测试理论
Testing Theory

By, Fahad Sabah

TFSE_Fall2025_北京工业大学，中国

TFSE_Fall2025_Beijing University of Technology, China

CONTENTS

1-1 基于图的测试
1-1 Graph-Based Testing

A graphical representation of all paths that might be traversed through a program during its execution.

Core Concept:

Model the software as a graph:

- ❑ *Nodes represent statements or code blocks*
  - ❑ *Decision nodes (diamonds) represent conditions (if, for, while)*
  - ❑ *Statement nodes (rectangles) represent operations*
- ❑ *Edges represent flow of control between nodes*

在程序执行过程中可能穿越的所有路径的图形表示。

核心概念：

将软件建模为图表：

- ❑ 节点表示语句或代码块
  - ❑ 决策节点（菱形）表示条件（if，for， while）
  - ❑ 语句节点（矩形）表示作
- ❑ 边表示节点间的控制流动

Key Coverage Types [主要覆盖类型]:

Node Coverage (Statement Coverage):

❑ Goal: Execute every node (statement) in the graph at least once.

❑ Weakness: It can easily miss bugs in decision logic. For example, an if condition might be tested as true but never as false.

节点覆盖率（语句覆盖率）：

❑ 目标：至少执行图中的每个节点（语句）一次。

❑ 缺点：它很容易漏掉决策逻辑中的漏洞。例如，一个if条件可能被测试为真，但从未被测试为假。

Key Coverage Types [主要覆盖类型]: (contd…)

Edge Coverage (Branch Coverage):

- ❑Goal: Traverse every edge in the graph at least once. This means taking both the true and false branches of every decision.
- ❑This is significantly stronger than node coverage and is a common requirement in safety-critical systems.

边缘覆盖（分支覆盖）：

- ❑ 目标：至少遍历图表中的每条边一次。这意味着每个决策都要同时选择真分支和假分支。
- ❑ 这远强于节点覆盖，是安全关键系统中常见的需求。

Key Coverage Types [主要覆盖类型]: (contd…)

Path Coverage:

- ❑Goal: Execute every possible path from the start node to the end node.
- ❑Problem: In any program with loops, the number of paths can be infinite. This makes full path coverage theoretically impossible in most cases.
- ❑Solution in Practice: We use weaker criteria like:
- ❑Linearly Independent Path Coverage (Basis Path): A manageable set of paths from which all other paths can be derived. This is the basis of McCabe's Cyclomatic Complexity.

路径覆盖:

- ❑ 目标：执行从起始节点到终点节点的所有可能路径。
- ❑ 问题：在任何有循环的程序中，路径数量可以是无限的。这使得在大多数情况下理论上不可能实现全路径覆盖。
- ❑ 实践中的解决方案：我们使用较弱的标准，如：
- ❑ 线性独立路径覆盖率（基准路径）：一组可管理的路径，所有其他路径均可由此推导。这就是麦凯布环状复杂性理论的基础。

Example [例]:

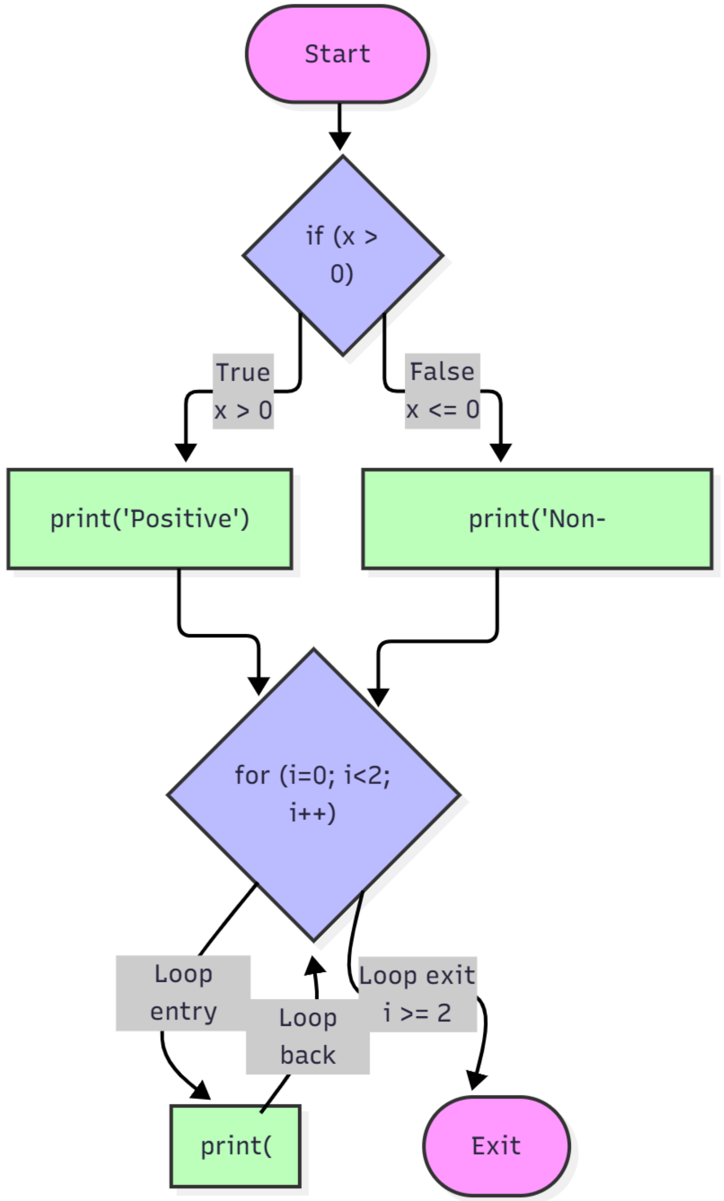
Graph:

Node 1 (if)

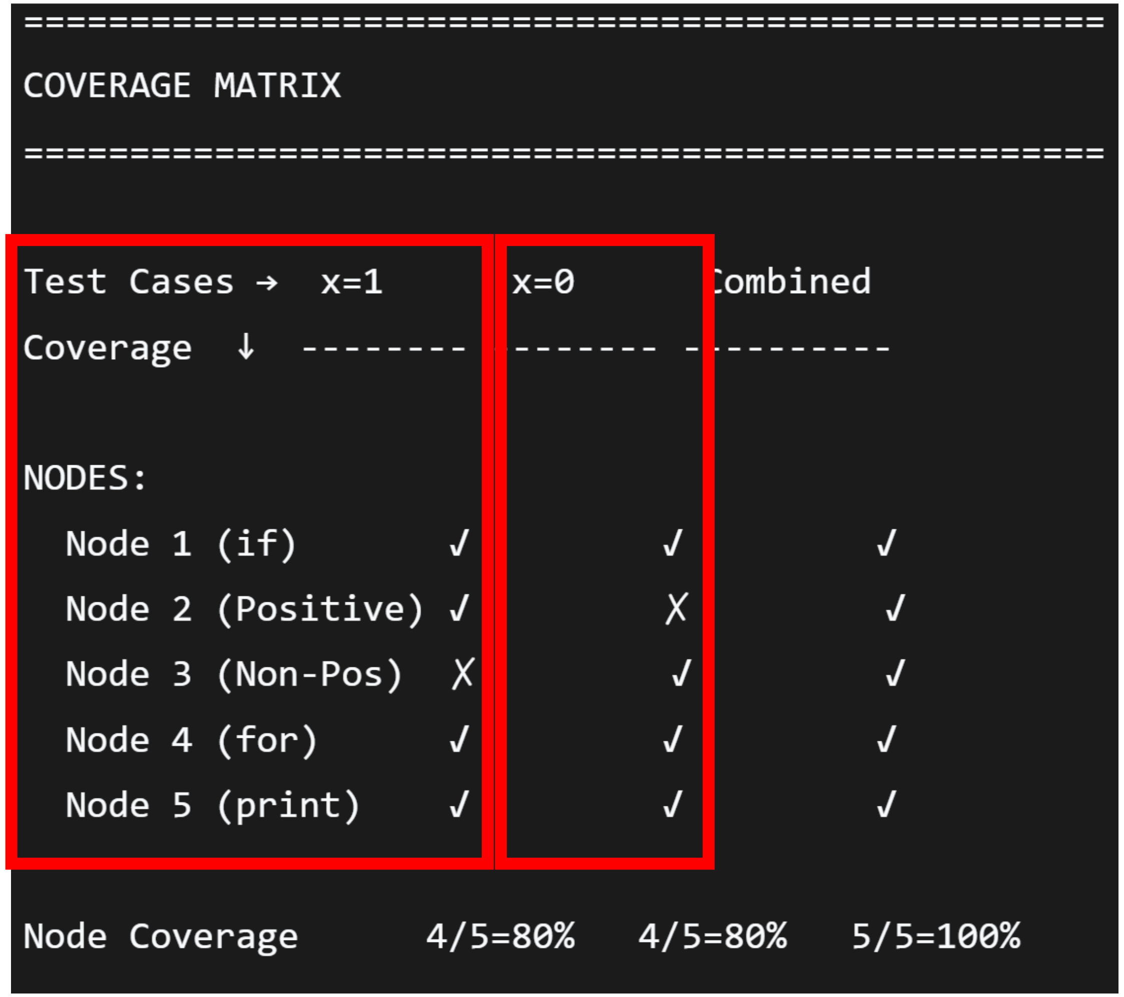
-> Edge(True) to Node 2, or Edge(False) to Node 3

-> Node 4 (for) -> (Loop back)

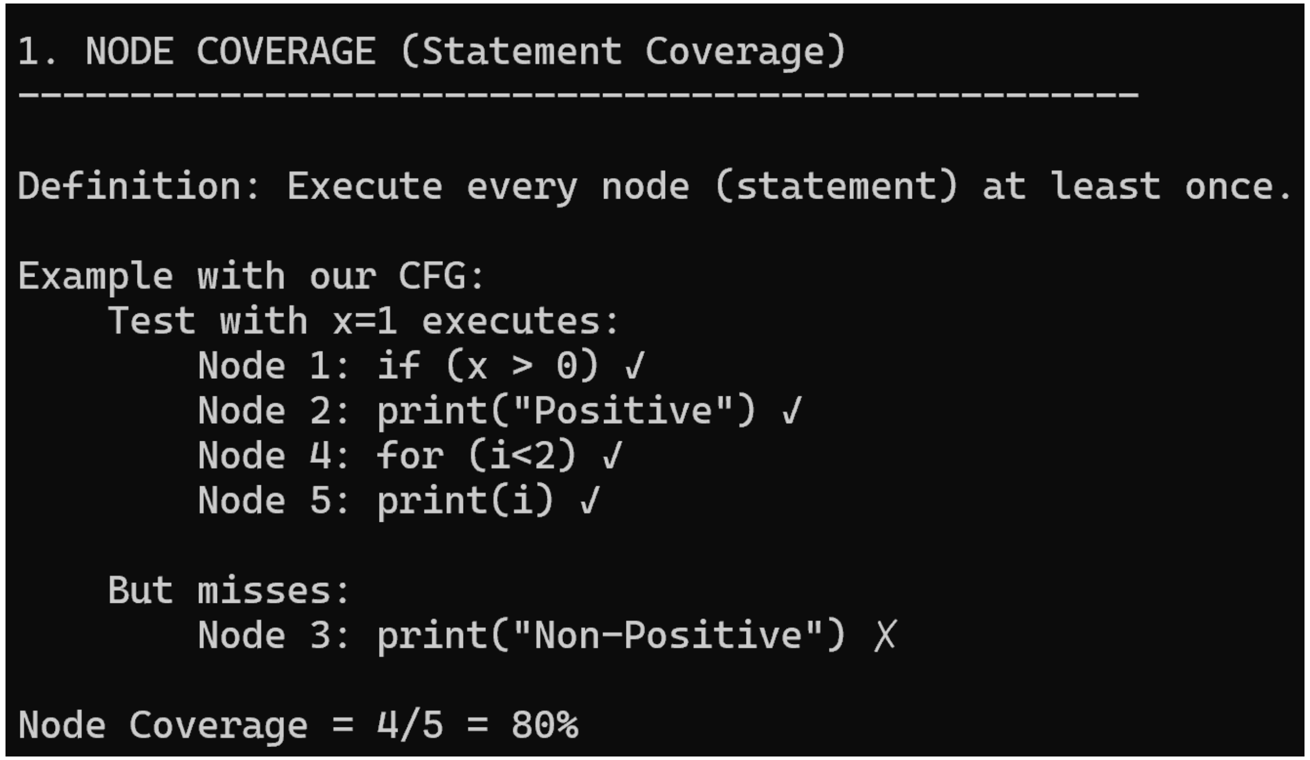
-> Node 5 -> Exit.



Node Coverage [节点覆盖率]:

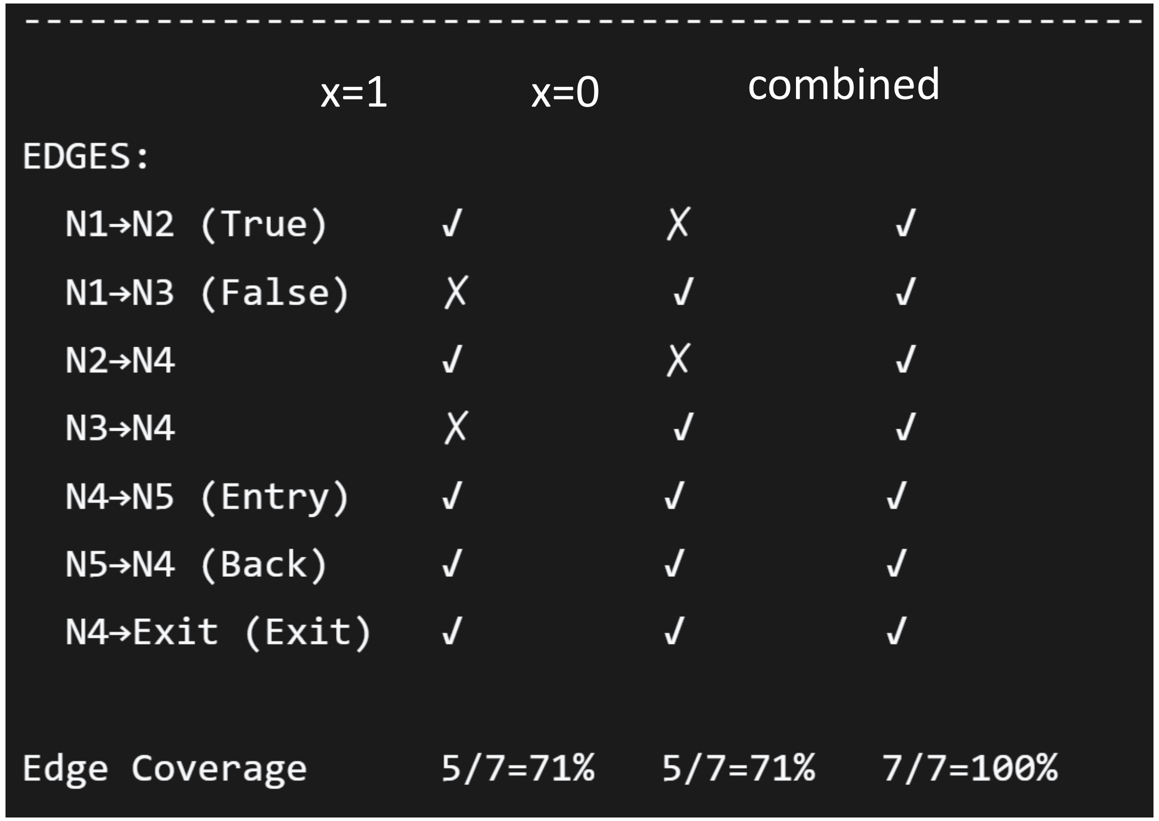


```
==================================================
COVERAGE MATRIX
==================================================

Test Cases →   x=1        x=0        Combined
Coverage  ↓  --------   --------   ----------

NODES:
  Node 1 (if)         ✓          ✓          ✓
  Node 2 (Positive)   ✓          ✗          ✓
  Node 3 (Non-Pos)    ✗          ✓          ✓
  Node 4 (for)        ✓          ✓          ✓
  Node 5 (print)      ✓          ✓          ✓

Node Coverage       4/5=80%    4/5=80%    5/5=100%
```

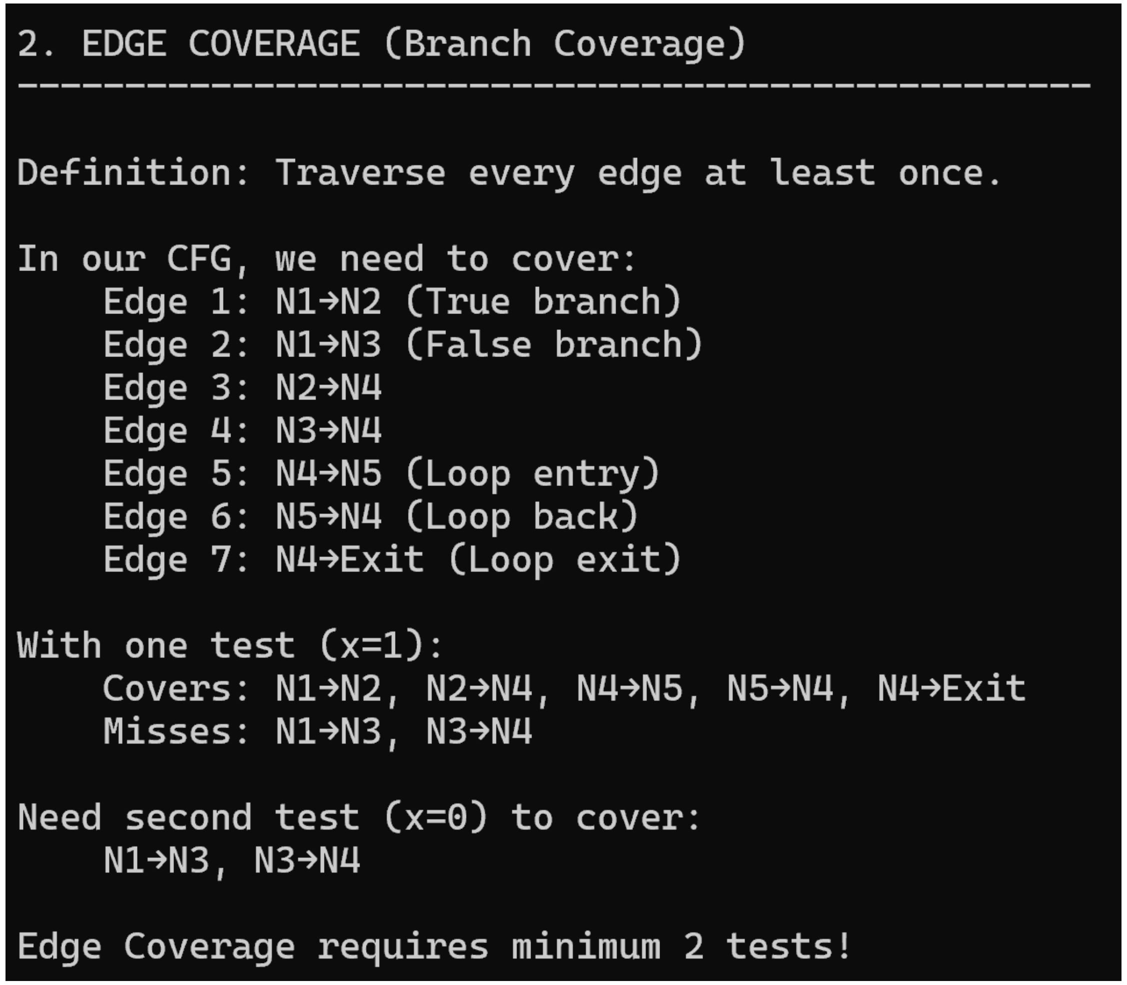


```
1. NODE COVERAGE (Statement Coverage)
---------------------------------------------------

Definition: Execute every node (statement) at least once.

Example with our CFG:
    Test with x=1 executes:
        Node 1: if (x > 0) ✓
        Node 2: print("Positive") ✓
        Node 4: for (i<2) ✓
        Node 5: print(i) ✓

    But misses:
        Node 3: print("Non-Positive") ✗

Node Coverage = 4/5 = 80%
```

Edge Coverage: Requires two tests.
边缘覆盖：需要两次测试。

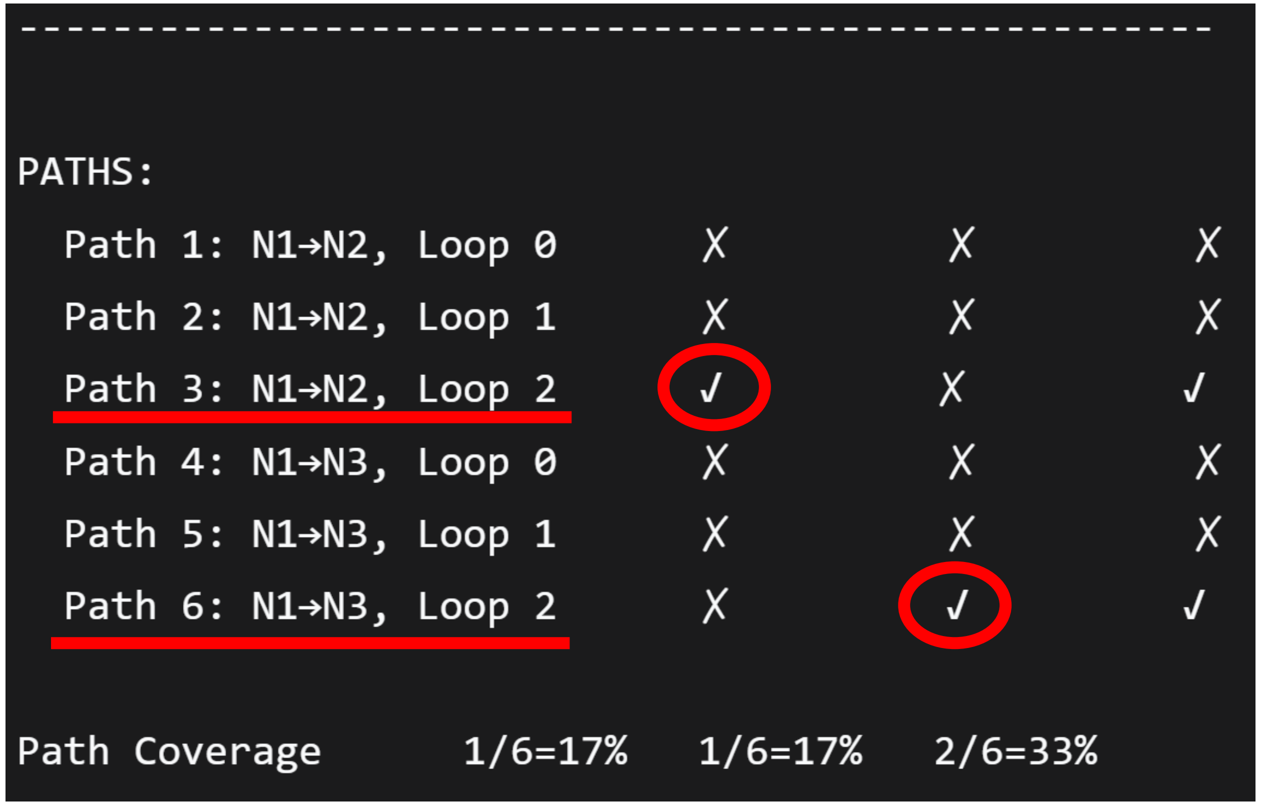


EDGES:	x=1	x=0	combined
N1→N2 (True)	✓	✗	✓
N1→N3 (False)	✗	✓	✓
N2→N4	✓	✗	✓
N3→N4	✗	✓	✓
N4→N5 (Entry)	✓	✓	✓
N5→N4 (Back)	✓	✓	✓
N4→Exit (Exit)	✓	✓	✓
Edge Coverage	5/7=71%	5/7=71%	7/7=100%

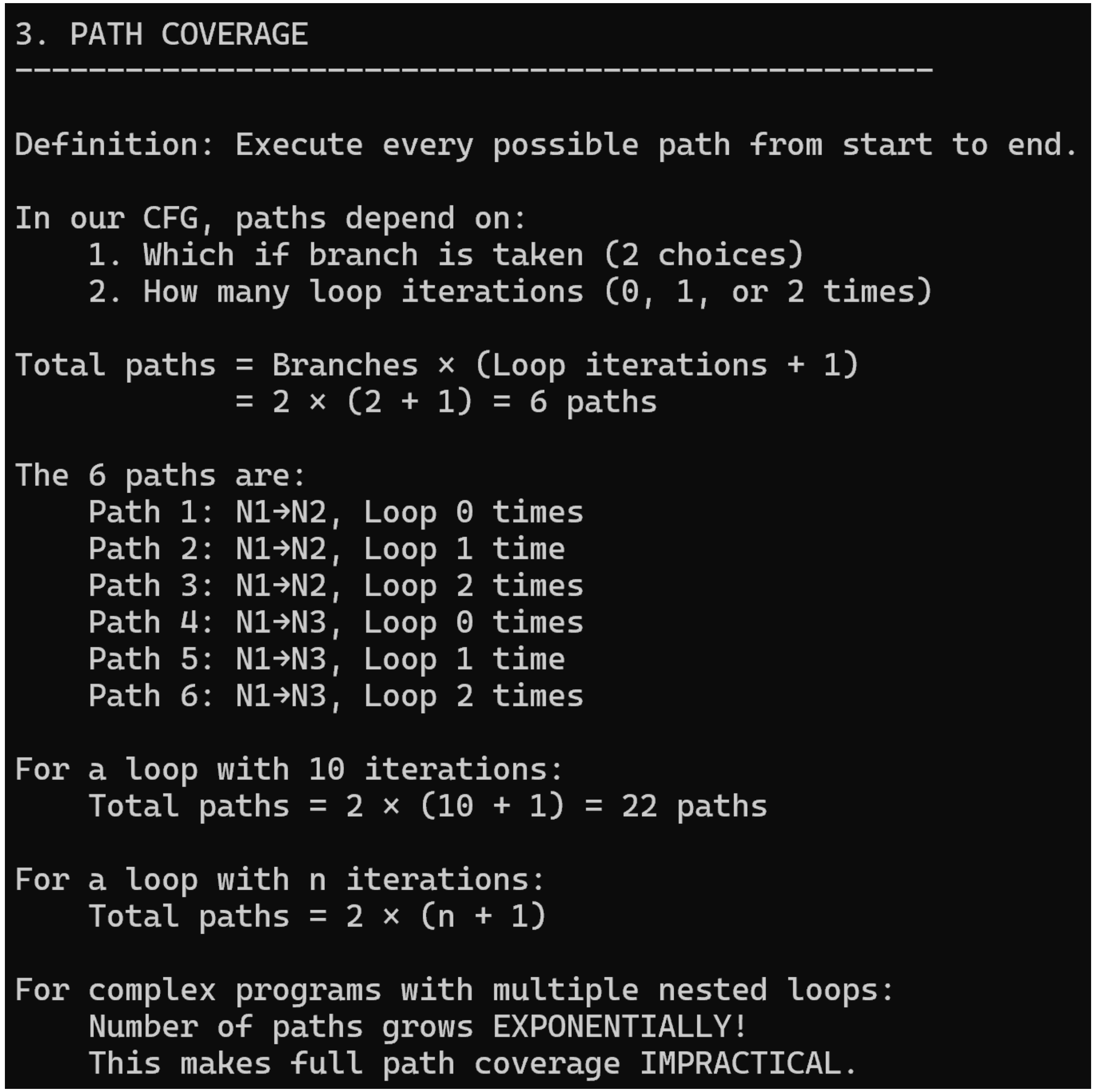


```
2. EDGE COVERAGE (Branch Coverage)
--------------------------------------------------

Definition: Traverse every edge at least once.

In our CFG, we need to cover:
    Edge 1: N1→N2 (True branch)
    Edge 2: N1→N3 (False branch)
    Edge 3: N2→N4
    Edge 4: N3→N4
    Edge 5: N4→N5 (Loop entry)
    Edge 6: N5→N4 (Loop back)
    Edge 7: N4→Exit (Loop exit)

With one test (x=1):
    Covers: N1→N2, N2→N4, N4→N5, N5→N4, N4→Exit
    Misses: N1→N3, N3→N4

Need second test (x=0) to cover:
    N1→N3, N3→N4

Edge Coverage requires minimum 2 tests!
```

Path Coverage:



```
----------------------------------------------------


PATHS:
  Path 1: N1→N2, Loop 0     ✗          ✗          ✗
  Path 2: N1→N2, Loop 1     ✗          ✗          ✗
  Path 3: N1→N2, Loop 2     ✓          ✗          ✓
  Path 4: N1→N3, Loop 0     ✗          ✗          ✗
  Path 5: N1→N3, Loop 1     ✗          ✗          ✗
  Path 6: N1→N3, Loop 2     ✗          ✓          ✓


Path Coverage        1/6=17%    1/6=17%    2/6=33%
```



```
3. PATH COVERAGE
--------------------------------------------------

Definition: Execute every possible path from start to end.

In our CFG, paths depend on:
    1. Which if branch is taken (2 choices)
    2. How many loop iterations (0, 1, or 2 times)

Total paths = Branches × (Loop iterations + 1)
            = 2 × (2 + 1) = 6 paths

The 6 paths are:
    Path 1: N1→N2, Loop 0 times
    Path 2: N1→N2, Loop 1 time
    Path 3: N1→N2, Loop 2 times
    Path 4: N1→N3, Loop 0 times
    Path 5: N1→N3, Loop 1 time
    Path 6: N1→N3, Loop 2 times

For a loop with 10 iterations:
    Total paths = 2 × (10 + 1) = 22 paths

For a loop with n iterations:
    Total paths = 2 × (n + 1)

For complex programs with multiple nested loops:
    Number of paths grows EXPONENTIALLY!
    This makes full path coverage IMPRACTICAL.
```

1-2. Mutation Testing [突变测试]

This is a powerful technique to evaluate the effectiveness of the test suite, not to design tests themselves. It answers the question: "How good are my tests at catching bugs?"

Core Concept:

- ❑**Create Mutants:** Automated tools (e.g., PITest for Java, MutPy for Python) make small, syntactic changes to your source code, creating faulty versions called "mutants."
- ❑**Examples:** Change > to >=, + to -, && to ||, delete a statement, change a constant value.
- ❑**Run Test Suite:** Your entire test suite is run against each mutant.

1-2. Mutation Testing [突变测试]

Analyze Results:

- **Killed:** If a test fails, the mutant is "killed." This is good! It means your tests detected the fault.
- **Survived:** If all tests pass, the mutant has "survived." This is bad. It means your test suite could not distinguish the mutant from the original program, revealing a deficiency in your tests.
- **The Mutation Score:** This is the key metric.

Mutation Score = (Number of Killed Mutants / Total Number of Mutants) * 100%

- The goal is to have a mutation score as close to 100% as possible.

Why it's powerful [它的强大原因]:

- ❑It forces you to write tests that are sensitive to the code's logic, not just its output in a few happy paths. It's one of the strongest criteria for test suite adequacy.

Challenges:

- ❑Computationally Expensive: Generating and running tests against thousands of mutants is slow.
- ❑Equivalent Mutants: Some mutants are semantically identical to the original code (e.g., changing i++ to ++i in an isolated statement). These can never be killed and "pollute" the score.

2. 理论极限：停机问题及其影响
2. Theoretical Limits: The Halting Problem and Its Implications

What is the Halting Problem?

Proven by Alan Turing, the Halting Problem states that it is impossible to create a general-purpose program (an algorithm) that can take any arbitrary program and its input and decide whether the program will eventually halt or run forever.

什么是停机问题?

由艾伦·图灵证明，停机问题指出，不可能创建一个通用程序（算法），能够对任意程序及其输入决定程序最终是停止还是永远运行。

Implications for Testing[测试启示]:

No Perfect Bug Detector:

- Claim: "I will build a tool that analyzes any program and finds all bugs."
- Refutation: One specific type of bug is an infinite loop. If you could build a perfect bug detector, you could use it to solve the Halting Problem (by checking for "will this loop halt?"). Since the Halting Problem is unsolvable, a perfect bug detector cannot exist.

No Perfect Test Case Generator:

- Claim: "I will build a tool that generates a set of test cases which, if they all pass, prove the program is correct."
- Refutation: This is the dream of "Complete Verification." For this to work, the tool would need to know everything about the program's behavior, including whether it halts on all inputs. Again, this reduces to the Halting Problem.

Undecidability of "All Programs Reach This Statement":

- Even a seemingly simple goal like "Does a test exist that will execute this particular line of code?" is, in general, undecidable. The tool cannot know if the conditions to reach that line are even possible to satisfy.

Summary [总结]

Concept	Purpose	Key Idea
Graph-Based Testing	Designing Test Cases	Model code as a graph and cover its nodes, edges, or paths.
Mutation Testing	Evaluating Test Suite Quality	Create small faults in code and check if tests can find them.
Theoretical Limits	Understanding What's Possible	The Halting Problem proves that perfect, automated testing is impossible.

Together, these concepts form a crucial part of testing theory: we have powerful methods (Graph-Based) to create tests, a powerful metric (Mutation) to judge them, and a fundamental theory to keep our expectations in check.

使用图书馆管理系统示例的高级软件测试技术
Software Testing Techniques Using an LMS Example

System Requirements:

- ❑Users can borrow books (if available)
- ❑Users can return books
- ❑Users can search for books by title/author
- ❑Users have a maximum borrowing limit (3 books)
- ❑Books become overdue after 14 days
- ❑System tracks fines (0.50 CNY per day overdue)

基于图的LMS测试示例
Graph-Based Testing with LMS Example

- Creating Control Flow Graphs (CFGs) from LMS Code
- Example 1: borrowBook Method

```
public BorrowResult borrowBook(String userId, String bookId) {
    // Node 1: Start
    if (!users.containsKey(userId)) { // Decision 1
        return new BorrowResult("User not found"); // Node 2
    }
    if (!books.containsKey(bookId)) { // Decision 2
        return new BorrowResult("Book not found"); // Node 3
    }
    User user = users.get(userId);
    Book book = books.get(bookId);
    if (!book.isAvailable) { // Decision 3
        return new BorrowResult("Book already borrowed"); // Node 4
    }
    if (user.borrowedBooks.size() >= 3) { // Decision 4
        return new BorrowResult("Borrowing limit reached"); // Node 5
    }
    if (user.fineAmount > 0) { // Decision 5
        return new BorrowResult("Outstanding fines must be paid"); // Node 6
    }
    // Node 7: Process borrowing
    book.isAvailable = false;
    book.dueDate = LocalDate.now().plusDays(14);
    user.borrowedBooks.add(bookId);
    return new BorrowResult("Success", book.dueDate); // Node 8
```

基于图的LMS测试示例
Graph-Based Testing with LMS Example

```
[Node 1: Start]
      [if(user exists?)] -----False-----> [Node 2: Return error]
           | True
           v
      [if(book exists?)] -----False-----> [Node 3: Return error]
           | True
           v
      [if(book available?)] --False---> [Node 4: Return error]
           | True
           v
      [if(user has <3 books?)] -False-> [Node 5: Return error]
           | True
           v
      [if(fineAmount == 0?)] ---False---> [Node 6: Return error]
           | True
           v
      [Node 7: Process borrowing] --->[Node 8: Return success]
```

带测试用例的覆盖标准
Coverage Criteria with Test Cases

Test Case Design Table:

Test ID	Input (userId, bookId)	Expected Output	Coverage
T1	("U999", "B001")	"User not found"	Node 2, Edge 1→2
T2	("U001", "B999")	"Book not found"	Node 3, Edge 1→3
T3	("U001", "B002") where B002 is borrowed	"Book already borrowed"	Node 4, Edge 1→4
T4	("U002", "B001") where U002 has 3 books	"Borrowing limit reached"	Node 5, Edge 1→5
T5	("U003", "B001") where U003 has fines	"Outstanding fines"	Node 6, Edge 1→6
T6	("U001", "B001") valid case	"Success"	Nodes 7,8, Edge 1→7→8

In-Class Exercise [课堂练习]:

❑What's the statement coverage with tests T1-T6?	(100%)
❑What's the branch coverage?	(100% - all edges covered)
❑How many independent paths exist?	Let's calculate...

McCabe's 的环形复杂性 McCabe's Cyclomatic Complexity

Formula: $V(G) = E - N + 2P$

Where:

- ❑E = Number of edges in graph
- ❑N = Number of nodes in graph
- ❑P = Number of connected components (usually 1)

For our borrowBook CFG:

- ❑E = 12 edges
- ❑N = 8 nodes
- ❑P = 1
- ❑$V(G) = 12 - 8 + 2 = 6$

线性独立路径 linearly independent paths

- ❑There are 6 linearly independent paths
- ❑We need at least 6 test cases to achieve basis path coverage
- ❑Our test suite (T1-T6) has exactly 6 tests!

Independent Paths:

- ❑1→2 (User doesn't exist)
- ❑1→3 (Book doesn't exist)
- ❑1→4 (Book unavailable)
- ❑1→5 (Limit reached)
- ❑1→6 (Has fines)
- ❑1→7→8 (Success path)

数据流测试示例 Data Flow Testing Example

- Consider the calculateFine method:

```
public double calculateFine(String bookId) {
    // Node A: Define book
    Book book = books.get(bookId);
    if (book == null) { // Decision 1
        return 0.0; // Node B
    }
    // Node C: Define today
    LocalDate today = LocalDate.now();
    if (book.dueDate == null || !today.isAfter(book.dueDate)) { // Decision 2
        return 0.0; // Node D
    }
// Node E: Calculate days late
    long daysLate =
ChronoUnit.DAYS.between(book.dueDate, today);
    // Node F: Calculate fine
    double fine = daysLate * 0.50;
    return fine; // Node G
}
```

确定使用对：
Define-Use Pairs:

- ❑book defined at Node A, used at Nodes C, E, F
- ❑today defined at Node C, used at Node E
- ❑daysLate defined at Node E, used at Node F
- ❑fine defined at Node F, used at Node G

Test Requirements:

- ❑Test paths that cover each define-use pair
- ❑Example: To test book definition at A and use at F, we need a path A→C→E→F→G

LMS的突变测试
Mutation Testing with LMS

Introducing Mutants to LMS Code

Original Code (simplified checkLimit method):

```
public boolean canBorrowMore(User user) {
   return user.borrowedBooks.size() < 3;
}
```

常见变异体生成：
Common Mutants Generated:

Relational Operator Replacement (ROR):

```
// Mutant 1: Changed < to <=
return user.borrowedBooks.size() <= 3;

// Mutant 2: Changed < to >
return user.borrowedBooks.size() > 3;

// Mutant 3: Changed < to >=
return user.borrowedBooks.size() >= 3;
```

常见变异体生成：
Common Mutants Generated:

Arithmetic Operator Replacement (AOR):

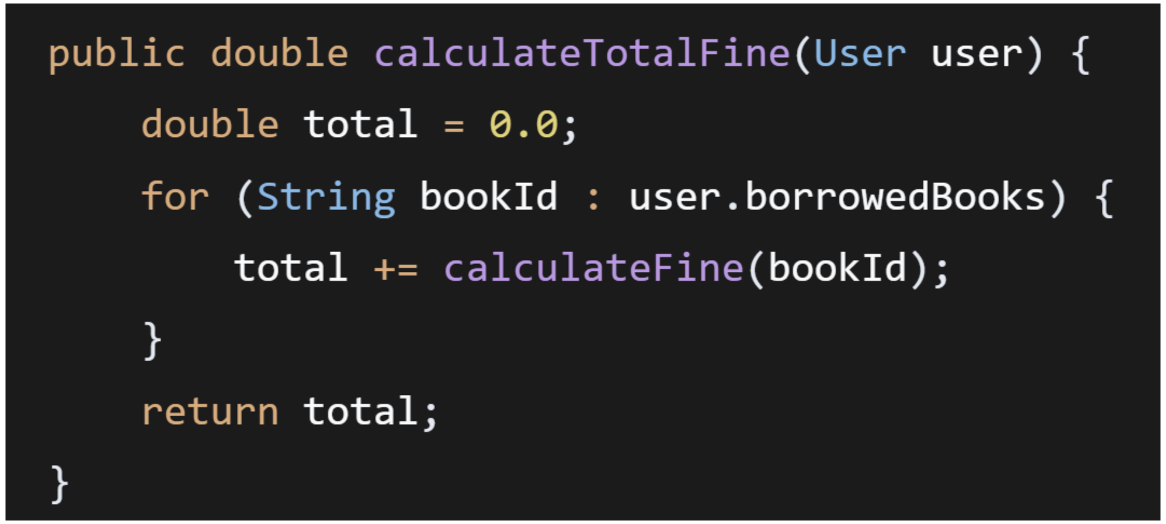
```
// Original: LocalDate dueDate = today.plusDays(14);
// Mutant: Changed plusDays to minusDays
LocalDate dueDate = today.minusDays(14);
```

Conditional Operator Replacement (COR):

```
// Original: if (user.fineAmount > 0)
// Mutant: Changed > to <
if (user.fineAmount < 0)
```

课堂活动：突变测试练习
In-Class Activity: Mutation Testing Exercise

Given this method from LMS:



```java
public double calculateTotalFine(User user) {
    double total = 0.0;
    for (String bookId : user.borrowedBooks) {
        total += calculateFine(bookId);
    }
    return total;
}
```

Questions:

- ❑ What mutants might be generated? (List at least 3)
- ❑ Design tests to kill these mutants
- ❑ Identify any potential equivalent mutants

Possible Mutants:

Change total = 0.0 to total = 1.0
Change += to -=
Change loop boundary condition
Remove the loop entirely

理论极限——对学习管理系统的实际启 Theoretical Limits - Practical Implications for LMS

- The Halting Problem in Library Context
- Consider this LMS "feature request":
- "Write a function willBookBeReturned that analyzes a user's borrowing history and predicts if they will ever return a currently borrowed book."

假设的“完美”预测方法（无法创建）Hypothetical "perfect" prediction method (IMPOSSIBLE TO CREATE)

```
public boolean willBookBeReturned(User user, Book book) {
    // This would need to solve the Halting Problem!
    // We'd need to determine if the user's "life program" halts at "return book"

    // Consider: What if the user's decision depends on:
    // - Whether they finish reading (unknowable future state)
    // - Whether they move cities
    // - Infinite loop: They keep renewing forever?
    return ???; // IMPOSSIBLE to guarantee correctness
}
```

LMS中的不可判定性质
Undecidable Properties in LMS

Property 1: "Will this search query return results in under 1 second?"

Equivalent to: "Does this program halt within X steps?"

Undecidable in general case

Property 2: "Is there any user input that will crash the system?"

This is a variant of the "will it halt?" question

We can't write a program that analyzes our LMS and guarantees it never crashes

Property 3: "Do these two library branches produce identical results?"

For complex systems, program equivalence is undecidable

Rice's Theorem Implications

Rice's Theorem: All non-trivial semantic properties of programs are undecidable.

For LMS:

"Does our overdue calculation always terminate?" - Undecidable

"Will the fine calculation ever produce a negative value?" - Undecidable

"Is there any scenario where a user can borrow infinite books?" - Undecidable

实用的变通方法
Practical Workarounds

Instead of trying to prove correctness absolutely, we:

- ❑Use testing to find bugs (not prove their absence)
- ❑Set timeouts for operations
- ❑Use assertions to catch impossible states
- ❑Implement circuit breakers for infinite loops
- ❑Accept uncertainty in predictions

Example: Realistic predictReturnProbability (not willReturn):

```
public double predictReturnProbability(User user, Book book) {
    // Heuristic approach, not perfect
    double probability = 0.9; // Base probability
    // Adjust based on observable factors
    if (user.hasHistoryOfLateReturns()) {
        probability -= 0.2;
    }
    if (book.isExpensiveTextbook()) {
        probability -= 0.1;
    }
    if (user.isFaculty()) {
        probability += 0.05;
    }
    return Math.max(0.0, Math.min(1.0, probability));
}
```

结合技术：基于图+突变测试
Combining Techniques: Graph-Based + Mutation Testing

Workflow:

- ❑Create CFG for critical LMS methods
- ❑Achieve high branch coverage with graph-based tests
- ❑Run mutation testing on the same code
- ❑Analyze surviving mutants to improve tests
- ❑Iterate until mutation score is acceptable

示例：借书集成测试套件 Example: Integrated Test Suite for borrowBook

```
// Graph-based tests (covering all branches)
@Test public void borrowBook_UserNotFound() { /* T1 */ }
@Test public void borrowBook_BookNotFound() { /* T2 */ }
@Test public void borrowBook_BookUnavailable() { /* T3 */ }
@Test public void borrowBook_LimitReached() { /* T4 */ }
@Test public void borrowBook_HasFines() { /* T5 */ }
@Test public void borrowBook_Success() { /* T6 */ }
// Mutation-killing tests (additional tests)
@Test
public void borrowBook_ExactlyThreeBooks() {
   // Kills mutant: user.borrowedBooks.size() >= 3
   // vs original: user.borrowedBooks.size() > 3
   User user = createUserWithBooks(3); // Exactly 3 books
   assertError(user, "B001", "Borrowing limit reached");
}
```

```
@Test
public void borrowBook_ZeroFinesVsPositiveFines() {
   // Kills mutant: if (user.fineAmount < 0) vs original > 0
   User user = createUser();
   user.setFineAmount(-5.0); // Negative fine (edge case)
   assertSuccess(user, "B001"); // Should work with negative
fine?
   // Actually reveals bug: Should we allow negative fines?
}
```

讨论：[简单书籍可用性查询] Discussion: [Simple Book Availability Check]

Nodes: 6 (Start, N1, N2/N3, N4, End)

Edges: 6

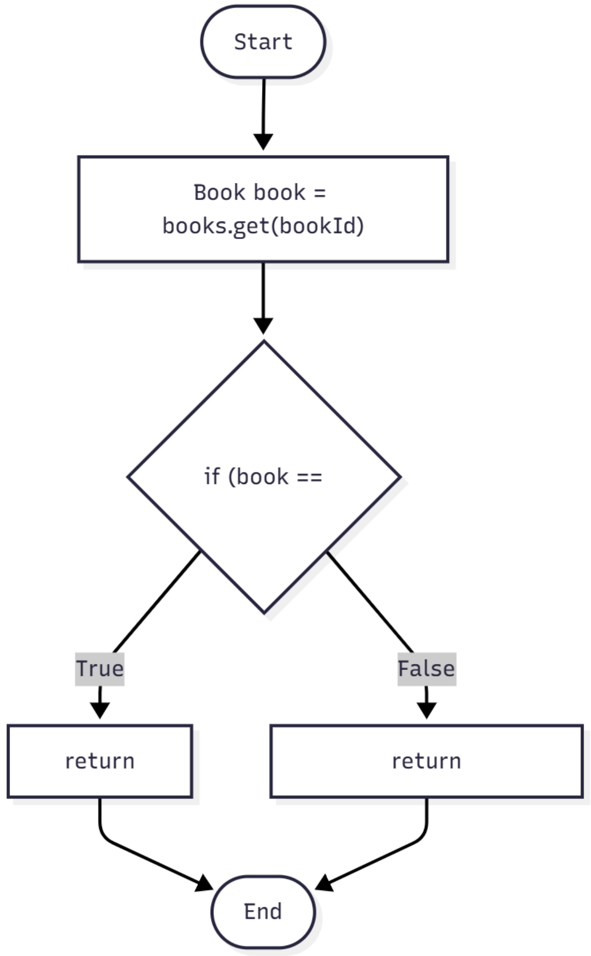
Cyclomatic Complexity: V(G) = E - N + 2 = 6 - 6 + 2 = 2

Independent Paths: 2

Path 1: Start → N1 → N2 → N3 → End (book is null)

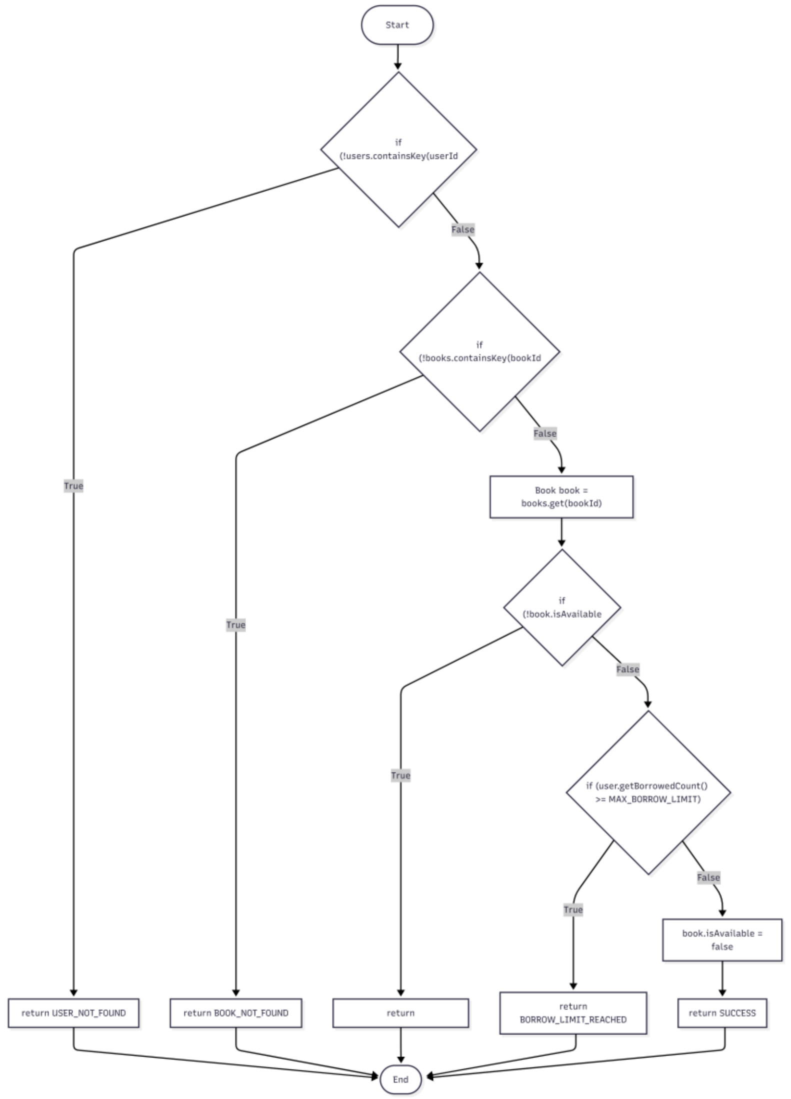
Path 2: Start → N1 → N2 → N4 → End (book exists, return availability)

Test Requirements: Minimum 2 tests to achieve branch coverage



Borrow Book Method with Multiple Conditions

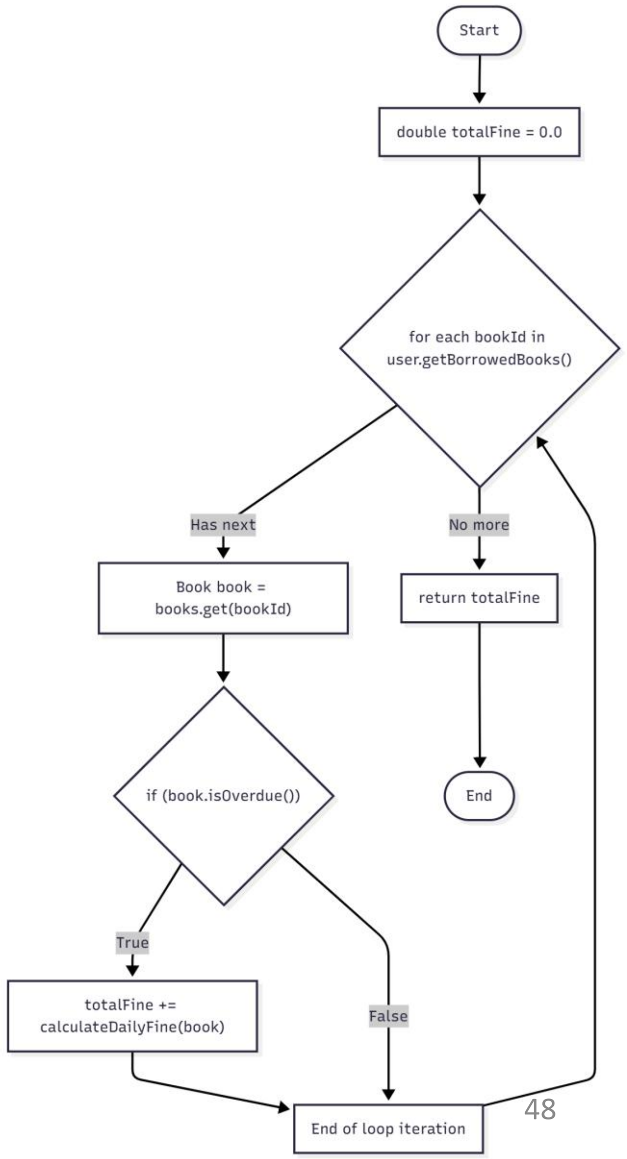
```
public BorrowResult borrowBook(String userId, String bookId) {
   if (!users.containsKey(userId)) {
      return BorrowResult.USER_NOT_FOUND;
   }
   if (!books.containsKey(bookId)) {
      return BorrowResult.BOOK_NOT_FOUND;
   }
   Book book = books.get(bookId);
   User user = users.get(userId);
   if (!book.isAvailable) {
      return BorrowResult.BOOK_UNAVAILABLE;
   }
   if (user.getBorrowedCount() >= MAX_BORROW_LIMIT) {
      return BorrowResult.BORROW_LIMIT_REACHED;
   }
   // Success case
   book.isAvailable = false;
   user.borrowBook(bookId);
   return BorrowResult.SUCCESS;
}
```



Start
if (!users.containsKey(userId
False
True
if (!books.containsKey(bookId
False
True
Book book = books.get(bookId)
if (!book.isAvailable
False
True
if (user.getBorrowedCount() >= MAX_BORROW_LIMIT)
True
False
book.isAvailable = false
return USER_NOT_FOUND
return BOOK_NOT_FOUND
return
return BORROW_LIMIT_REACHED
return SUCCESS
End

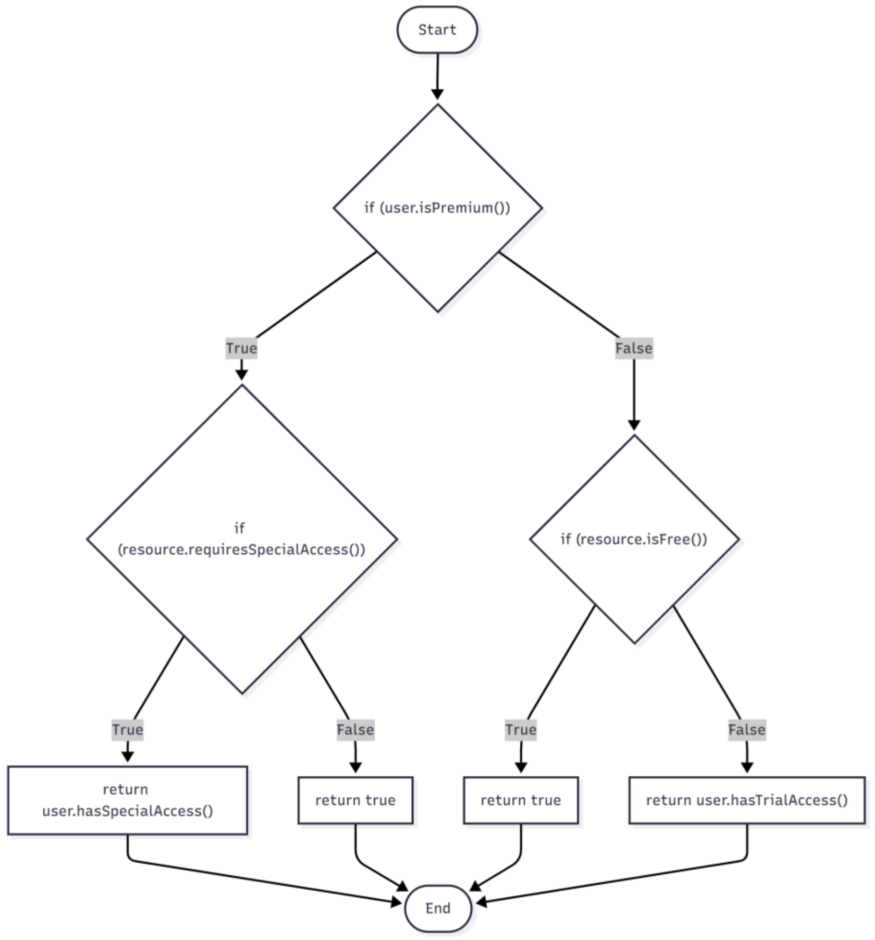
CFG with Loop (Calculate Total Fine)

```
public double calculateTotalFine(User user) {
    double totalFine = 0.0;
    for (String bookId : user.getBorrowedBooks()) {
        Book book = books.get(bookId);
        if (book.isOverdue()) {
            totalFine += calculateDailyFine(book);
        }
    }
    return totalFine;
}
```



Nested If-Else CFG (User Privilege Check)

```
public boolean canAccessPremium(User user, Resource resource) {
   if (user.isPremium()) {
      if (resource.requiresSpecialAccess()) {
         return user.hasSpecialAccess();
      } else {
         return true;
      }
   } else {
      if (resource.isFree()) {
         return true;
      } else {
         return user.hasTrialAccess();
      }
   }
}
```



Discussion [讨论]:

❑Nested decisions: This creates 4 possible paths

❑Cyclomatic Complexity: V(G) = 3 decision nodes + 1 = 4

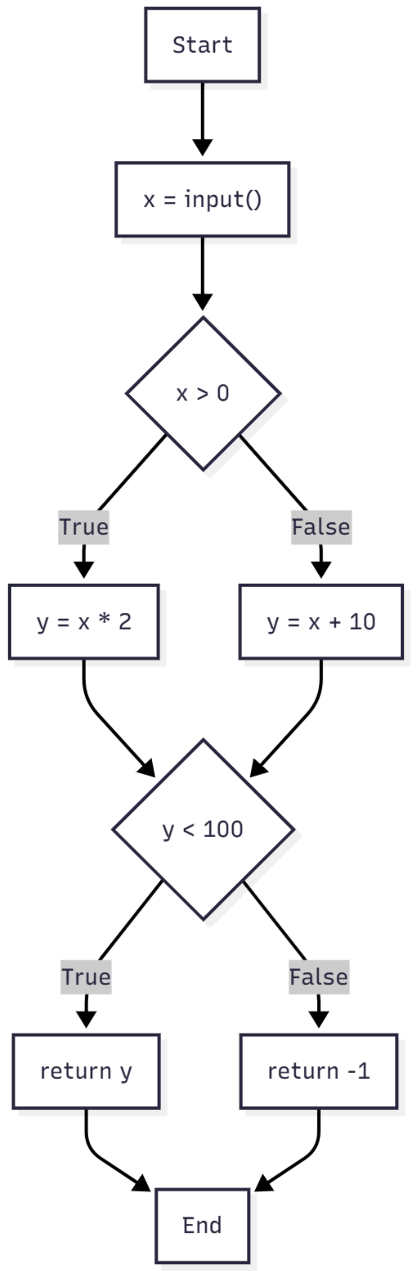
❑Test Requirements: Need 4 tests for full branch coverage:

1. Premium user, resource requires special access
2. Premium user, resource doesn't require special access
3. Non-premium user, free resource
4. Non-premium user, paid resource

❑Matrix Testing: This is ideal for decision table testing

学生实践练习
Practical Exercise for Students

- ❑What is the cyclomatic complexity?
- ❑How many test cases for branch coverage?
- ❑List the independent paths
- ❑Design test cases to cover all branches



主要要点
Key Takeaways

❑Graph-Based Testing provides systematic way to achieve structural coverage

❑Mutation Testing evaluates test suite quality by creating artificial bugs

❑Theoretical Limits (Halting Problem) explain why perfect testing is impossible

❑Practical Approach: Combine techniques for maximum effectiveness