Introduction to Software Engineering

CSCI 5828: Foundations of Software Engineering
Lecture 04 — 09/03/2015

Returning to SE Intro

- Lets continue our “Overview of Software Engineering” that was started in Lecture 1
    - This draws on material from Software Engineering: Theory and Practice by Pfleeger and Atlee
        - As such, some material is copyright © 2006 Pearson/Prentice Hall.

What is Software Engineering?

- Simply Put: It is **solving problems with software-based systems**
  - Design and development of these systems require
    - **Analysis**
      - decomposing large problems into smaller, understandable pieces
        - abstraction is the key
    - **Synthesis**
      - building large software systems from smaller building blocks
        - composition is challenging

Solving Problems (I)

- To aid us in solving problems, we apply techniques and tools
  - **techniques**: a formal “recipe” for accomplishing a goal that is typically independent of the tools used
    - automated builds, configuration management, software testing, etc.
  - **tools**: an instrument or automated system for accomplishing something in a better way, where “better” can mean more efficient, more accurate, faster, etc.
    - maven, git, jenkins, etc.

Solving Problems (II)

- To aid us in solving problems, we apply
  - **procedures**: a **combination of tools and techniques** that, in concert, **produce a particular product**
  - **paradigms**: a particular philosophy or approach for building a product
    - Think: “cooking style”: may share procedures, tools, and techniques with other styles but apply them in different ways
    - By analogy: **OO approach** to development vs. **the structured approach**
      - Both approaches use similar things:
        - reqs., design, code, editors, compilers, etc.
      - But think about the problem in fundamentally different ways

Software Engineering: The Good

- Software engineering has helped to produce systems that improve our lives in numerous ways
  - helping us to perform tasks more quickly and effectively
  - supporting advances in medicine, agriculture, communication, transportation, and other industries
- Indeed, software-based systems are now ubiquitous

Software Engineering: The Bad (I)

- Software is not without its problems
  - Systems function, but not in the way we expect
  - Or systems crash, generate the wrong output, etc.
  - Or the process for producing a system is riddled with problems leading to a failure to produce the entire system
    - many projects get cancelled without ever producing a system
- One study in the late 80s found that in a survey of 600 firms, more than 35% reported having a **runaway development project**. A runway project is one in which the budget and schedule are completely out of control.

Software Engineering: The Bad (II)

- CHAOS Report from Standish Group
    - Has studied over 40,000 industry software development projects over the course of 1994 to 2004.
    - Success rates (projects completed on-time, within budget) in 2004 was 34%, up from 16.2% in 1994
    - Failure rates (projects cancelled before completion) in 2004 was 15%, down from 31% in 1994.
    - In 2004, “challenged” projects made up 51% of the projects included in the survey.
        - A challenged project is one that was over time, over budget and/or missing critical functionality

Software Engineering: The Bad (III)

- Most challenged projects in 2004 had a cost overrun of under 20% of the budget, compared to 60% in 1994
- The average cost overrun in 2004 was 43% versus an average cost overrun of 180% in 1994.
- In 2004, total U.S. project waste was 55 billion dollars with 17 billion of that in cost overruns; Total project spending in 2004 was 255 billion
    - In 1994, total U.S. project waste was 140 billion (80 billion from failed projects) out of a total of 250 billion in project spending

Software Engineering: The Bad (IV)

- So, things are getting better (attributed to better project management skills industry wide), but we still have a long way to go.
  - 66% of the surveyed projects in 2004 did not succeed!

Software Engineering: The Ugly (I)

- Loss of NASA's Mars Climate Observer
  - due to mismatch of English and Metric units!
  - even worse: problem was known but politics between JPL and Houston prevented fix from being deployed
- Denver International Airport
  - Luggage system: 16 months late, 3.2 billion dollars over budget!
- IRS hired Sperry Corporation to build an automated federal income tax form processing process
  - An extra $90 M was needed to enhance the original $103 M product
  - IRS lost $40.2 M on interest and $22.3 M in overtime wages because refunds were not returned on time

Software Engineering: The Ugly (II)

- Therac-25 (safety critical system: failure poses threat to life or health)
  - Machine had two modes:
    - “electron beam” and “megavolt x-ray”
  - “megavolt” mode delivered x-rays to a patient by colliding high energy electrons into a “target”
    - Patients died when a “race condition” in the software allowed the megavolt mode to engage when the target was not in position
    - Related to a race between a “type ahead” feature in the user interface and the process for rotating the target into position

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

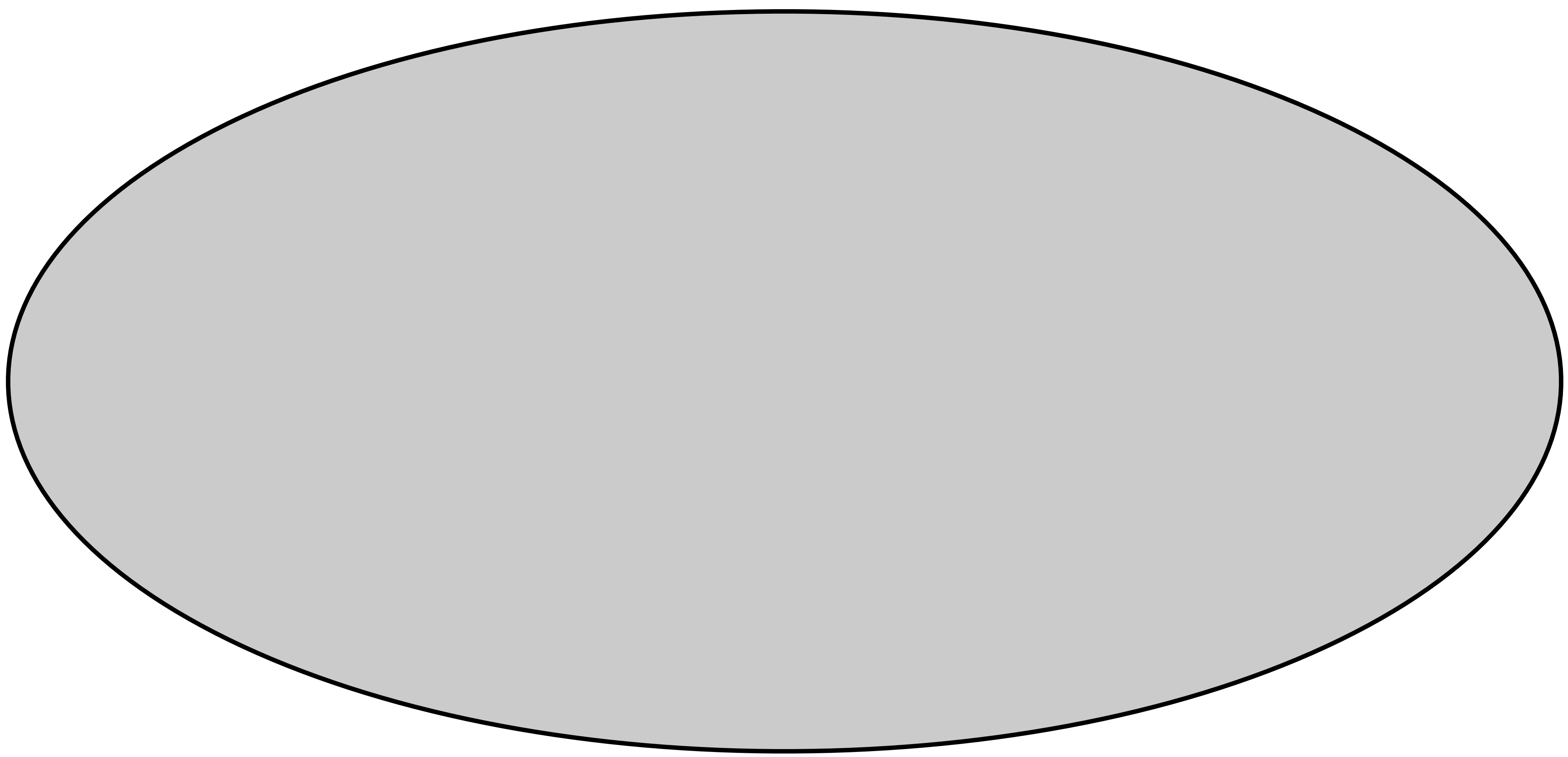
Terminology

- 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?

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

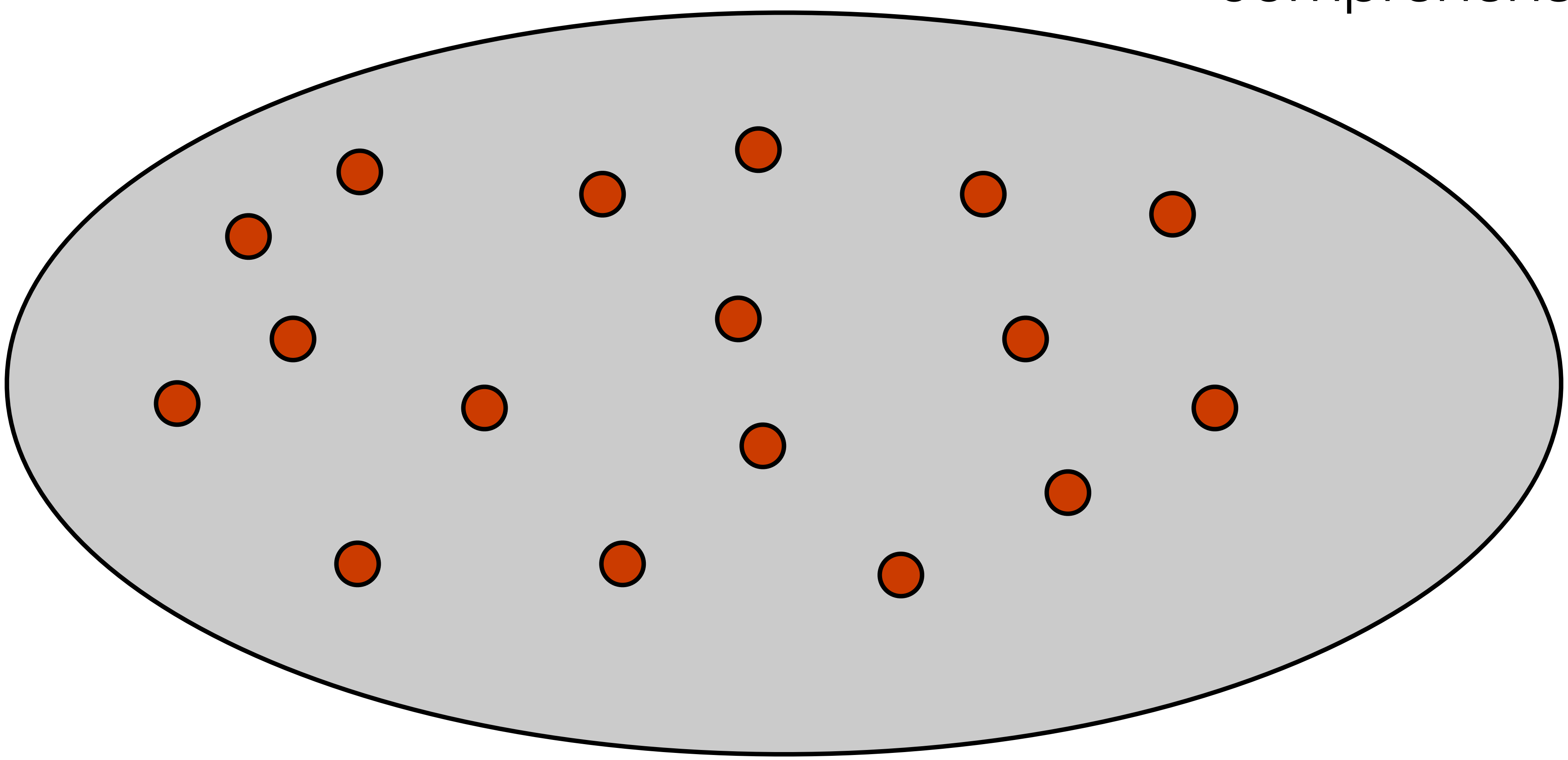
Looking for Faults



All possible states/behaviors of a system

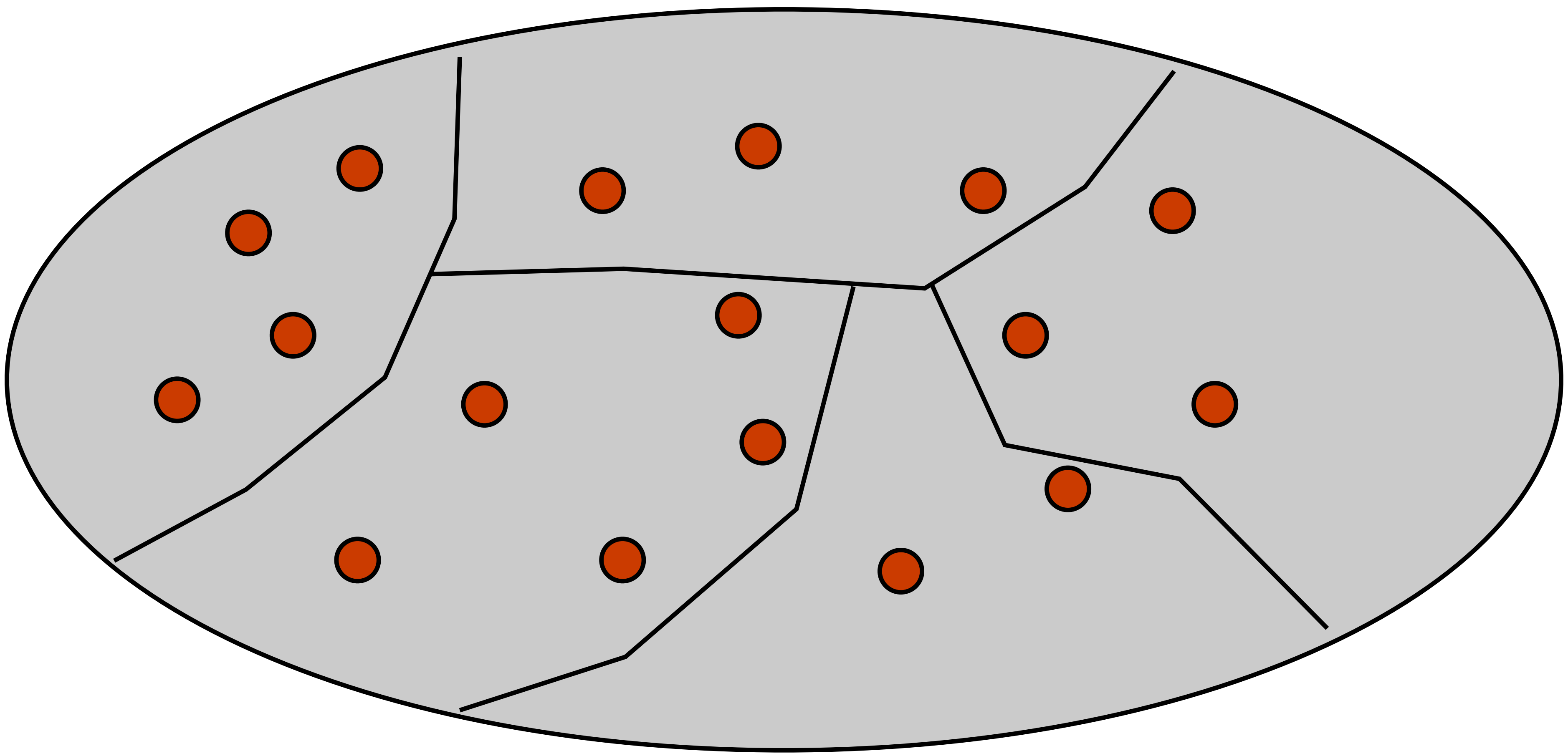
Looking for Faults

As you can see, its not very comprehensive



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

One way forward? Fold



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

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

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?

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)

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)

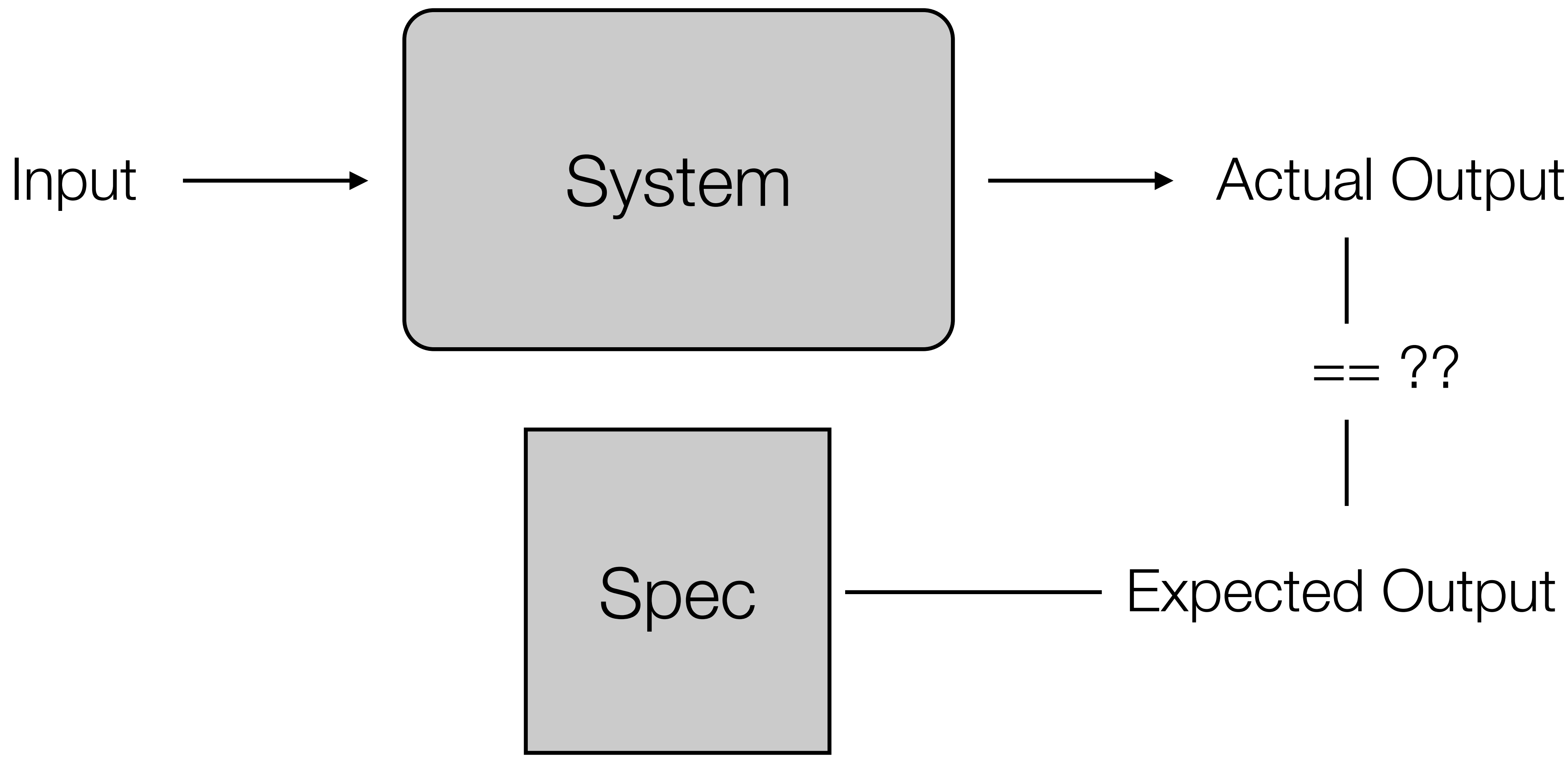
Testing the System (II)

- **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

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, …

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!

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

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

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

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?

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)

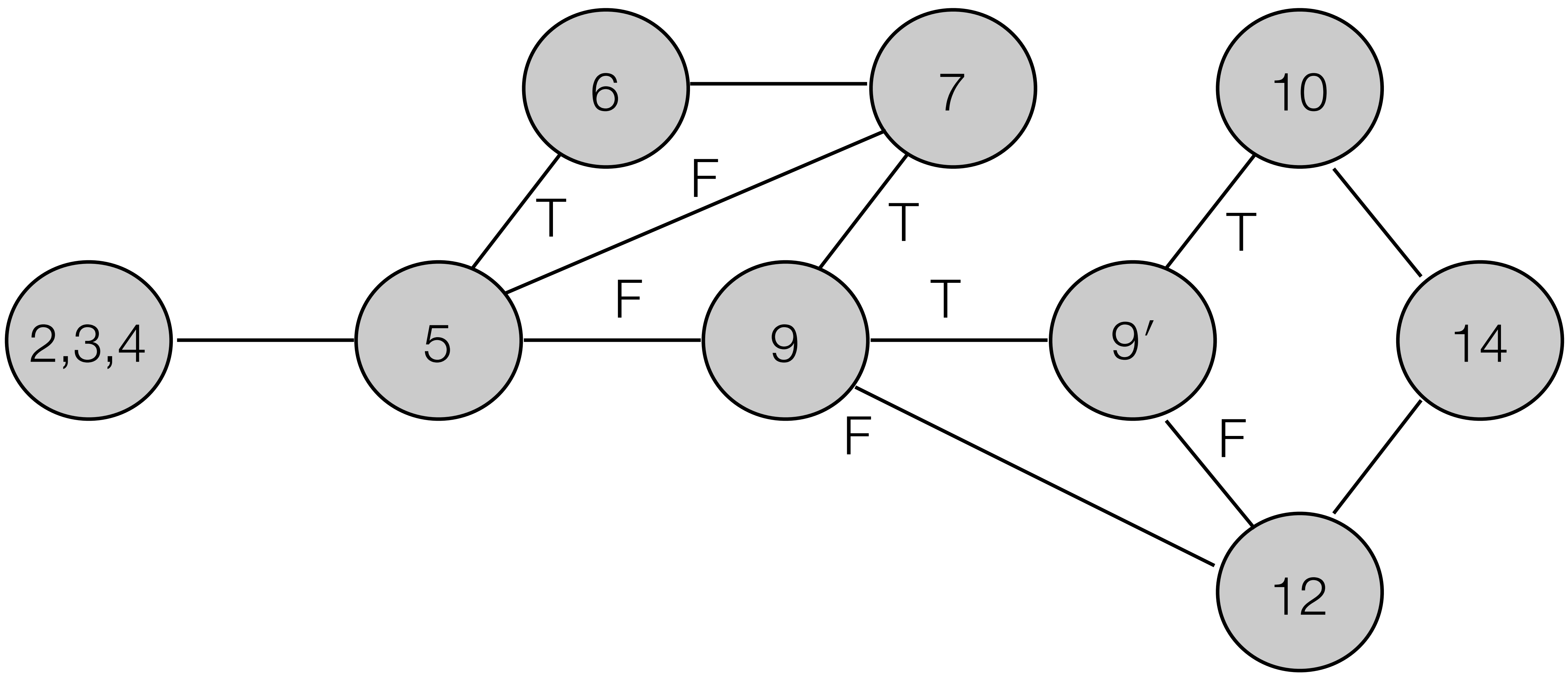
Code Coverage (II)

- 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)

A Sample Ada Program to Test

```
function P return INTEGER is
begin
   X, Y: INTEGER;
   READ(X); READ(Y);
   while (X > 10) loop
      X := X – 10;
      exit when X = 10;
   end loop;
   if (Y < 20 and then X mod 2 = 0) then
      Y := Y + 20;
   else
      Y := Y – 20;
   end if;
   return 2 ∗ X + Y;
end P;
```

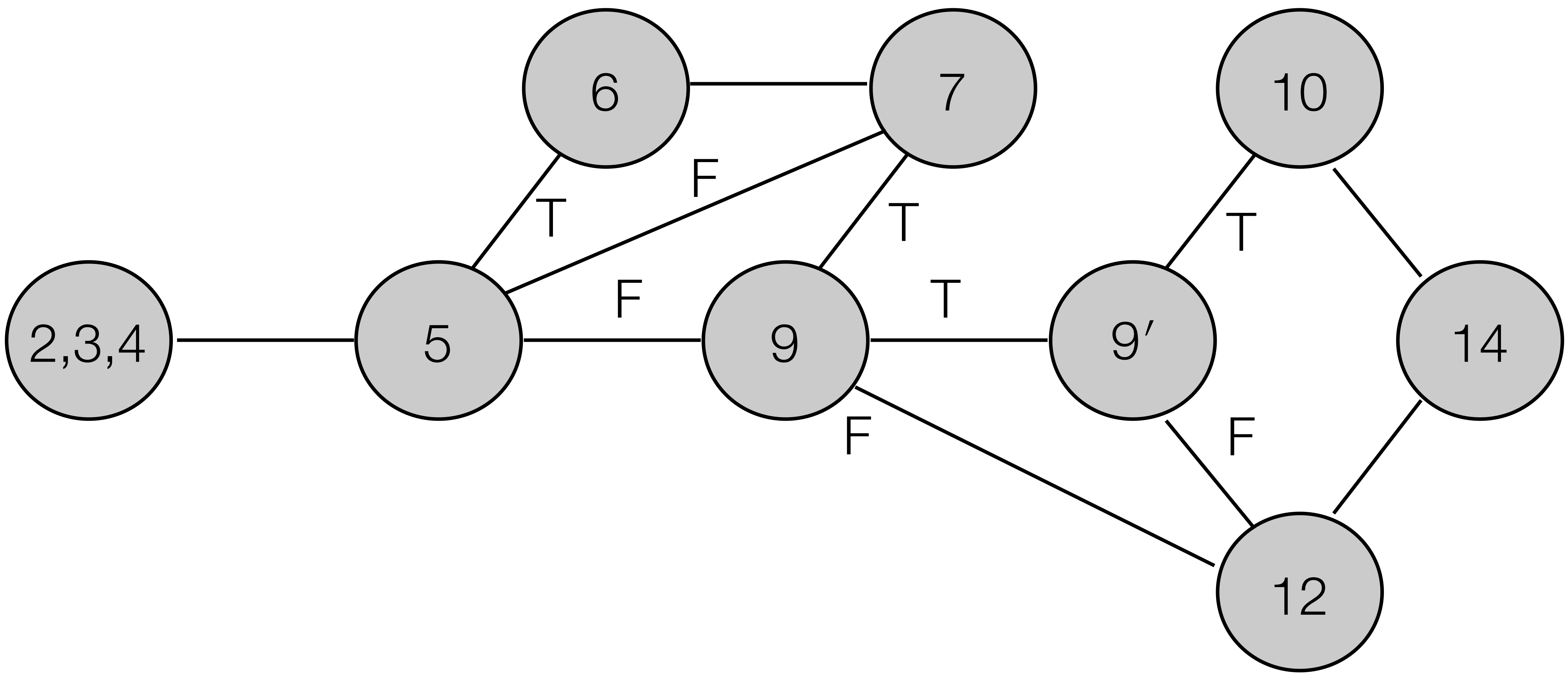
P’s Control Flow Graph (CFG)



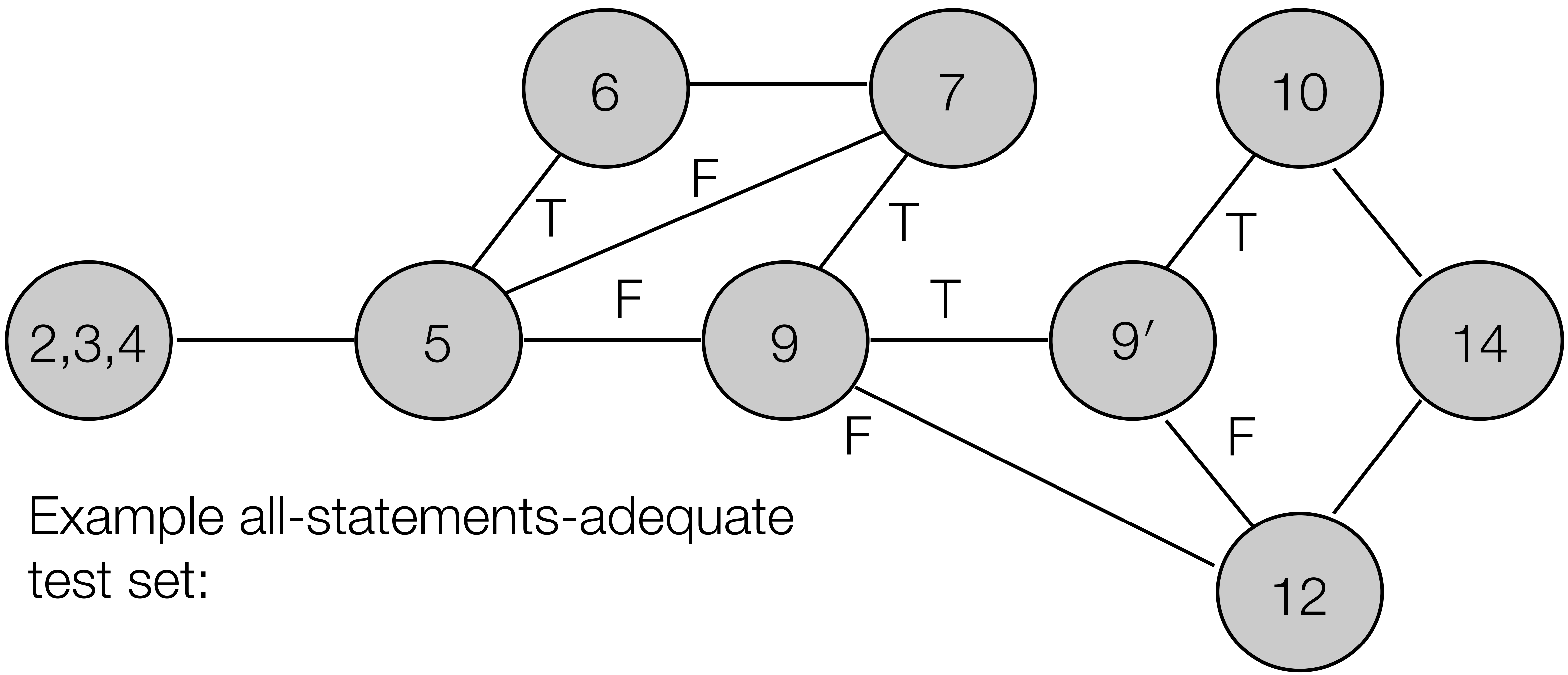
White-box Testing Criteria

- 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

All-Statements Coverage of P

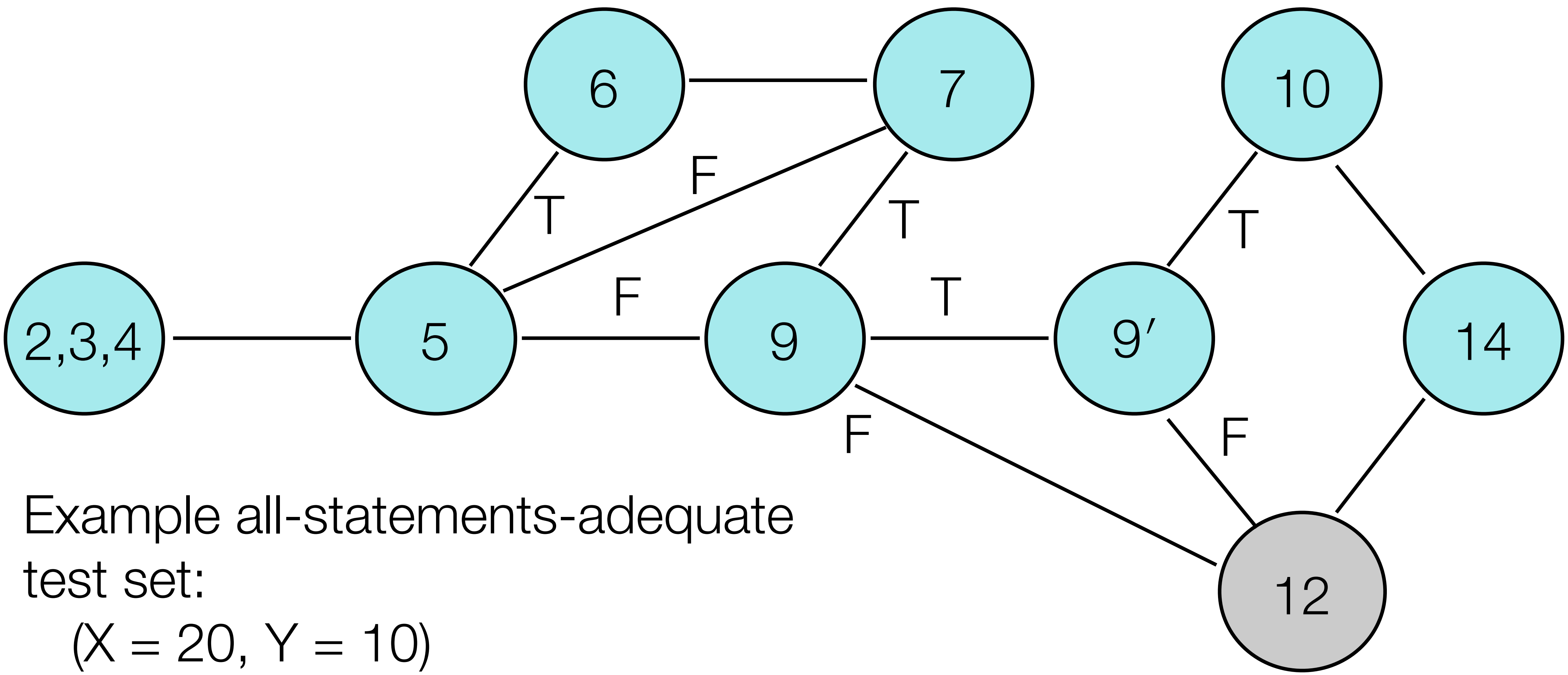


All-Statements Coverage of P



Example all-statements-adequate test set:

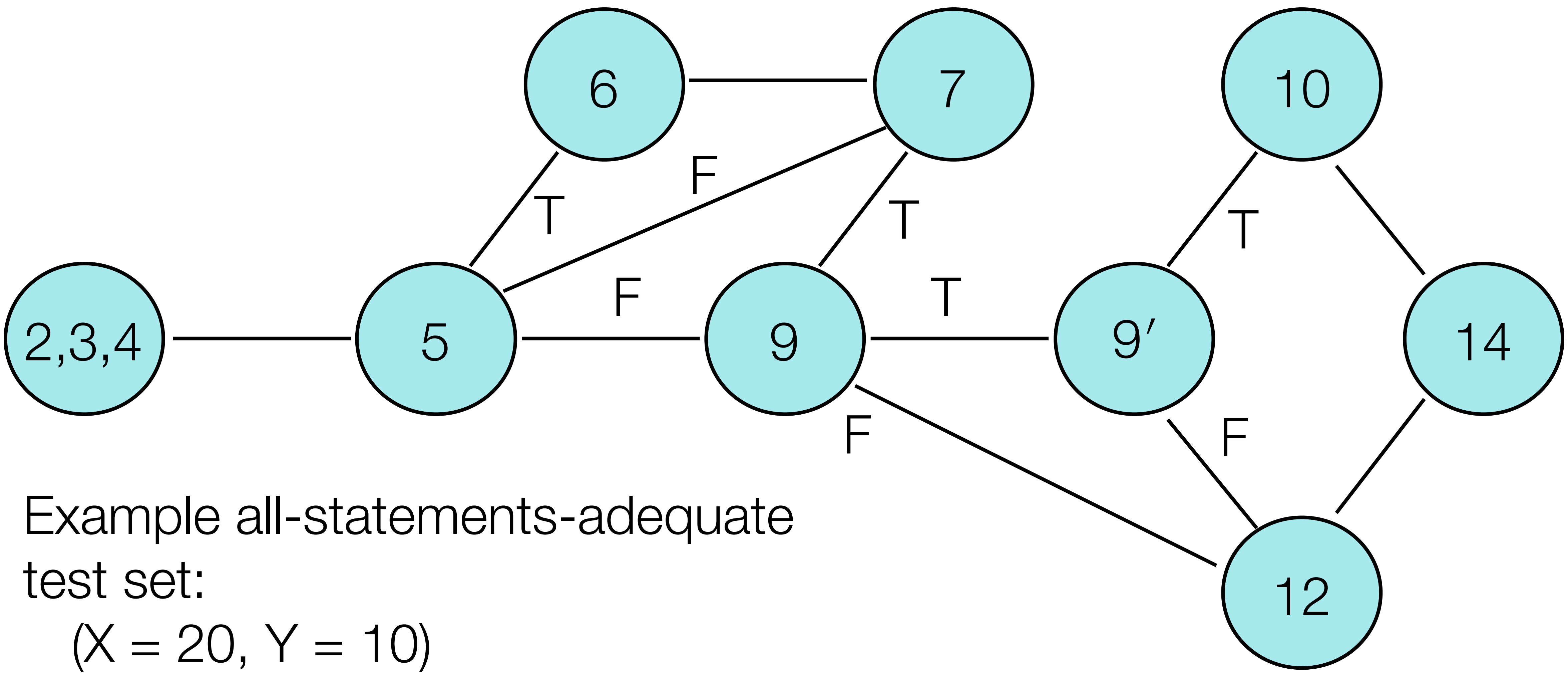
All-Statements Coverage of P



Example all-statements-adequate test set:

(X = 20, Y = 10)

All-Statements Coverage of P



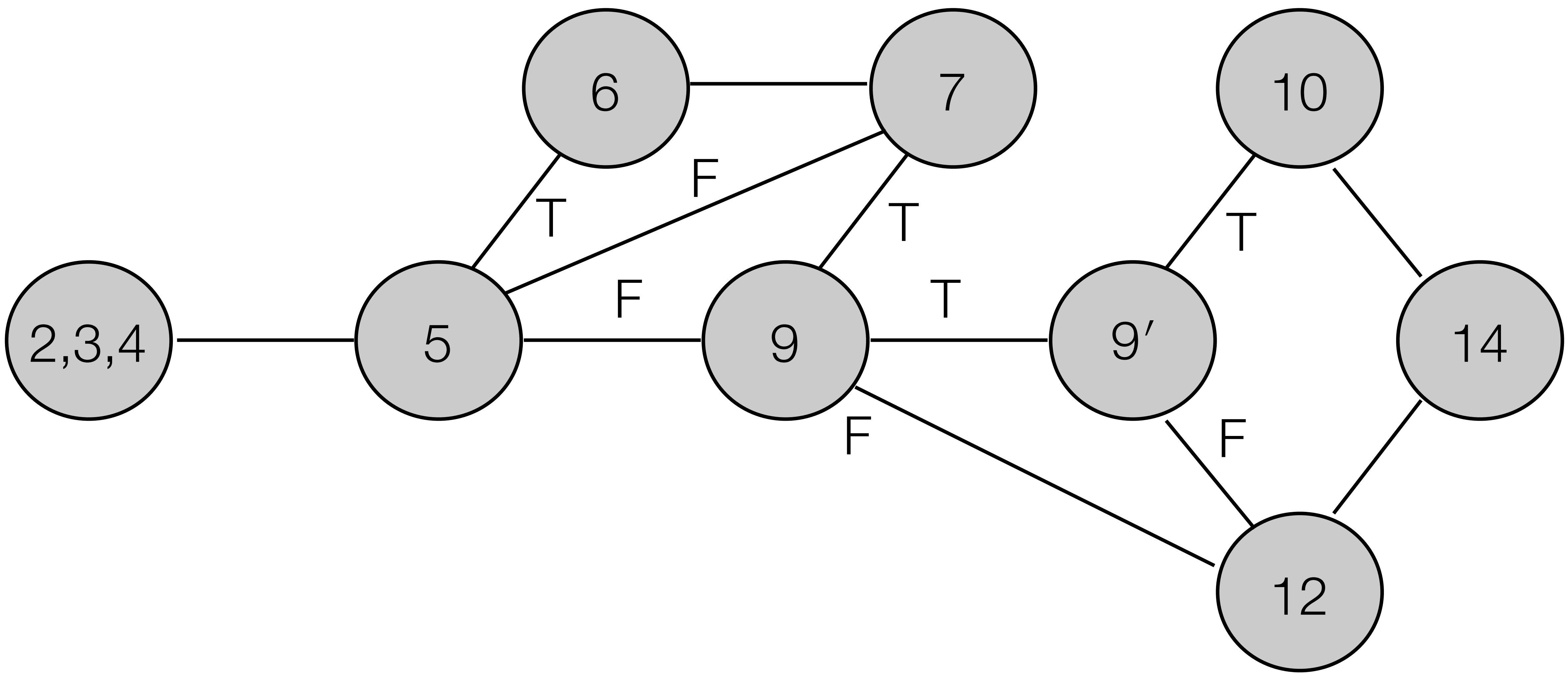
Example all-statements-adequate test set:

(X = 20, Y = 10)
(X = 20, Y = 30)

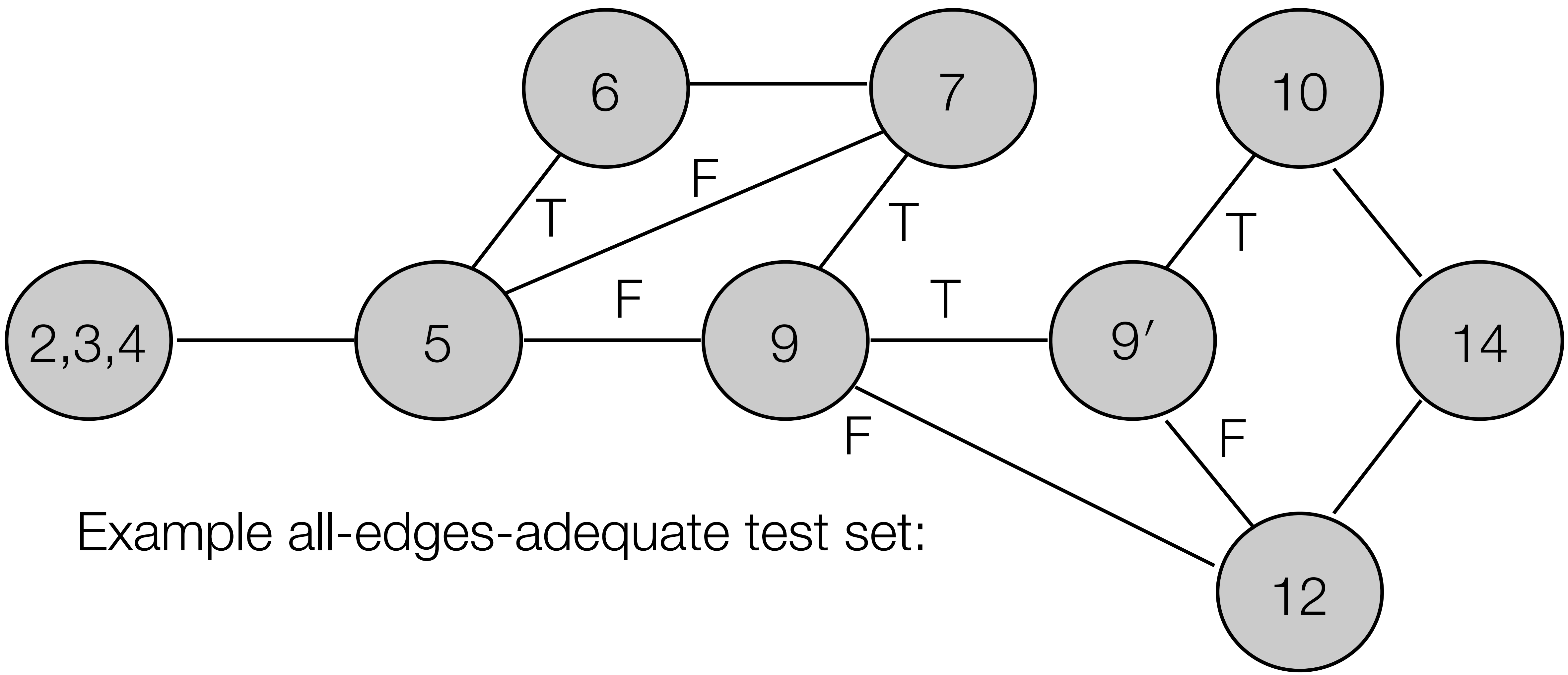
White-box Testing Criteria

- 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

All-Edges Coverage of P

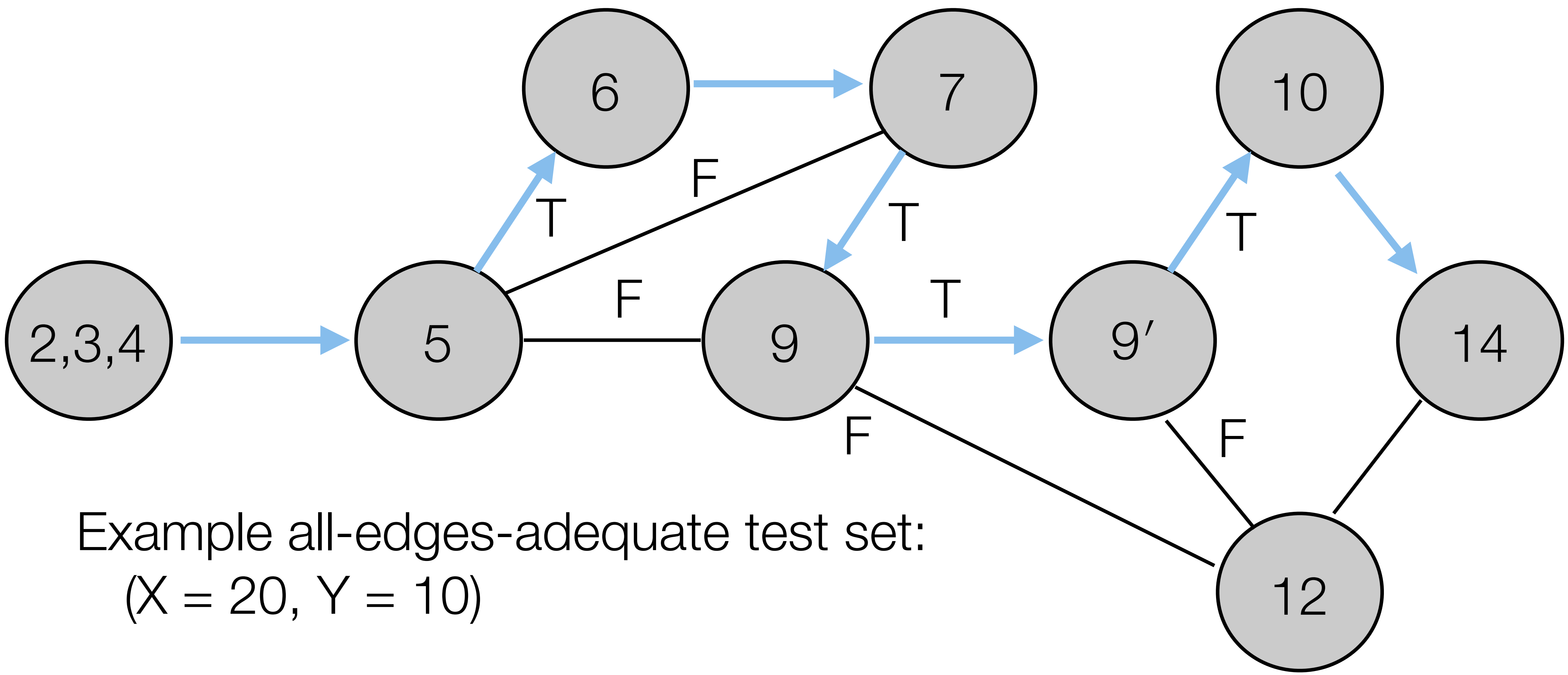


All-Edges Coverage of P



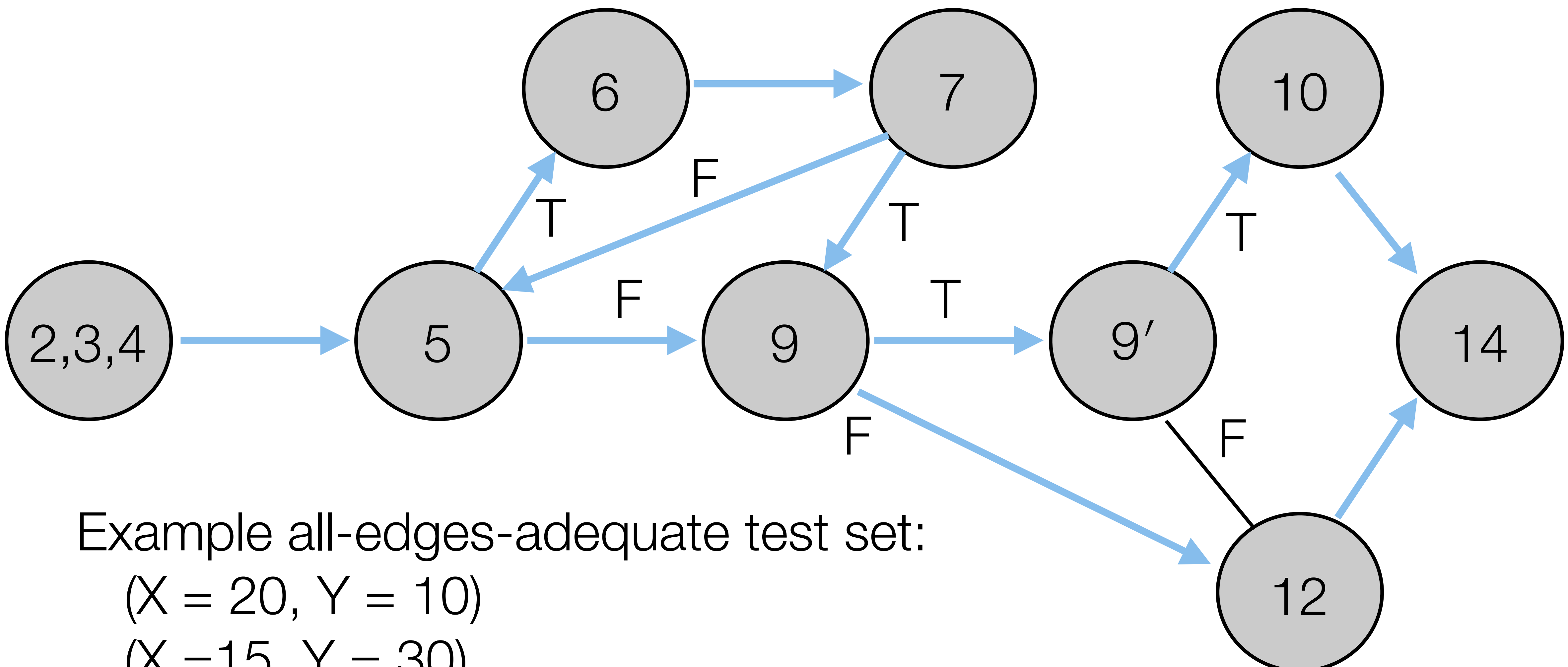
Example all-edges-adequate test set:

All-Edges Coverage of P



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

All-Edges Coverage of P



Example all-edges-adequate test set:
(X = 20, Y = 10)
(X =15, Y = 30)

What is Good Software?

- “Good” is often associated with some definition of quality. The higher the quality, the better the software.
- The problem? Many different definitions of quality!
  - **Transcendental**: where quality is something we can recognize but not define (“I know it when I see it”)
  - **User**: where quality is determined by evaluating the fitness of a system for a particular purpose or task (or set of tasks)
  - **Manufacturing**: quality is conformance to a specification
  - **Product**: quality is determined by internal characteristics (e.g. number of bugs, complexity of modules, etc.)
  - **Value**: quality depends on the amount customers are willing to pay
    - customers adopt “user view”; developers adopt “manufacturing view”, researchers adopt “product view”; “value view” can help to tie these together

What is Good Software?

- Good software engineering must always include a **strategy for producing high quality software**
- Three common ways that SE considers quality:
  - The quality of the product (product view)
  - The quality of the process (manufacturing view)
  - The quality of the product in the context of a business environment (user view)
- The results of the first two are termed the “technical value of a system”; The latter is the “business value of a system”

The Quality of the Product

- Users judge a system on external characteristics
    - correct functionality, number of failures, types of failures
- Developers judge the system on internal characteristics
    - types of faults, reliability, efficiency, etc.
- Quality models can be used to relate these two views
    - An example is McCall's quality model
        - This model can be useful to developers: want to increase "reliability" examine your system's "consistency, accuracy, and error tolerance"

The Quality of the Process (I)

- Quality of the development and maintenance process is as important as the product quality
  - The development process needs to be modeled

The Quality of the Process (II)

- Modeling will address questions such as
    - What steps are needed and in what order?
    - Where in the process is effective for finding a particular kind of fault?
    - How can you shape the process to find faults earlier?
    - How can you shape the process to build fault tolerance into a system?

The Quality of the Process (III)

- Models for Process Improvement
  - SEI's Capability Maturity Model (CMM)
  - ISO 9000
  - Software Process Improvement and Capability dEtermination (SPICE)

Business Environment Quality (I)

- The business value being generated by the software system
  - Is it helping the business do things faster or with less people?
    - Does it increase productivity?
- To be useful, the business value must be quantified

Business Environment Quality (II)

- A common approach is to use “return on investment” (ROI)
- Problem: Different stakeholders define ROI in different ways!
  - Business schools: “what is given up for other purposes”
  - U.S. Government: “in terms of dollars, reducing costs, predicting savings”
  - U.S. Industry: “in terms of effort rather than cost or dollars; saving time, using fewer people”
- Differences in definition means that one organization’s ROI can NOT be compared with another organization’s ROI without careful analysis

Software Engineering: More than just Programming

- It should now be clear that software engineering is more than just
  - programming, data structures, algorithms, etc.
- It takes advantage of these very useful computer science techniques but adds
  - quality concerns
    - testing, code reviews, validation and verification of requirements
  - process concerns
    - Are we using the right software life cycle? Are we monitoring our ability to execute the process? Are we consistent? Are we getting better?
  - reliance on tools, people, and support processes
    - debugging, profiling, configuration management, deployment, issue tracking

Summary

- In this lecture, we discussed
  - Brooks's definition of a silver bullet
    - A single tool or technique that by itself produces an order of magnitude improvement in the production of software
  - and his argument for why there is no silver bullet for software engineering
- We continued our introduction to the field of software engineering
  - Additional definitions and concerns
  - Challenges faced by the field
  - The importance of quality assurance and why it is difficult to define "quality" for software engineering

SE Conferences

- International Conference on Software Engineering (ICSE)
  - http://www.icse-conferences.org/
- International Symposium on the Foundations of Software Engineering (FSE)
- Automated Software Engineering
- Many, many more; See for instance
  - http://www.sigsoft.org/conferences/listOfEvents.htm

Professional Societies

- For Computer Science in general
  - ACM: Association for Computing Machinery
    - http://www.acm.org/
  - IEEE Computer Society
    - http://www.computer.org/
- For Software Engineering
  - ACM Special Interest Group on Software Engineering (ACM SIGSOFT)
    - http://www.sigsoft.org/

SE Journals

- The Big Two
  - ACM Transactions on Software Engineering and Methodology
    - http://tosem.acm.org/
  - IEEE Transactions on Software Engineering
    - <http://www.computer.org/portal/web/tse>
- Papers are also available at ACM’s and IEEE’s digital libraries
  - ACM Digital Library: http://dl.acm.org/
  - IEEE Digital Library: http://www.computer.org/portal/web/csdl

SE-Related Sites/Blogs

- A great combination: **a good developer with a blog**
  - loudthinking.com; inessential.com; http://daringfireball.net/
  - http://joelonsoftware.com; http://ridiculousfish.com/blog/posts.html
  - http://www.tbray.org/ongoing/; scripting.com; http://blog.wilshipley.com/
  - http://jeff-vogel.blogspot.com/; http://notch.tumblr.com/
- More general: slashdot.org; stackoverflow.com; semat.org; Hacker News
- **Humor**:
  - xkcd.org, The Order of the Stick, thedailywtf.com
- Please send me others that you find useful