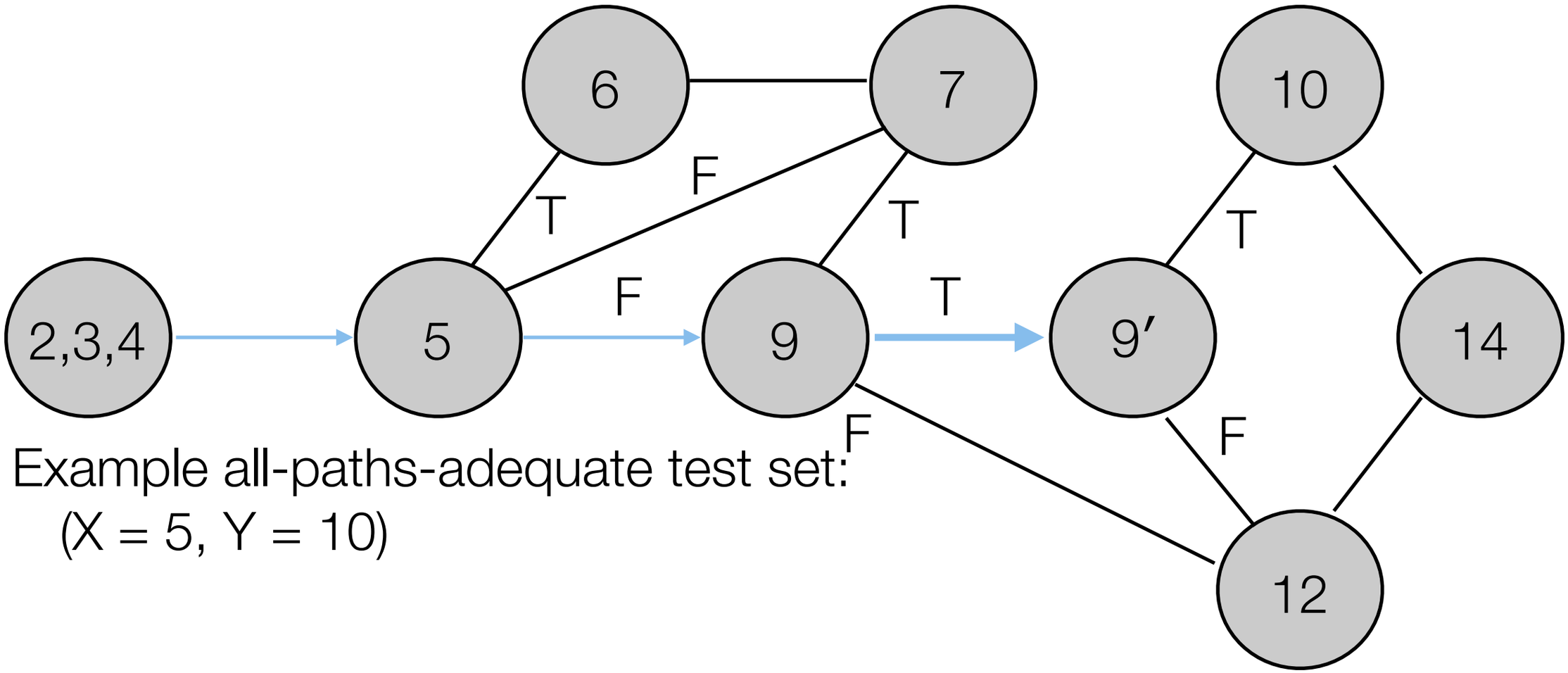
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

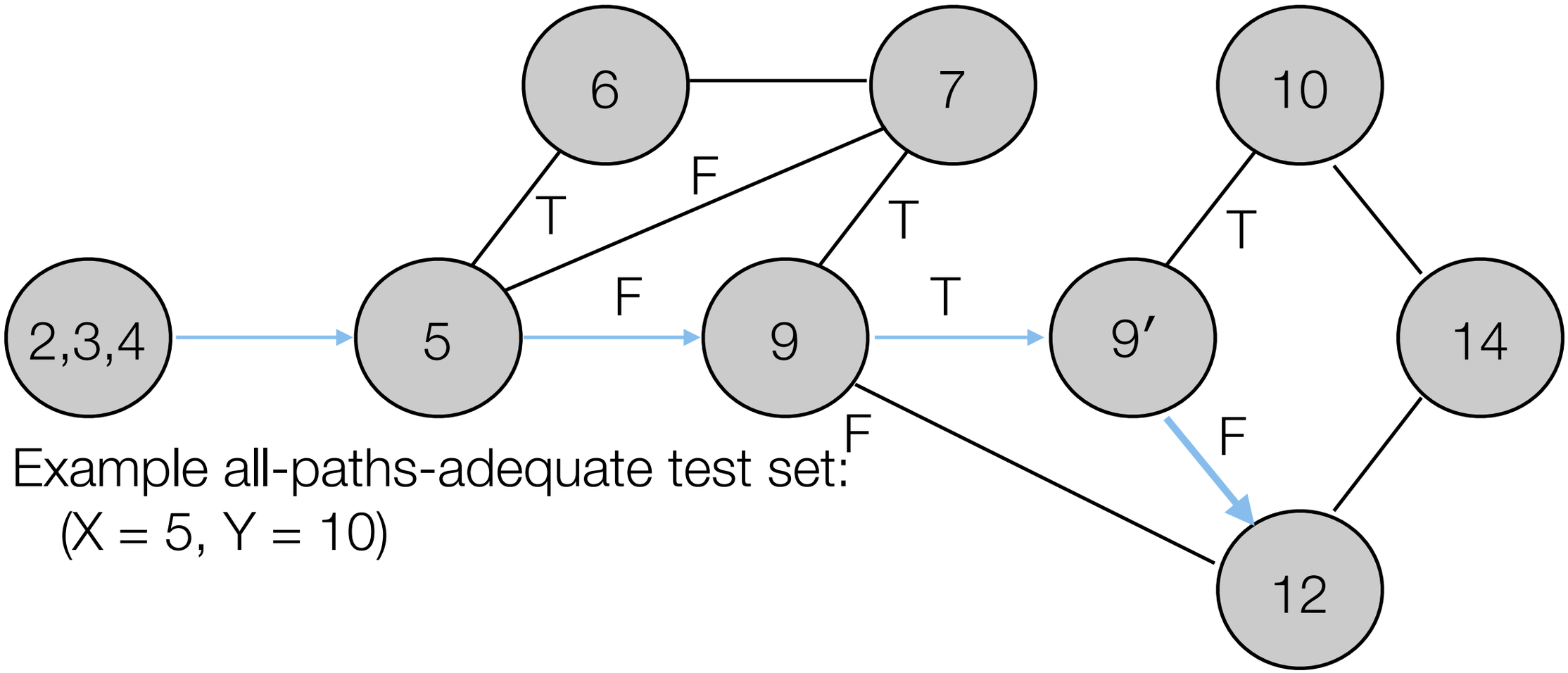
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

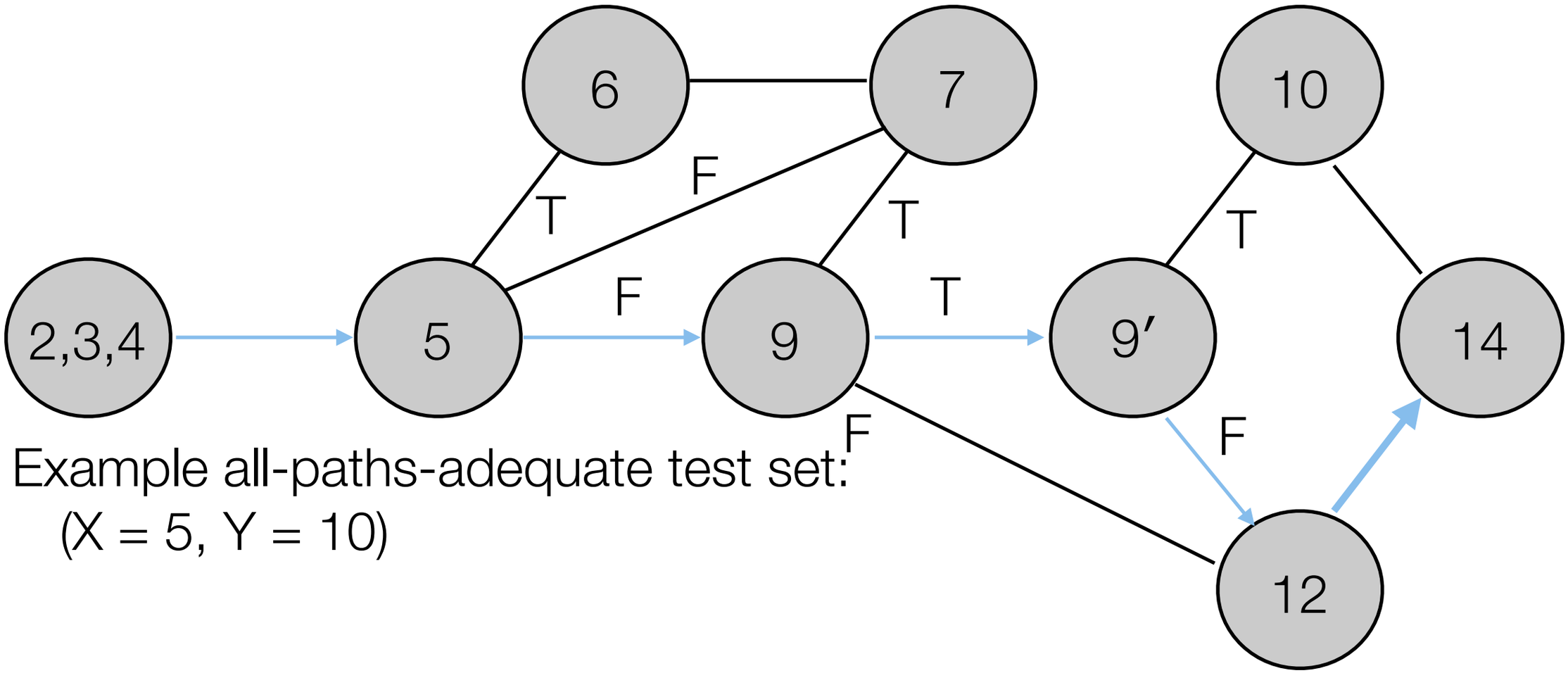
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

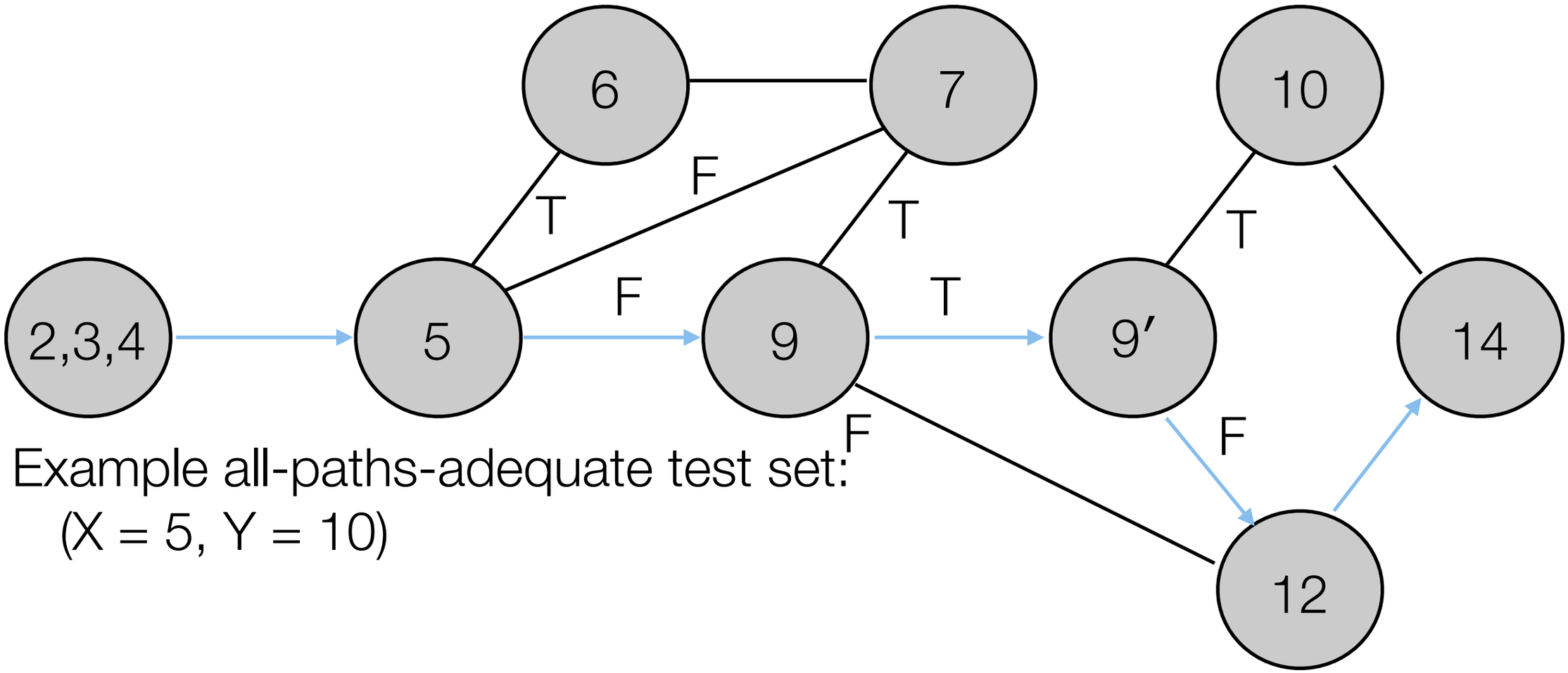
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

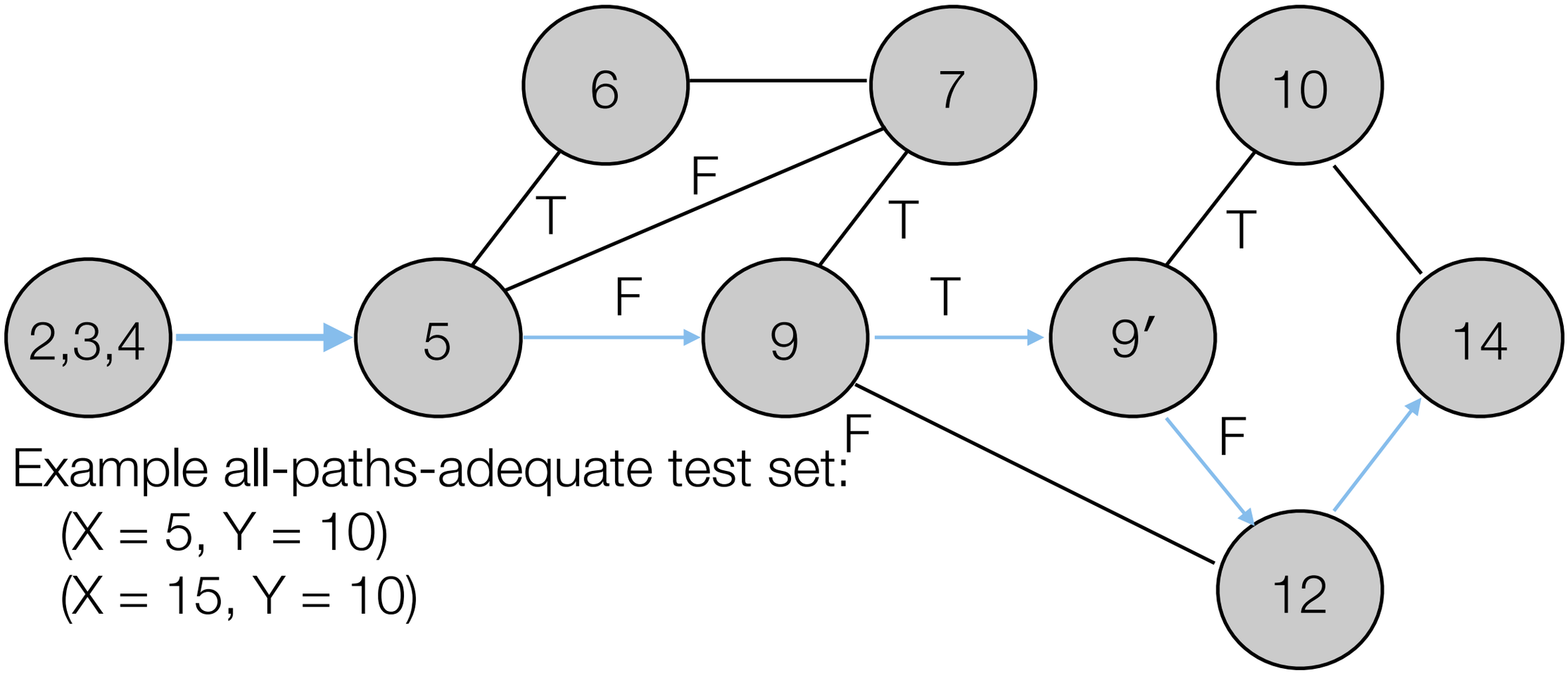
All-Paths Coverage of P



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

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



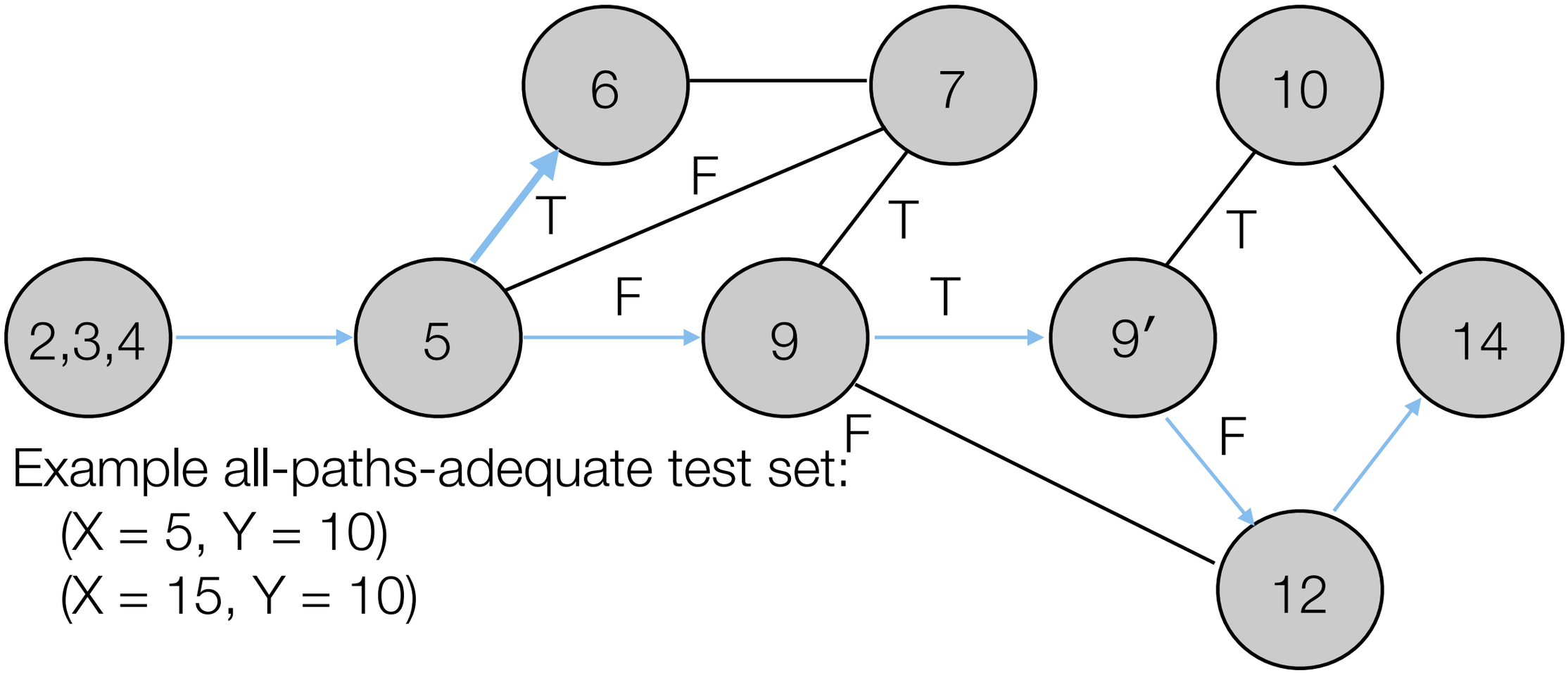
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



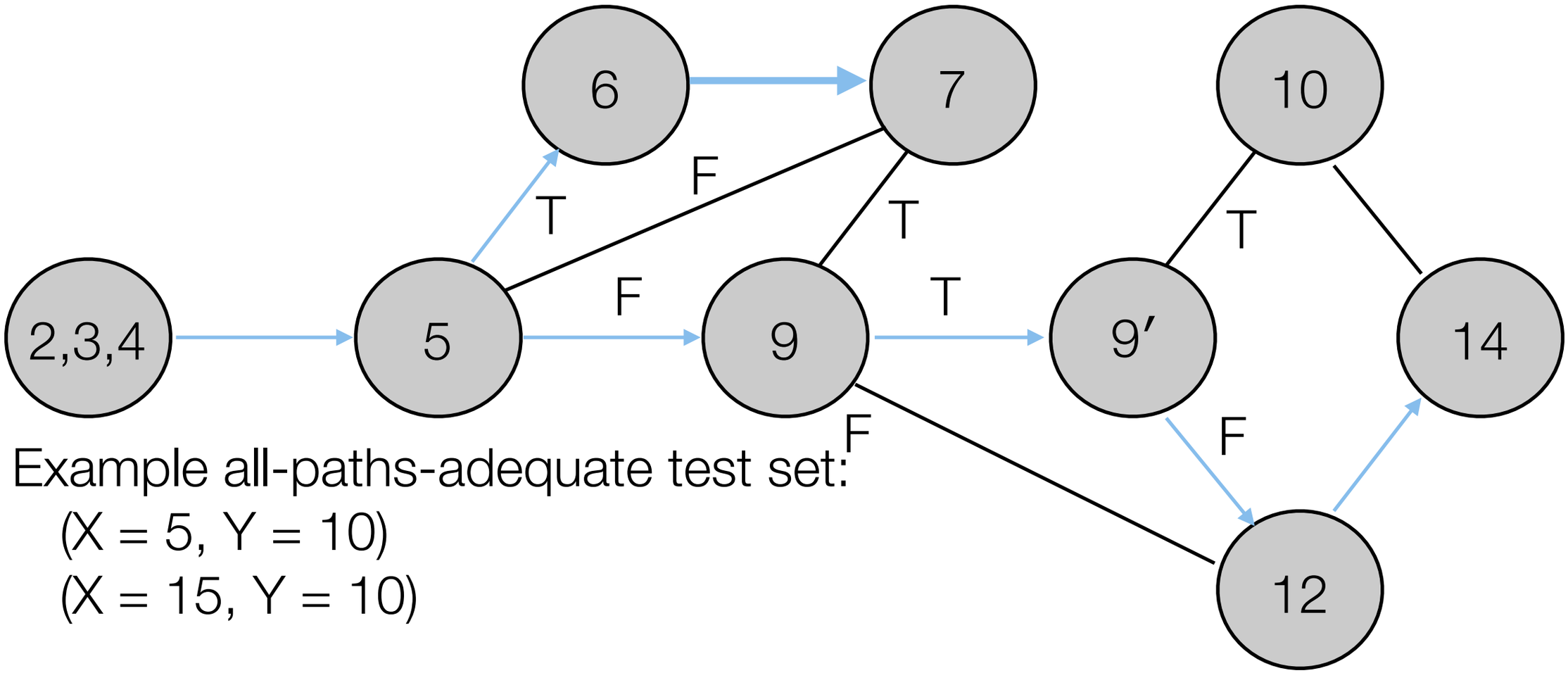
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



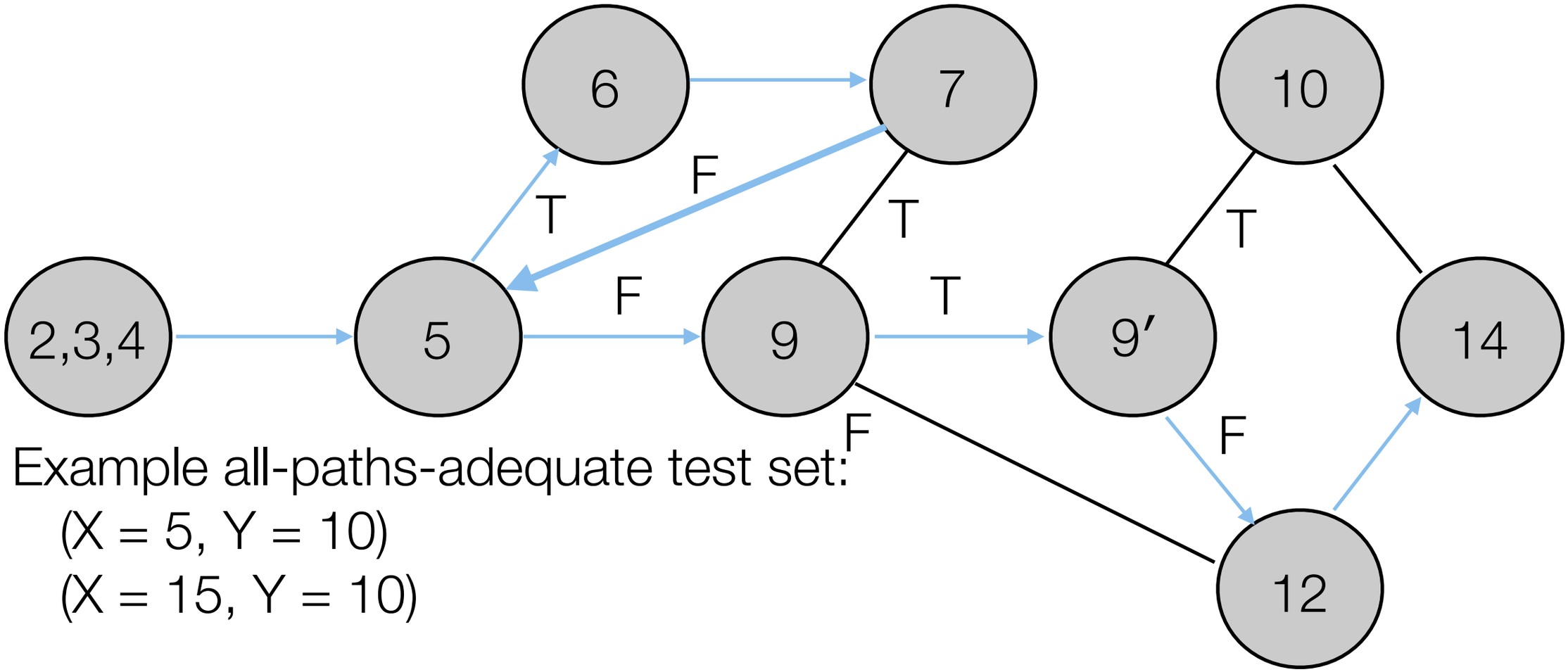
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



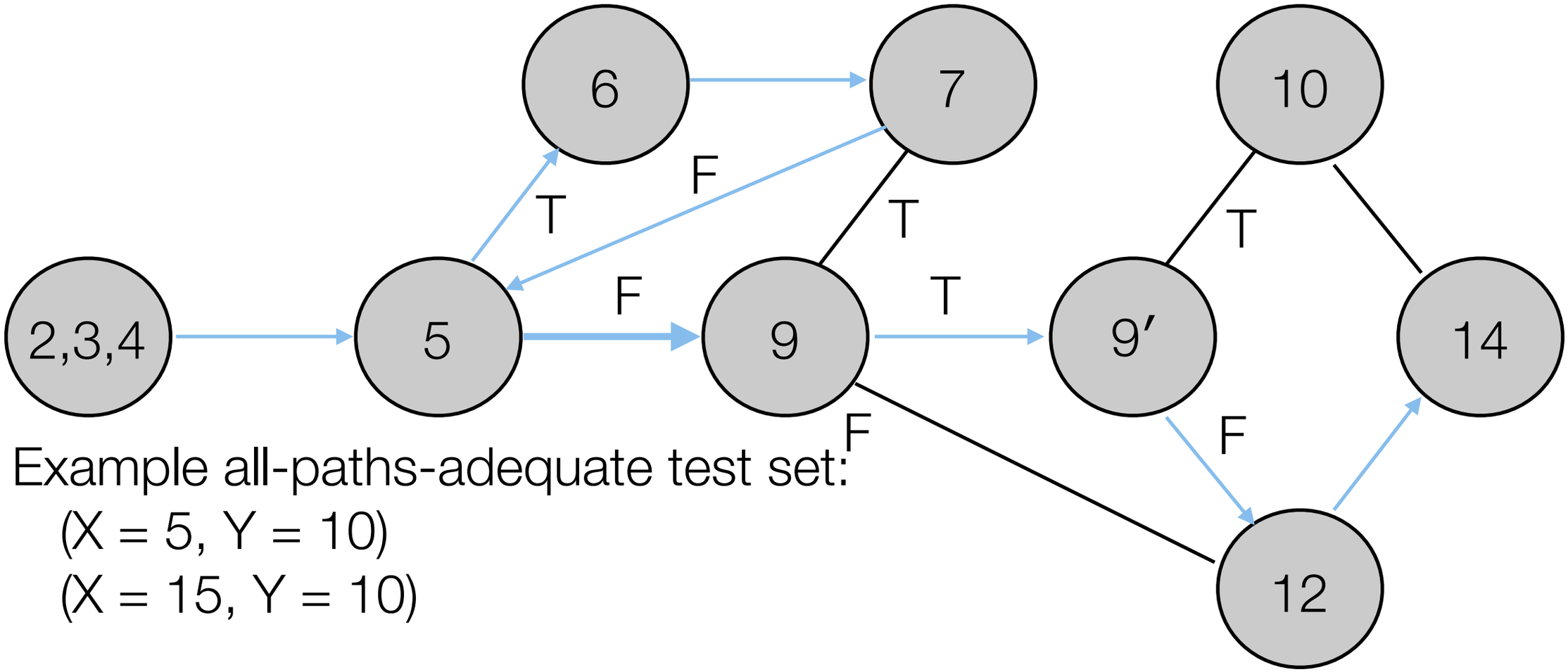
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



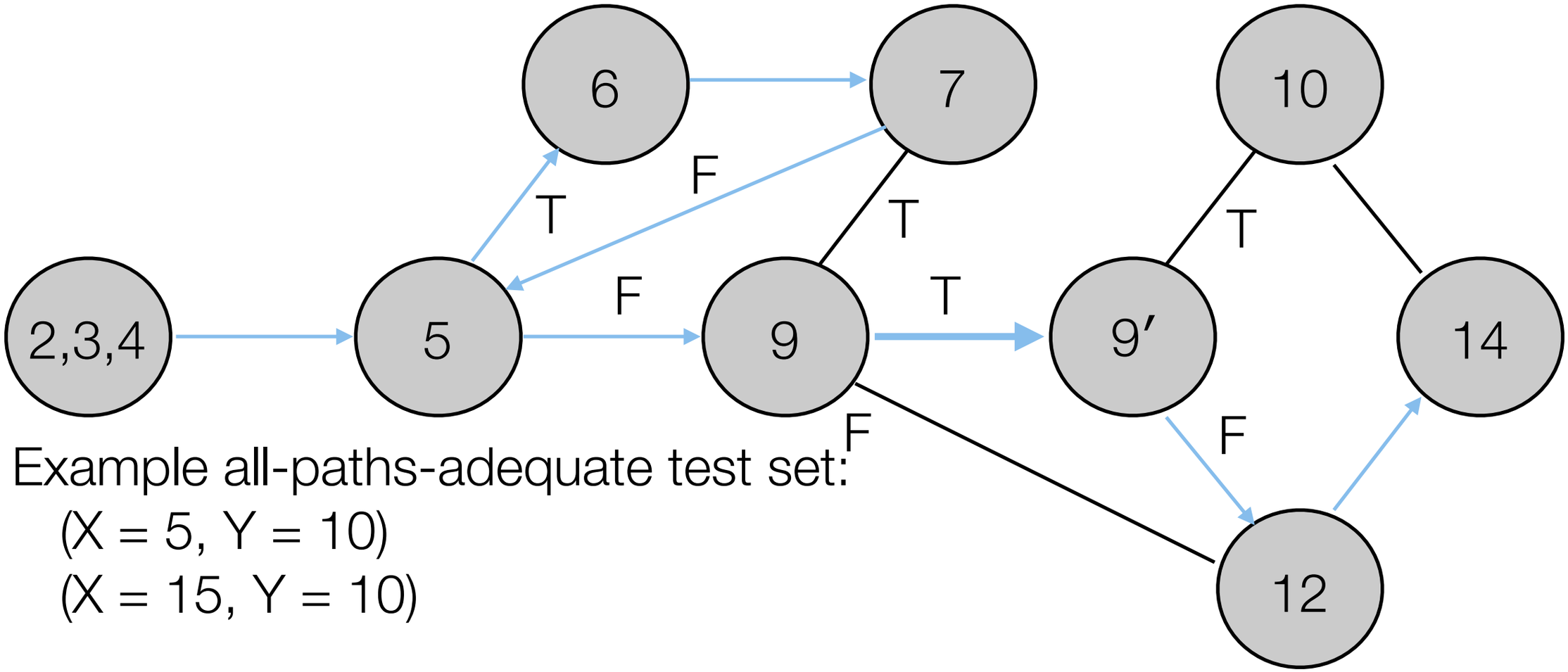
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



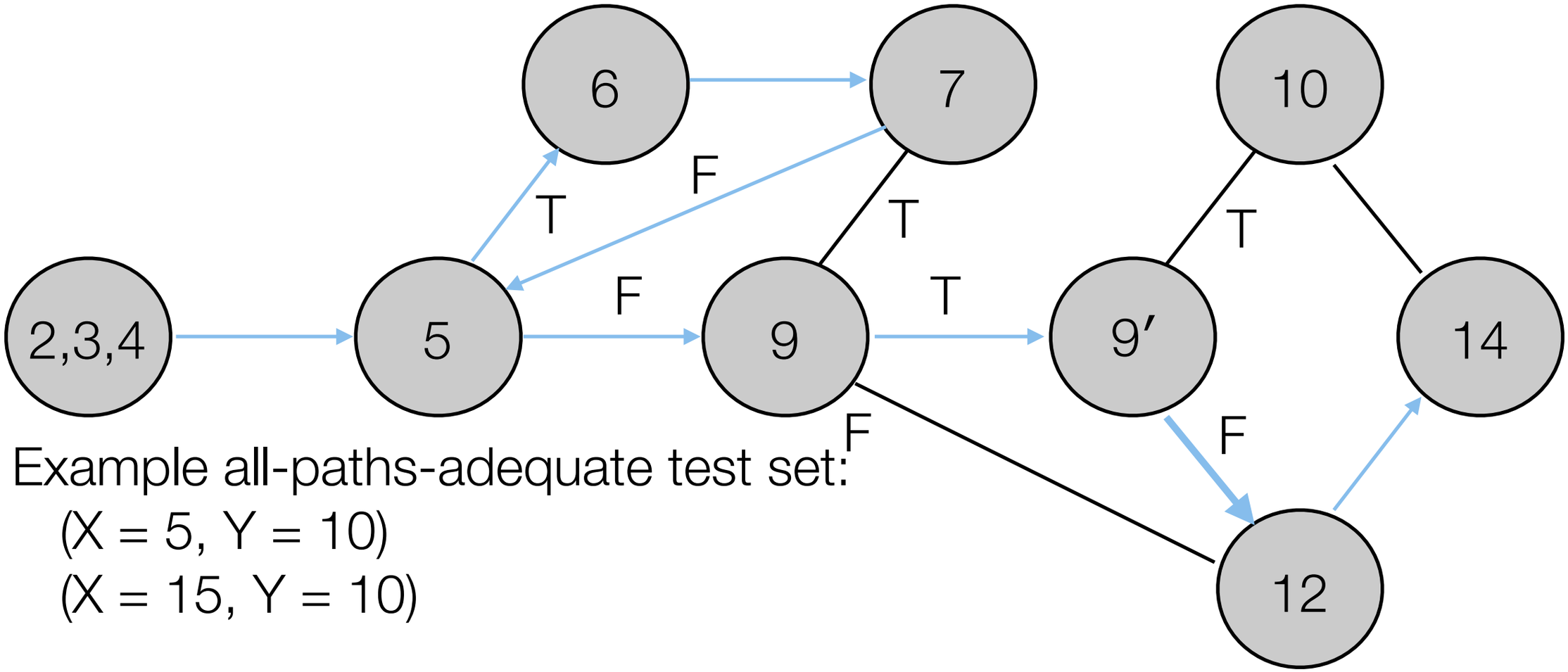
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



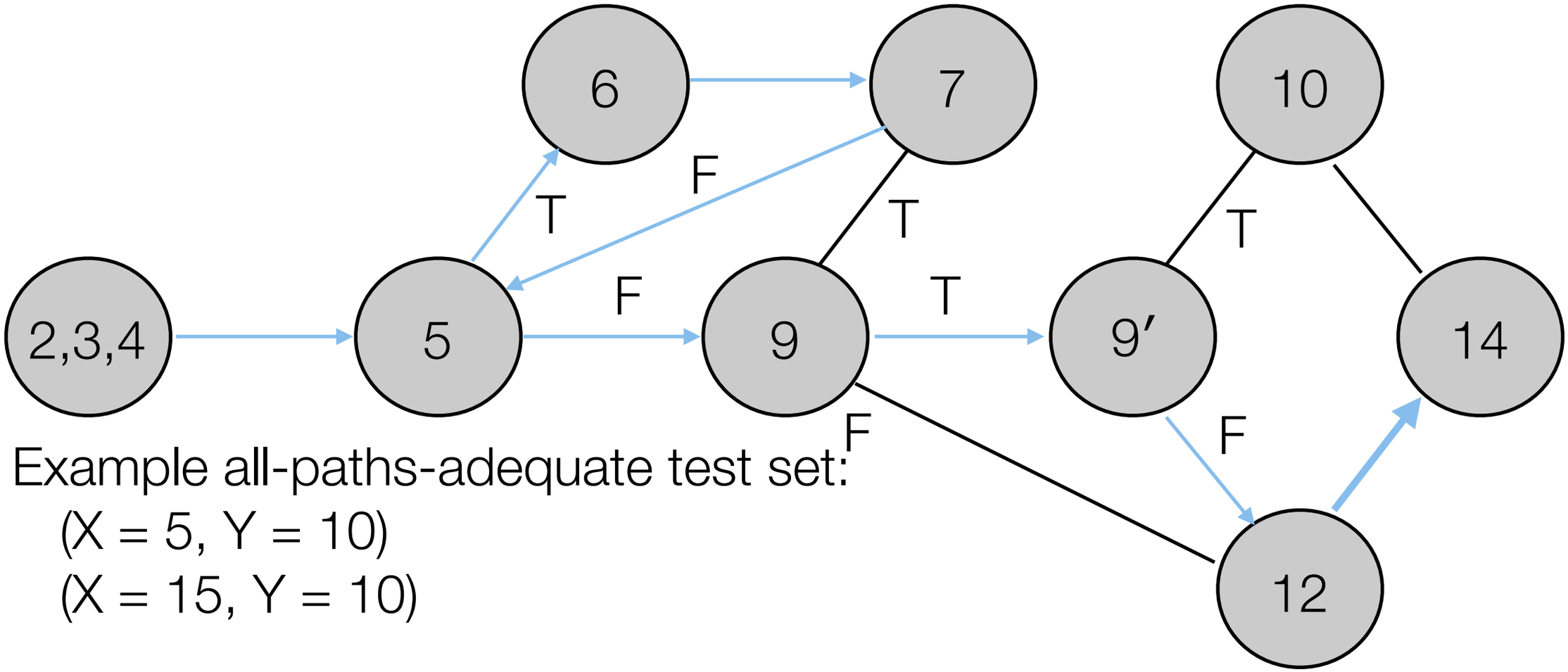
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



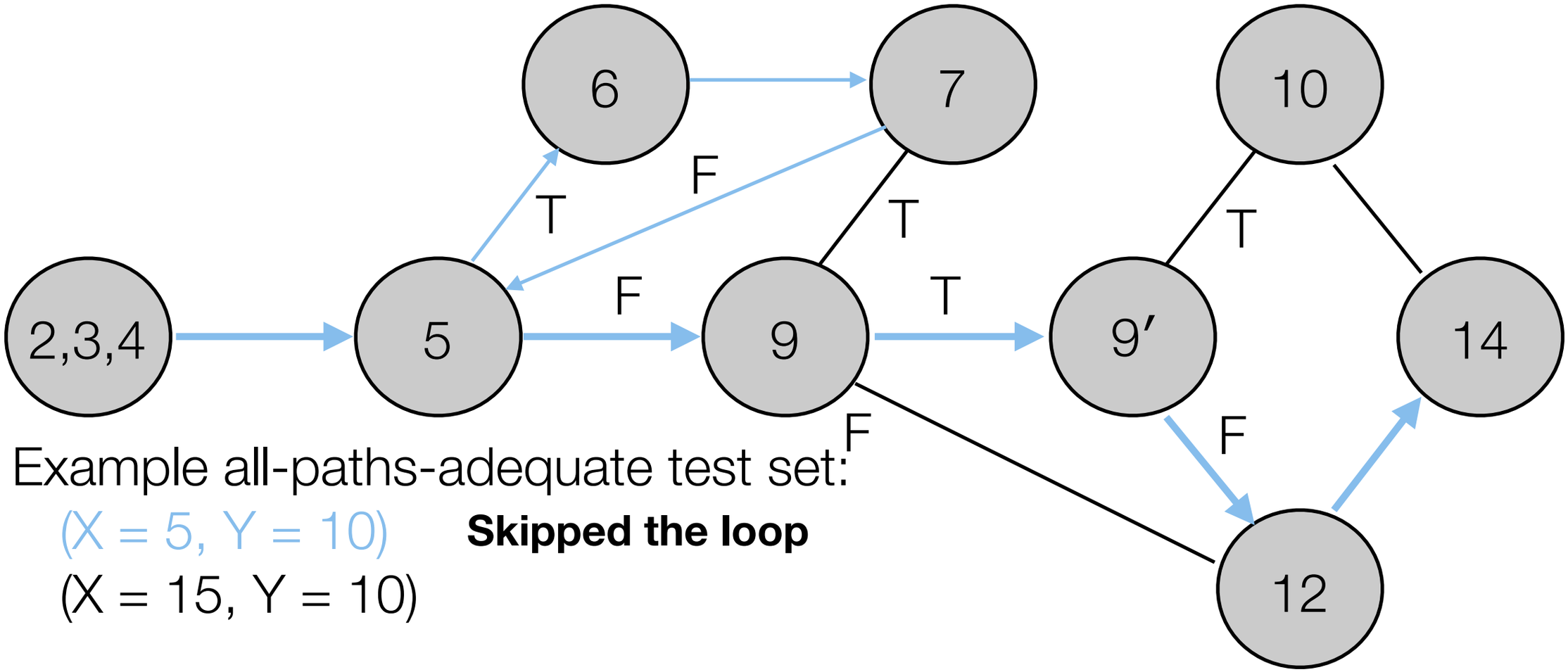
Example all-paths-adequate test set:

(X = 5, Y = 10)

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



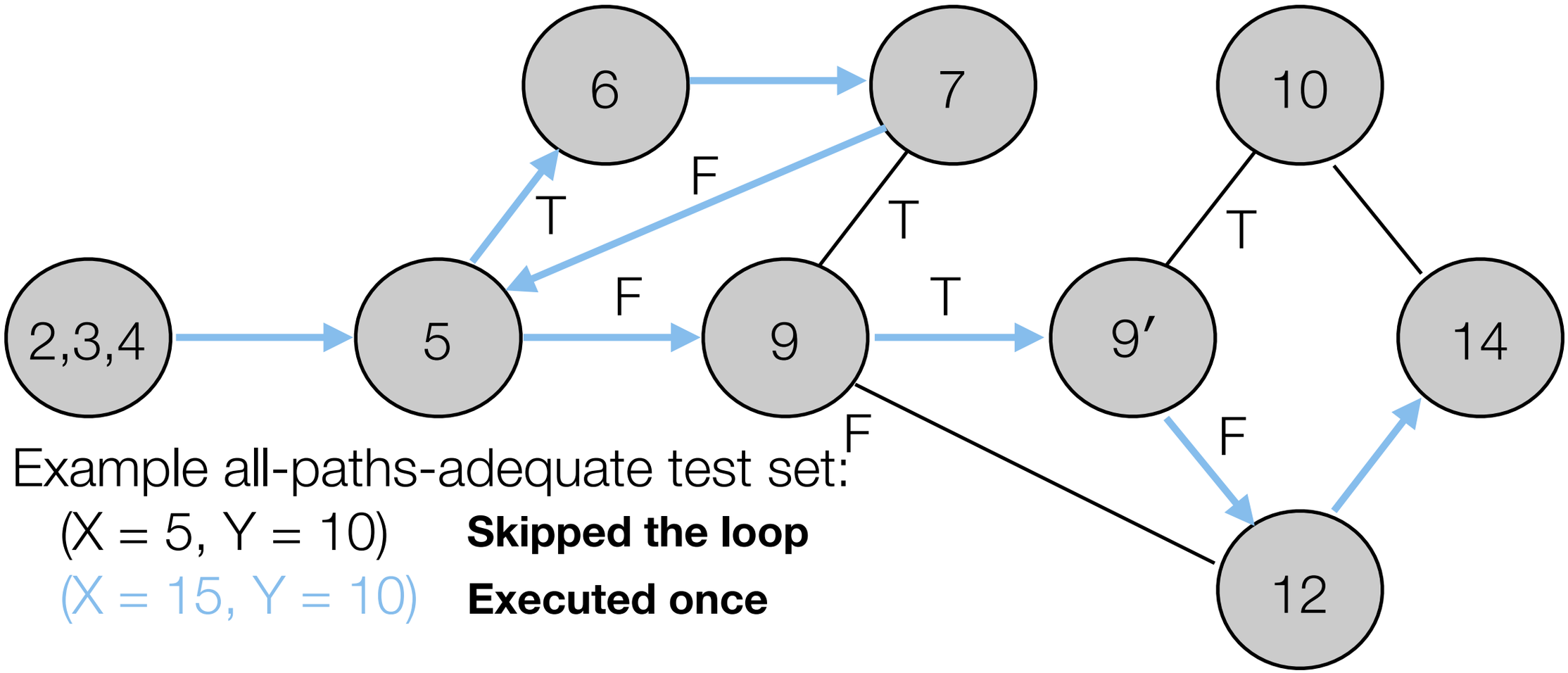
Example all-paths-adequate test set:

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

(X = 15, Y = 10)

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



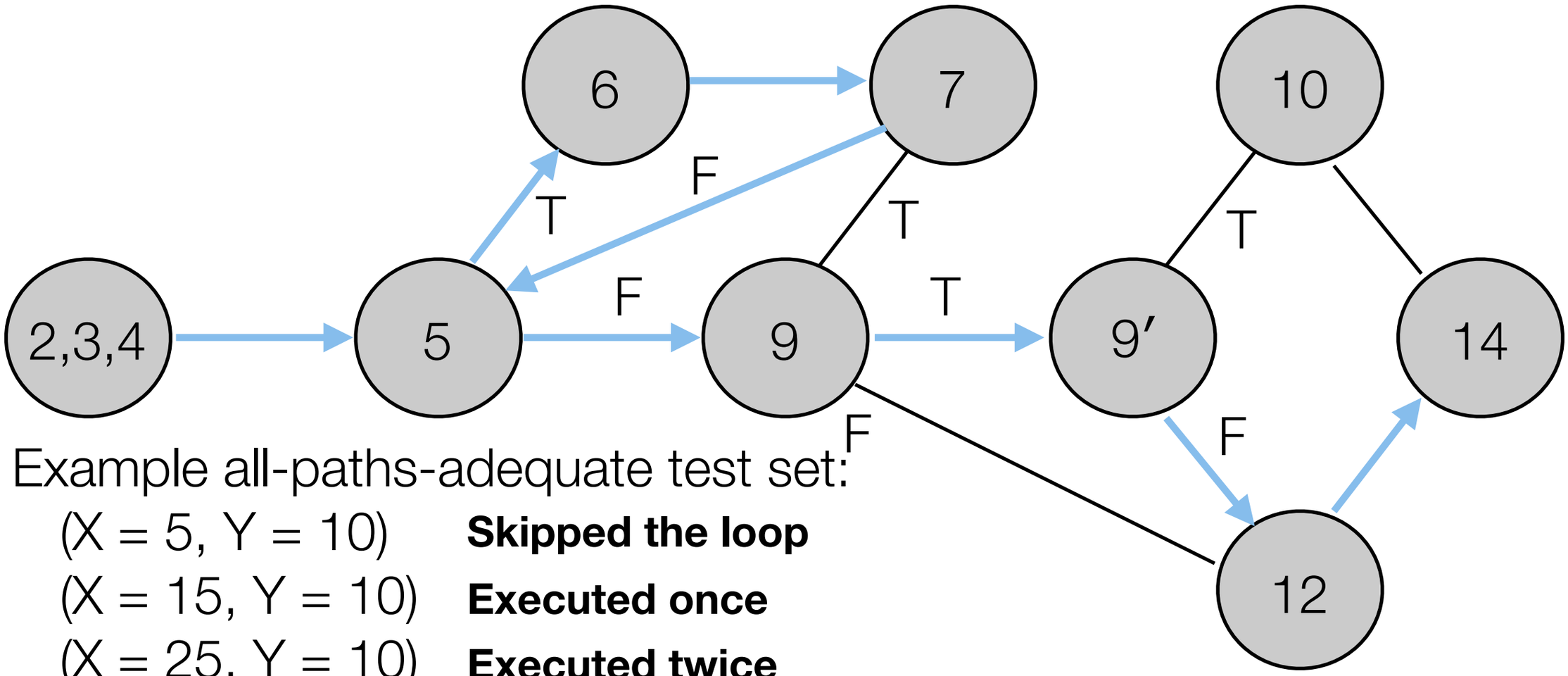
Example all-paths-adequate test set:

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

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

© Kenneth M. Anderson, 2012

All-Paths Coverage of P



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**

And so on… you would also want permutations that exit the loop early

© Kenneth M. Anderson, 2012

Code Coverage Tools

- 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
    - These systems 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

© Kenneth M. Anderson, 2012

Testing Automation (I)

- 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!!)

© Kenneth M. Anderson, 2012

Testing Automation (II)

- 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
- We'll see examples of this later in the semester

© Kenneth M. Anderson, 2012

Continuous Integration

- Since test automation is so critical, systems known as continuous integration frameworks have emerged
- 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

© Kenneth M. Anderson, 2012

Summary

- 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 exists to integrate testing into the version control and build management processes of a development organization

© Kenneth M. Anderson, 2012

Coming Up Next:

- Lecture 6: Agile Methods and Agile Teams
- Lecture 7: Division of Labor and Design Approaches in Concurrency

© Kenneth M. Anderson, 2012