中山大學
SUN YAT-SEN UNIVERSITY

Chapter 4
Software Dynamic Testing

Software Testing: Approaches & Technologies
School of Data & Computer Science, Sun Yat-sen University

Outline

- 4.1 白盒测试
- 4.2 黑盒测试
- 4.3 灰盒测试
- 4.4 测试用例设计
- 4.5 单元测试
- 4.6 集成测试
- 4.7 确认测试
- 4.8 系统测试
- 4.9 动态测试工具

Outline

- 4.1 白盒测试
  - 逻辑覆盖
  - 路径测试
  - 数据流测试
  - 信息流分析
  - 覆盖率分析
  - 覆盖测试准则
  - 实例：基本路径测试

Outline

- **动态测试概述**
  - 动态测试基本流程
    - ✧ 运行被测程序
    - ✧ 检查运行结果与预期结果的差异
    - ✧ 分析运行效率和健壮性等性能
  - 动态测试由三部分组成
    - ✧ 测试实例的构造
    - ✧ 被测程序的执行
    - ✧ 输出结果分析

Outline

- **动态测试概述**
  - 动态测试的分类
    - 从是否关注软件内部结构的角度划分：
      - 白盒测试
      - 黑盒测试
      - 灰盒测试
    - 从软件开发过程的角度划分：
      - 单元测试、集成测试、确认测试、系统测试、验收测试及回归测试
    - 从测试执行时是否需要人工干预的角度划分：
      - 人工测试
      - 自动化测试
    - 从测试实施组织的角度划分：
      - 开发方测试 (α 测试)、用户测试 (β 测试)、第三方测试

4.1 白盒测试

4.1.1 白盒测试概述

- 白盒测试的概念
  - 白盒测试按照程序内部逻辑结构和编码结构来设计测试数据并完成测试，是一种典型的动态测试方法。
    - 白盒测试又称为结构测试或逻辑驱动测试。
    - 白盒测试覆盖全部代码、分支、路径和条件。
    - 白盒测试直接通过已知代码的实现方式，确定测试内容和测试方法。
- 白盒测试的主要特点
  - 可以构造测试数据以测试特定程序部分
  - 有一定的充分性度量手段
  - 可以获得较多工具支持
  - 通常只用于单元测试

4.1 白盒测试

4.1.1 白盒测试概述

- 白盒测试的基本测试内容
  - 对程序模块的所有独立执行路径至少测试一次。
  - 对程序模块中所有的逻辑判定，取“真”与取“假”的两种情况都至少测试一次。
  - 在循环边界和运行边界的界限内执行循环体。
  - 测试内部数据结构的有效性。
- 白盒测试采用的测试方法
  - 逻辑覆盖
    - 包括语句覆盖、分支覆盖、条件覆盖、分支-条件覆盖、条件组合覆盖以及路径覆盖 。
  - 路径测试
  - 数据流测试
  - 信息流分析

4.1 白盒测试

4.1.2 逻辑覆盖

- 逻辑覆盖概述
  - 什么是逻辑覆盖
    - ✧ 逻辑覆盖是以程序内部的逻辑结构为基础的一种白盒测试方法。
    - ✧ 逻辑覆盖方法建立在测试人员对程序的逻辑结构清晰了解的基础上，是一大类测试过程的总称。
  - 逻辑覆盖方法的主要分类
    - ✧ 语句覆盖
    - ✧ 判定覆盖
    - ✧ 条件覆盖
    - ✧ 判定/条件覆盖
    - ✧ 条件组合覆盖
    - ✧ 路径覆盖

4.1 白盒测试

4.1.2 逻辑覆盖

- 语句覆盖
  - 语句覆盖要求设计足够多的测试用例，使得被测程序的每一条语句至少被执行一次。
    - ✧ 例1：有 C 语言程序段落

```
func(int a, b, x)  {
   if ((a > 1) && (b == 0))
      x = x / a;
   if ((a == 2) || (x > 1))
      x = x + 1;
}
```

    - ✧ 测试用例 (a, b, x) = (2, 0, 3) 满足语句覆盖要求。
    - ✧ 语句覆盖可能对程序的逻辑覆盖很少，是弱的逻辑覆盖标准。

4.1 白盒测试

4.1.2 逻辑覆盖

- 语句覆盖 (续)
  - 语句覆盖的优点
    - ✧ 检查所有语句
    - ✧ 结构简单的代码的测试效果较好
    - ✧ 容易实现自动测试
    - ✧ 代码覆盖率高
    - ✧ 如果是程序块覆盖，则不用考虑程序块中的源代码
  - 语句覆盖不能检查出的错误
    - ✧ 条件语句错误
      - ◌ 例如 if ((a>1) && (b==0)) 误为 if ((a>0) && (b==0))
    - ✧ 逻辑运算错误
      - ◌ 例如 if ((a>1) && (b==0)) 误为 if ((a>0) || (b==0))
    - ✧ 循环语句错误，例如循环控制次数错误、循环条件错误。

4.1 白盒测试

4.1.2 逻辑覆盖

- 语句覆盖 (续)
  - 语句覆盖与编程规范
    - ✧ 思考：有 C 语言程序段落

```
func(int a, b, x) {
   if ((a > 1) && (b = 0))
      x = x / a;
   if ((a = 2) || (x > 1))
      x = x + 1;
}
```

- ✧ 编码中有哪些地方不符合规范？
- ✧ 例1的测试用例 (a, b, x) = (2, 0, 3) 还能不能满足语句覆盖要求？

4.1 白盒测试

4.1.2 逻辑覆盖

- 判定覆盖 (分支覆盖)
  - 判定覆盖的测试用例
    - 判定覆盖要求设计足够多的测试用例，使得被测程序中的每一个 (判定) 分支至少通过一次。
      - 每一条分支语句的“真”值和“假”值都至少执行一次。
      - While 语句、switch 语句、异常处理、跳转语句和三目运算符(a?b:c) 等等同样可以使用判定覆盖来测试。
      - 对多分支语句，例如 C 语言中的 case 语句，判定覆盖必须对每一个分支的每一种可能的结果都进行测试。

4.1 白盒测试

4.1.2 逻辑覆盖

- 判定覆盖 (续)
  - 判定覆盖的测试用例 (续)
    - ✧ 例2：有 C 语言程序段落

```
func(int a, b, x)  {
   if ((a > 1) && (b == 0))
      x = x / a;
   if ((a == 2) || (x > 1))
      x = x + 1;
   else
      x = x - 1;
}
```

- ✧ 测试用例 (a, b, x) = (3, 0, 1) 和 (a, b, x) = (2, 1, 3) 满足判定覆盖要求。

4.1.2 逻辑覆盖

- 判定覆盖 (续)
  - 判定覆盖的利弊
    - ✧ 判定覆盖的查错能力强于语句覆盖。
      - ◌ 执行了判定覆盖，实际上也就执行了语句覆盖
    - ✧ 判定覆盖与语句覆盖存在同样的缺点。
      - ◌ 不能查出条件语句错误
      - ◌ 不能查出逻辑运算错误
      - ◌ 不能查出循环次数错误
      - ◌ 不能查出循环条件错误

4.1 白盒测试

4.1.2 逻辑覆盖

- 条件覆盖
  - 条件覆盖的测试用例
    - ✧ 条件覆盖要求设计足够多的测试用例，使得程序中的每一个判断中的每个条件获得所有各种可能结果。
      - ◌ 条件覆盖不一定能够覆盖全部分支。
      - ◌ 判定覆盖只关心整个判定表达式的结果，条件覆盖关心的则是每个条件各种取值的结果。

4.1.2 逻辑覆盖

- 条件覆盖 (续)
  - 条件覆盖的测试用例 (续)
    - ✧ 例2：考虑 C 语言程序段落

```
func(int a, b, x) {
   if ((a > 1) && (b == 0))
      x = x / a;
   if ((a == 2) || (x > 1))
      x = x + 1;
   else
      x = x - 1;
}
```

- ✧ 测试用例 (a, b, x) = (2, 0, 4) 和 (a, b, x) = (1, 1, 1) 满足条件覆盖。

4.1 白盒测试

4.1.2 逻辑覆盖

- 条件覆盖 (续)
  - 条件覆盖的测试用例 (续)
    - ✧ 例2：考虑 C 语言程序段落

```
func(int a, b, x) {
   if ((a > 1) && (b == 0))
      x = x / a;
   if ((a == 2) || (x > 1))
      x = x + 1;
   else
      x = x - 1;
}
```

- ✧ 思考：测试用例 (a, b, x) = (2, 0, 1) 和 (a, b, x) = (1, 1, 2) 满足条件覆盖，但不满足判定覆盖，分支 x = x - 1 不能得到执行。

4.1.2 逻辑覆盖

- 条件覆盖 (续)
  - 条件覆盖的利弊
    - ✧ 能够检查所有的条件错误
    - ✧ 不一定能够实现对每个分支的检查
    - ✧ 可能需要增加用例数量

4.1 白盒测试

4.1.2 逻辑覆盖

- 判定/条件覆盖
  - 判定/条件覆盖的测试用例
    - ✧ 判定/条件覆盖要求设计足够多的测试用例，使得判定中每个条件的所有可能取值至少能够获取一次，同时每个判断的所有可能的判定结果至少执行一次。
      - ◌ 用于解决条件覆盖不一定包括判定覆盖、判定覆盖也不一定包括条件覆盖的问题。

4.1 白盒测试

4.1.2 逻辑覆盖

- 判定/条件覆盖
  - 判定/条件覆盖的测试用例
    - ✧ 例2：考虑 C 语言程序段落

```
func(int a, b, x) {
   if ((a > 1) && (b == 0))
      x = x / a;
   if ((a == 2) || (x > 1))
      x = x + 1;
   else
      x = x - 1;
}
```

- ✧ 测试用例 (a, b, x) = (2, 0, 4) 和 (a, b, x) = (1, 1, 1) 既满足条件覆盖，也满足判定覆盖。

4.1 白盒测试

4.1.2 逻辑覆盖

- 判定/条件覆盖
  - 判定/条件覆盖的利弊
    - ✧ 既考虑了每一个条件，又考虑了每一个分支，发现错误能力强于单独的分支覆盖和条件覆盖。
    - ✧ 满足判定/条件覆盖要求的测试用例不一定能覆盖所有路径。
    - ✧ 可能需要增加用例数量。

4.1 白盒测试

4.1.2 逻辑覆盖

- 条件组合覆盖
  - 条件组合覆盖的测试用例
    - ✧ 条件组合覆盖要求设计足够多的测试用例，使得每个判定中条件的各种组合至少出现一次。
      - ◌ 满足条件组合覆盖标准的测试用例，也一定满足判定覆盖、条件覆盖和判定/条件覆盖标准。
      - ◌ 条件组合覆盖是前面几种覆盖标准中最强的。
      - ◌ 满足条件组合覆盖要求的测试用例并不一定能覆盖所有路径。
      - ◌ 用例数量可能大量增加。

4.1 白盒测试

4.1.2 逻辑覆盖

- 路径覆盖
  - 路径覆盖的测试用例
    - ✧ 要求设计足够多的测试用例，使得程序中所有的路径都至少执行一次。
  - 路径覆盖的利弊
    - ✧ 实现了所有路径的测试，发现错误能力较强。
    - ✧ 路径数量庞大，覆盖所有路径存在困难。
    - ✧ 用例数量增加。
    - ✧ 某些条件错误可能无法发现。

4.1 白盒测试

4.1.3 路径测试

- 概述
  - 路径测试是根据程序的逻辑控制所产生的路径进行测试用例设计的方法。
  - 路径测试是从一个程序的入口开始，执行所经历的各个语句的完整过程；广义上任何有关路径分析的测试都可以被称为路径测试。
  - 完成路径测试的理想情况是做到路径覆盖。对于高复杂性的程序，要做到路径覆盖 (测试所有可执行路径) 存在很大困难。

4.1.3 路径测试

- DD-路径测试
  - DD-Path
    - ✧ DD-path, decision-to-decision path, the name refers to a sequence of statements that begins with the “out-way” of a decision statement and ends with the “in-way” of the next decision statement. No internal branches occur in such a sequence, so the corresponding code is like a row of dominoes lined up so that when the first falls, all the rest in the sequence fall.

4.1 白盒测试

4.1.3 路径测试

- DD-路径测试
  - DD-Path
    - A chain is defined as a path in which:
      - initial and terminal nodes are distinct, and all interior nodes have indeg = 1 and outdeg = 1.
    - A maximal chain is a chain that is not part of a bigger chain.
    - A DD-path is a set of nodes in a program's flow graph such that one of the following holds:

      (1) It consists of a single node with indeg = 0 (initial node)

      (2) It consists of a single node with outdeg = 0 (terminal node)

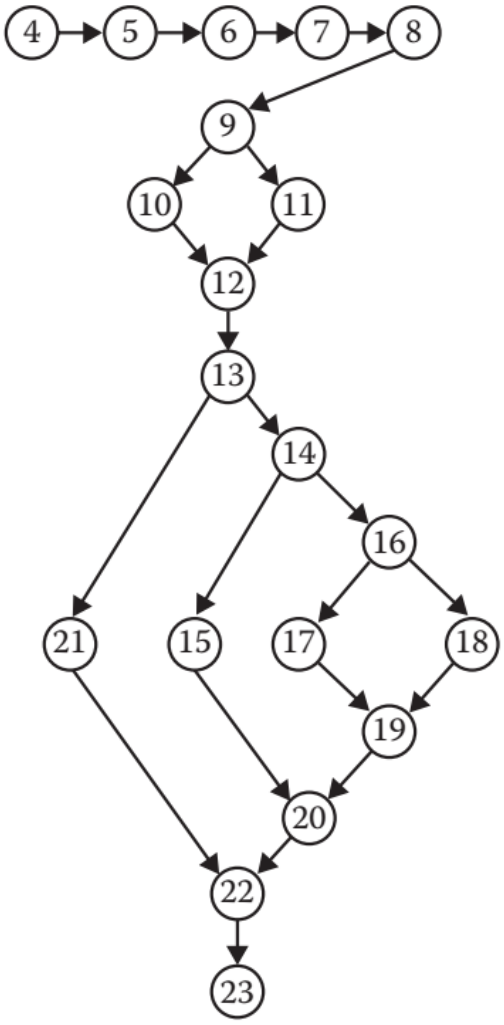
      (3) It consists of a single node with indeg ≥ 2 or outdeg ≥ 2 (decision/merge points)

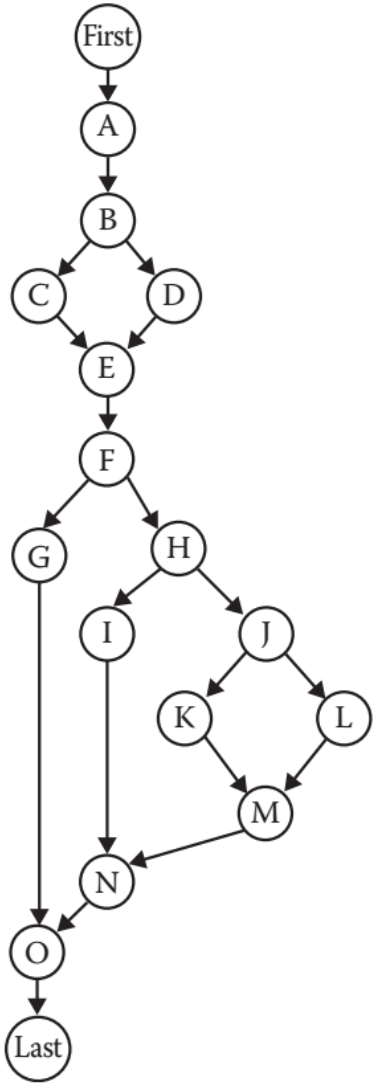
      (4) It consists of a single node with indeg = 1 and outdeg = 1

      (5) It is a maximal chain of length ≥ 1.

4.1 白盒测试

Nodes	DD-Path	Case of definition
4	First	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	Last	2





4.1 白盒测试

4.1.3 路径测试

- DD-路径测试
  - 测试覆盖率
    - ✧ 测试覆盖率考虑对命令式语言 (Imperative Language) 程序流程图中各个分支情况的测试覆盖程度，因此可以对流程图中线性串行的部分进行压缩，在 DD-路径的基础上进行测试用例设计，用测试覆盖指标考察测试效果。压缩图清晰描述了程序执行的分支情况，便于进行覆盖率分析。
    - ✧ 很多质量机构把 DD-路径覆盖作为测试覆盖的最低可接受级别
  - DD-路径的作用
    - ✧ 使用一组满足 DD-路径覆盖要求的测试用例，可以发现约85%的代码缺陷。
    - ✧ 如果 DD-路径图中的每条边得到遍历，则每个判断分支都得到了执行。

4.1 白盒测试

4.1.3 路径测试

- 基本路径测试
  - 概述
    - ✧ 在实际的应用中，一个不太复杂的程序，其路径都是一个庞大的数字。
    - ✧ 要想在测试中覆盖所有路径是不现实的。在不能做到所有路径覆盖的情况下，如果被测程序的每一个独立路径都被测试过，那么可以认为程序中的每个语句都已经检验过或覆盖到。
  - 基本路径测试是 *McCabe* 提出的一种白盒测试方法
    - ✧ 根据过程设计画出程序控制流图 (CFG)；
    - ✧ 计算程序控制流图的 *McCabe* 环路复杂度；
    - ✧ 确定一个线性独立路径 (数量由环路复杂度确定) 的基本集合；
    - ✧ 为上述每条独立路径设计可强制执行该路径的测试用例；
    - ✧ 测试用例总体保证了语句覆盖和条件覆盖。

4.1 白盒测试

4.1.3 路径测试

- 循环路径测试覆盖
  - 循环路径测试分为
    - ✧ 0 次循环 (检查跳出循环)；
    - ✧ 1 次循环 (检查循环初始值)；
    - ✧ 2 次循环 (检查多次循环)；
    - ✧ m 次循环 (检查某次循环)；
    - ✧ 最大次数、比最大次数多一次、比最大次数少一次的循环，检查循环次数边界。
  - 循环过程简化
    - ✧ 循环使路径数量急剧增长，为此需要对循环过程进行简化。
    - ✧ 无论循环的形式和实际执行循环体的次数多少，只考虑循环1次和0次 (即进入循环体一次和跳过循环体)。

4.1 白盒测试

4.1.4 数据流测试

– 数据流测试也称“定义/引用”测试，其目的是发现定义/引用的异常缺陷。
  - ✧ 发现被定义后从未引用的变量
  - ✧ 发现没有被定义的变量
  - ✧ 发现重复定义的变量

4.1 白盒测试

4.1.4 数据流测试

- 数据流测试重点关注的是变量的定义与使用。
  - ✧ 调试修改代码错误时，我们可能会在一段代码中搜索某个变量的定义和引用，考察程序运行时该变量的值的变化，据此分析错误产生的原因。数据流测试将这种方法形式化，便于构造测试算法，实现自动化分析。
  - ✧ 从数据流的角度看，程序是一个程序元素对数据访问的过程。
  - ✧ 数据流测试关注变量定义与引用位置，它是一种结构性测试方法，也可看作是基于路径测试的一种改良方案 (进行“真实性检查”)。
  - ✧ 数据流测试用数据流图描述数据的“定义-使用”对，通过对路径进行“真实性检查”，发现数据的不正确定义及使用。
  - ✧ 一种简单的数据流测试策略要求测试用例覆盖每个“定义-使用”路径一次。

4.1 白盒测试

4.1.5 信息流分析

- 信息流测试通过分析输入数据、输出数据和语句之间的关系来检查程序错误。
  - ✧ 还可用来分析是否存在无用的语句。
- 信息流分析的具体作用
  - ✧ 能够列出对输入变量的所有可能的引用。
  - ✧ 在程序的任何指定点检查某些语句，其执行可能影响某一输出变量值。
  - ✧ 为输入输出关系提供一种检查，看每个输出值是否有相关的输入值，而不是由其他值导出。

4.1 白盒测试

4.1.6 覆盖率分析

- 代码覆盖率是指进行白盒测试时测试用例对程序内部逻辑的覆盖程度。
  - ✧ 最理想的白盒测试是实现程序的语句覆盖、分支覆盖以及路径覆盖，实现难度大，需要采用其它标准来度量覆盖的程度。
  - ✧ 覆盖率分析对代码的执行路径覆盖范围进行评估。这些覆盖从不同要求出发，为测试用例的设计提供依据。
- 覆盖测试的目标
  - ✧ 对程序模块的所有独立的执行路径至少测试一次；
  - ✧ 对所有的逻辑判定，取“真”与取“假”的两种情况都至少测试一次；
  - ✧ 在循环的边界和运行界限内执行循环体；
  - ✧ 测试内部数据结构的有效性；
  - ✧ 其它。

4.1 白盒测试

4.1.6 覆盖率分析

- 覆盖测试实施要点
  - ✧ 内部动作是否合规
    - ◌ 按照程序内部的结构测试程序，检验程序中的每条通路是否都能按规格说明书的规定要求正确工作，而不考虑外部特性。
    - ◌ 覆盖法无法发现程序内在的逻辑错误。
  - ✧ 覆盖法是穷举路径测试
    - ◌ 覆盖法要求全面了解程序内部逻辑结构，对所有逻辑路径进行测试。
  - ✧ 覆盖测试的主要方法
    - ◌ 覆盖测试的主要方法有逻辑驱动、基本路径测试等。
    - ◌ 覆盖测试方法取决于对软件的需求标准。

4.1 白盒测试

4.1.6 覆盖率分析

– 逻辑覆盖率计算方法

✧ 逻辑覆盖率主要指语句覆盖率、判定覆盖率、条件覆盖率、判定/条件覆盖率、条件组合覆盖率和路径覆盖率。

覆盖率 = (至少被执行一次的 item 数)/item 的总数.

公式对 item 的覆盖情况进行计算，item 可以是需求、语句、分支、条件、路径等等。

✧ 覆盖率是用来度量测试完整性的一个手段，不是测试的目的。

✧ 通过覆盖率数据，可以估计测试是否充分，测试弱点在哪些方面，进而指导我们去设计能够增加覆盖率的测试用例。

4.1 白盒测试

4.1.7 覆盖测试准则

- ESTCA 准则
  - ESTCA：错误敏感测试用例分析 (Error Sensitive Test Cases Analysis, *K. A. Foster*)
  - 规则1：对于 A rel B (rel 是关系符 <, ==, 或 >， A, B 是变量) 型的分支谓词，适当选择 A 与 B 的值，使得测试执行到该分支语句时，A < B, A == B 和 A > B 的情况分别出现一次。
    - 目的：检测 rel 的错误。
  - 规则2：对于 A rel1 C (rel1 是 < 或 >，A 是变量，C 是常量) 型的分支谓词，当 rel1 为 < 时，适当选择 A，使 A = C-M (M 是距 C 最小的容许正数，若 A 和 C 均为整型时，M = 1)；同理，当 rel1 为 > 时，适当选择 A，使 A = C + M。
    - 目的：检测“差1”类错误。
      - 如本应是“if A > 1”而错成“if A > 0”。

4.1.7 覆盖测试准则

- ESTCA 准则 (续)
  - 规则3：对外部输入变量赋值，使其在每一测试用例中均有不同的值与符号，并与同一组测试用例中其它变量的值与符号不一致。
    - 目的：检测程序语句中的错误。
      - 例：如果将引用一个外部输入变量错写成引用一个常量，将导致应用规则3的两次测试结果相同。
  - 上述三项规则适用基于经验的测试用例设计，虽然不是完备的，但规则本身针对的是程序编写人员容易发生的错误，或是围绕着发生错误的频繁区域，提高了发现错误的命中率，在普通程序的测试中确实有效。

4.1 白盒测试

4.1.7 覆盖测试准则

- LCSAJ 覆盖准则
  - LCSAJ：线性代码序列和跳转 (Linear Code Sequence and Jump Coverage, *M. R. Woodward*)。
  - 一个 LCSAJ 是一组顺序执行的程序代码。
    - 起始于程序的入口，或者是一个转移语句的入口点，或者是一个控制流可跳达的点；
    - 结束于程序的出口或者是一个可能导致控制流跳转的点。
  - 程序的 LCSAJ 路径
    - 几个 LCSAJ 首尾相接构成一个 LCSAL 串，如果第一个 LCSAJ 起点为程序起点，最后一个 LCSAJ 终点为程序终点，则组成程序的一条 LCSAJ 路径。
  - LCSAJ 路径不同于 DD-Path。DD-Path 由程序流程图决定，一个 DD-Path 是两个判断之间的路径，但其中不再有判断。

4.1 白盒测试

4.1.7 覆盖测试准则

- LCSAJ 覆盖准则 (续)
  - LCSAJ 覆盖准则是一个分层的覆盖准则：
    - 第1层：语句覆盖
    - 第2层：分支覆盖
    - 第3层：LCSAJ 覆盖
      - 程序中的每一个 LCSAJ 至少在测试中经历一次。
    - 第4层：两两 LCSAJ 覆盖
      - 程序中每两个首尾相连的 LCSAJ 组合起来在测试中都要经历一次。
    - 第 n+2 层：每 n 个首尾相连的 LCSAJ 组合在测试中都要经历一次。

4.1.7 覆盖测试准则

- LCSAJ 覆盖准则 (续)
  - LCSAJ 覆盖准则的应用
    - 按照 LCSAJ 覆盖准则，越是高层的覆盖越难满足。
    - 在实施测试时，若要实现上述层次 LCSAJ 覆盖，需要产生被测程序的所有 LCSAJ。
    - 尽管 LCSAJ 覆盖要比判定覆盖复杂的多，但是 LCSAJ 的自动化过程相对比较容易实现。
    - 一个模块的微小改动都可能对 LCSAJ 产生重大影响，因此维护 LCSAJ 的测试数据相当困难。
    - 一个大模块包含极其庞大的 LCSAJ，因此要获得100%的覆盖率并不现实。
    - 证据表明，把测试100%的 LCSAJ 作为目标比100%的判定覆盖要有效的多。

4.1 白盒测试

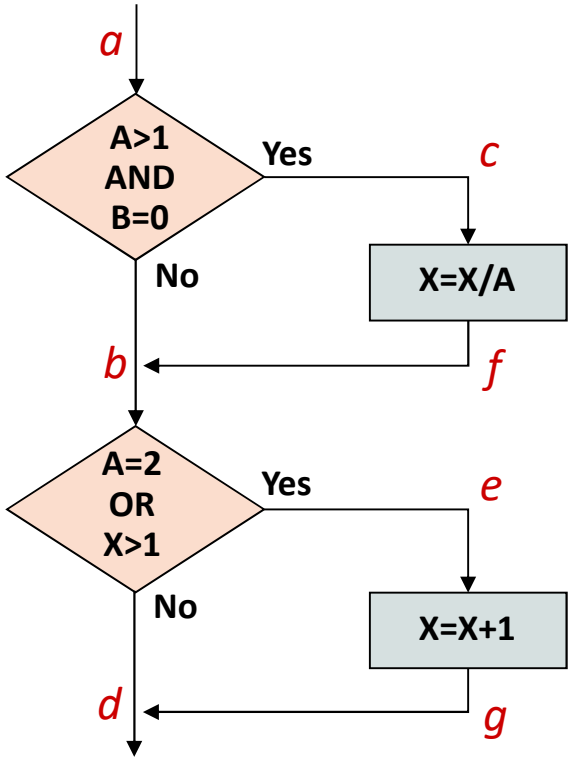
4.1.8 实例：基本路径测试

– 路径测试

✧ 设计足够多的测试用例，覆盖被测试对象中的所有可能路径。

✧ 例4：程序流程图如右图所示，设计路径测试用例。

测试用例	通过路径标志
A=2, B=0, X=3	*acfbegd*
A=1, B=0, X=1	*abd*
A=2, B=1, X=1	*abegd*
A=3, B=0, X=1	*acfbd*



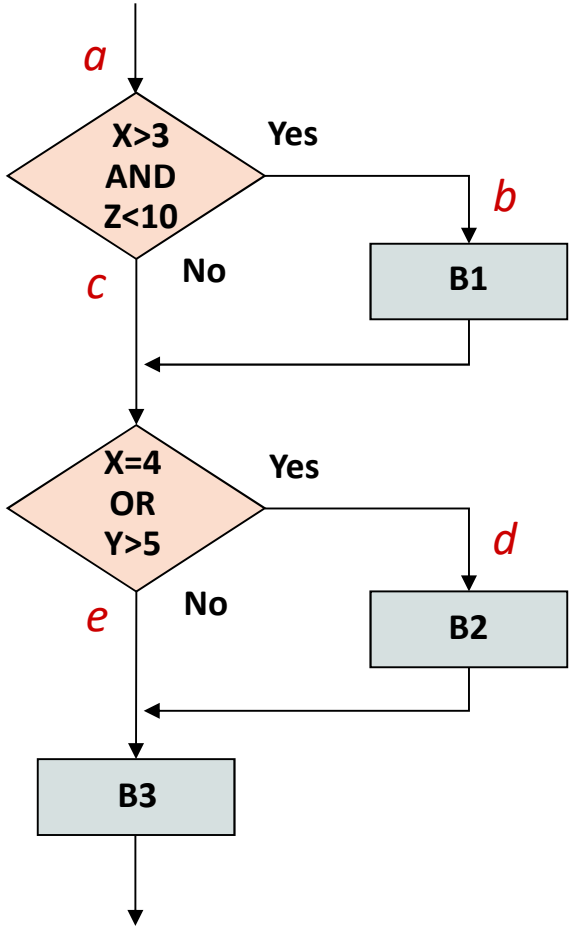
4.1.8 实例：基本路径测试

– 路径测试 (续)

✧ 例5：程序流程图如右图所示。

◌ 条件定义

- T1: X>3, T2: Z<10.
- T3: X=4, T4: Y>5.



测试用例	通过路径	覆盖条件
x=4, y=6, z=5	*abd*	T1、T2、T3、T4
x=4, y=5, z=15	*acd*	T1、F2、T3、F4
x=2, y=5, z=15	*ace*	F1、F2、F3、T4
x=5, y=5, z=5	*abe*	T1、T2、F3、F4

4.1 白盒测试

4.1.8 实例：基本路径测试

- 基本路径测试
  - ✧ 在实际的应用中，一个不太复杂的程序，其路径都是一个庞大的数字，要想在测试中覆盖所有路径是不现实的；在不能做到所有路径覆盖的前提下，如果某一程序的每一个独立路径都被测试过，那么可以认为程序中的每个语句都已经检验过或覆盖到。
  - ✧ 基本路径测试在程序控制流图的基础上，通过分析 *McCabe* 环路复杂性，导出基本可执行路径集合，实现测试用例的设计。设计出的测试用例要保证在测试中程序的每一个可执行语句至少执行一次。

4.1 白盒测试

4.1.8 实例：基本路径测试

- 基本路径测试 (续)
  - 前提条件
    - 测试人员已经对被测试对象有了一定的了解，基本明确被测试软件的逻辑结构。
  - 测试过程
    - 针对程序逻辑结构设计和加载测试用例，驱动程序执行，对程序路径进行测试。
  - 测试结果
    - 分析实际的测试结果与预期的结果是否一致。

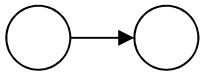
4.1 白盒测试

4.1.8 实例：基本路径测试

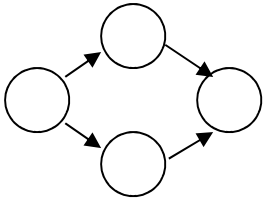
- 基本路径测试的设计步骤
  - ✧ 根据过程设计 (通常是程序流程图) 画出程序的控制流图。
  - ✧ 计算程序环路复杂度。
    - ◌ 环路复杂度是 *McCabe* 复杂性度量之一，从其中导出的程序基本路径集合中的独立路径条数，是确定程序中每个可执行语句至少执行一次所必须的最少测试用例数目。
  - ✧ 确定一个线性独立路径 (数量由环路复杂度确定) 的基本集合。
  - ✧ 准备测试用例：设计用例数据输入，确保基本路径集中的每一条路径的执行。
  - ✧ 应用测试用例，结合预期结果进行结果分析。

4.1.8 实例：基本路径测试

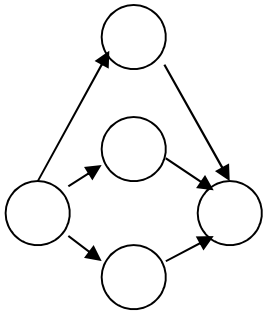
- 程序控制流图
  - ✧ 程序控制流图 (CFG, Control Flow Graph) 不同于程序流程图 (Program Flowchart)，是测试程序过程处理的一种表示。
  - ✧ 控制流图使用下面的符号描述逻辑控制流，每一种结构化元素有一个相应的流图符号。



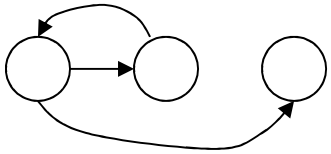
顺序结构



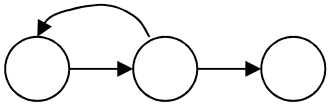
if 结构



case 结构



while 结构



until 结构

4.1 白盒测试

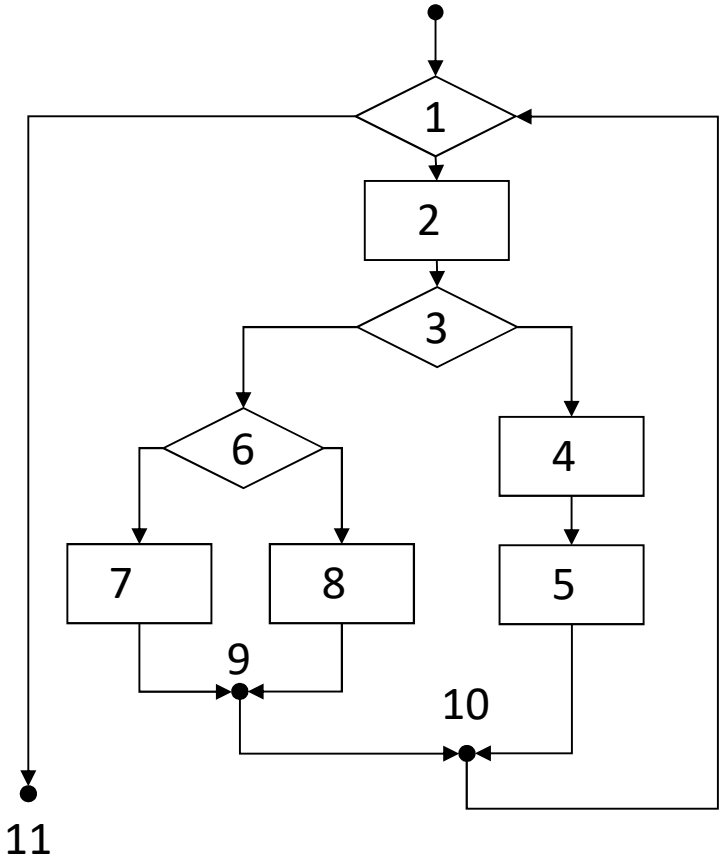
4.1.8 实例：基本路径测试

- 程序控制流图 (续)
  - 程序控制流图只有2种图形符号：
    - 图中的一个圆圈称为流图的结点，代表一条或多条语句。
    - 图中的箭头称为边或连接，代表控制的流向。
  - 任何过程设计都要被翻译成控制流图。
    - 在将程序流程图转化成程序控制流图时，在选择或多分支结构中，分支的汇聚处应有一个汇聚结点。

4.1.8 实例：基本路径测试

– 程序控制流图 (续)

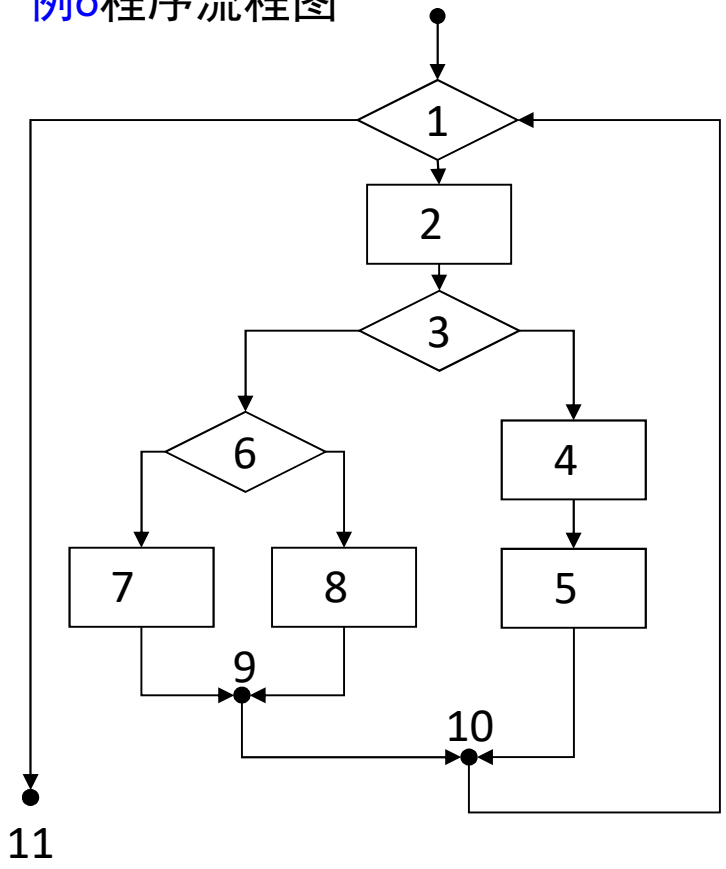
✧ 例6：将下面的程序流程图转换成为等价的程序控制流图。



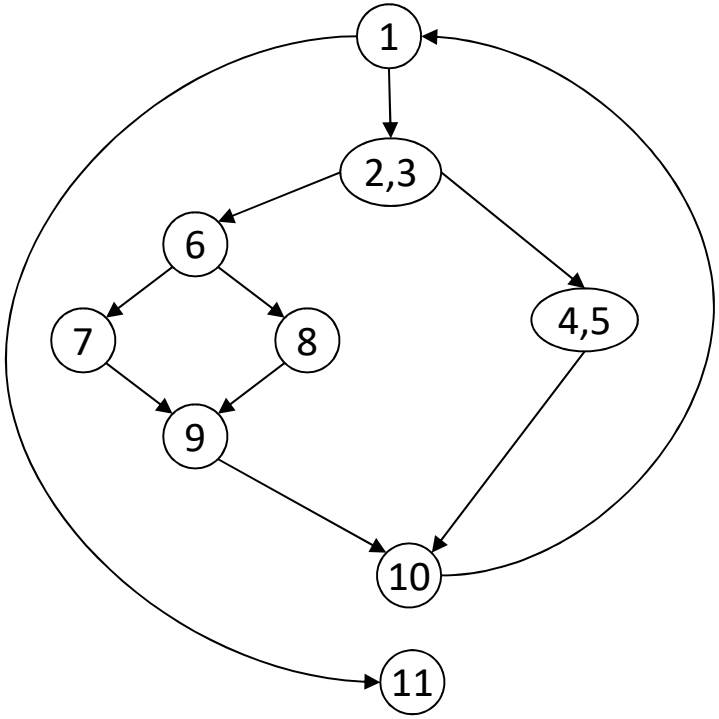
4.1.8 实例：基本路径测试

– 程序控制流图 (续)

例6程序流程图



例6程序控制流图

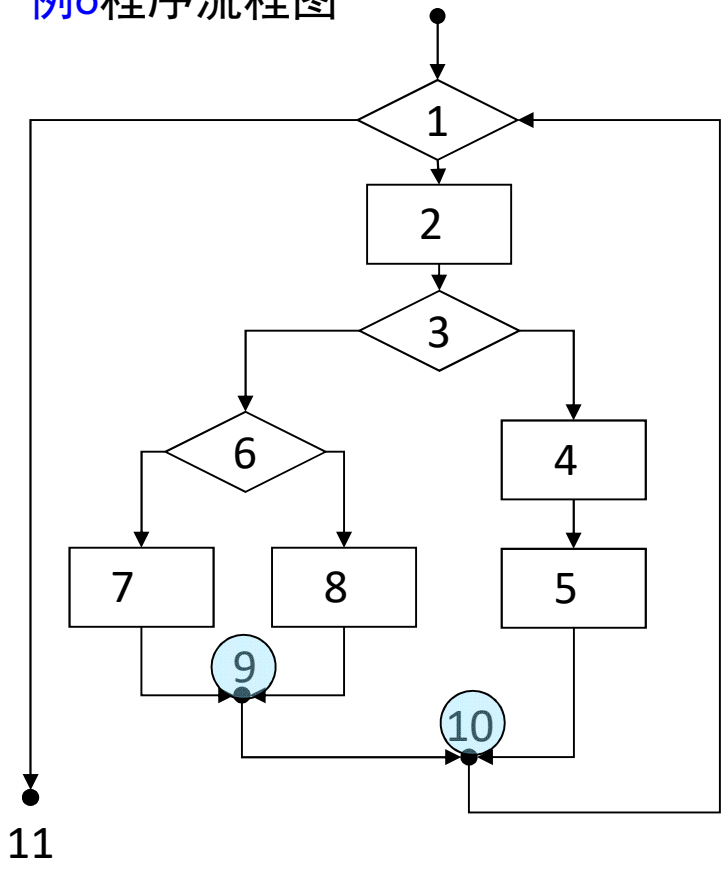


图中有9个结点，11条边和4个域 (包括无穷域)。注意到结点9和结点10是两个汇聚结点。

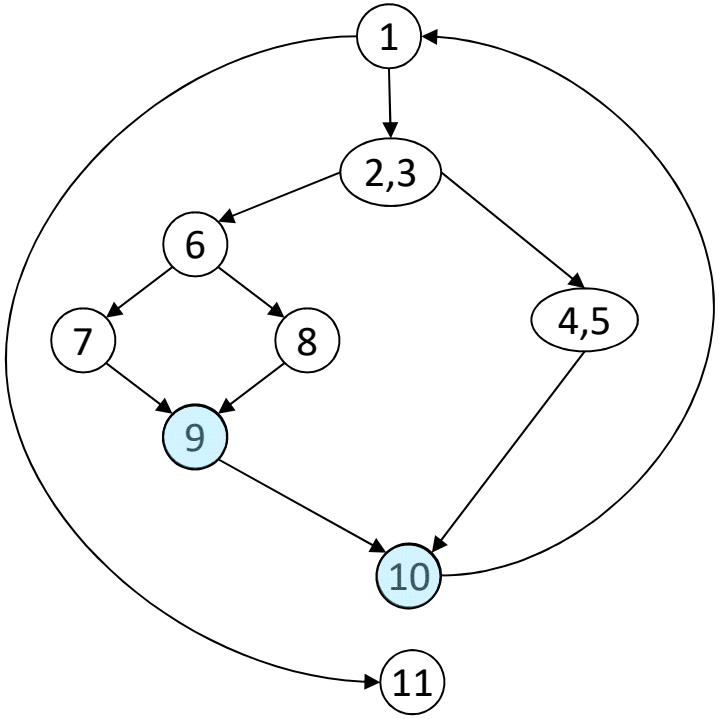
4.1.8 实例：基本路径测试

– 程序控制流图 (续)

例6程序流程图



例6程序控制流图



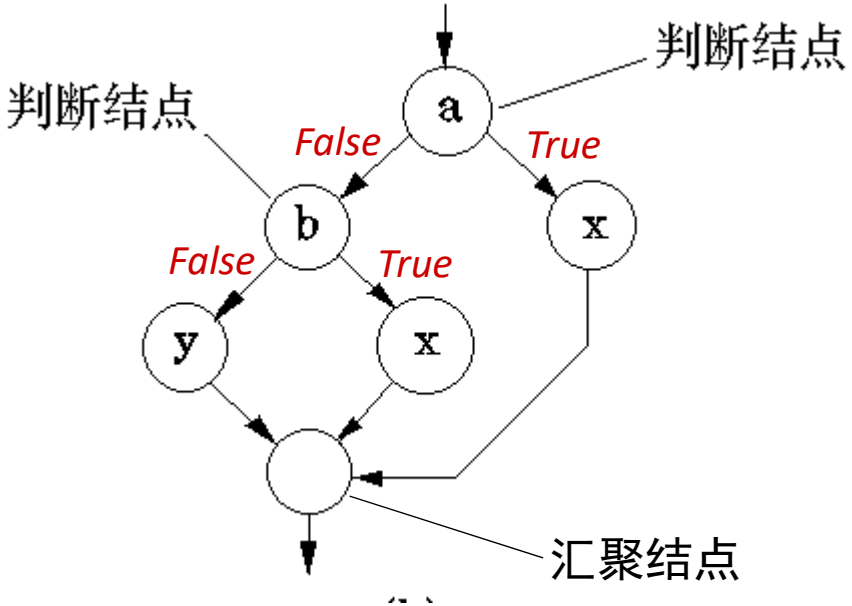
图中有9个结点，11条边和4个域 (包括无穷域)。注意到结点9和结点10是两个汇聚结点。

4.1.8 实例：基本路径测试

- 程序控制流图 (续)
  - ✧ 如果判断中的条件表达式是由一个或多个逻辑运算符连接 (如 OR, AND, NAND, NOR) 的复合条件表达式，则需要等价转换成一系列只含有单条件判定的嵌套的条件表达式。
  - ✧ 例7：下列语句

```
if (a or b)
    x;
else
    y;
```

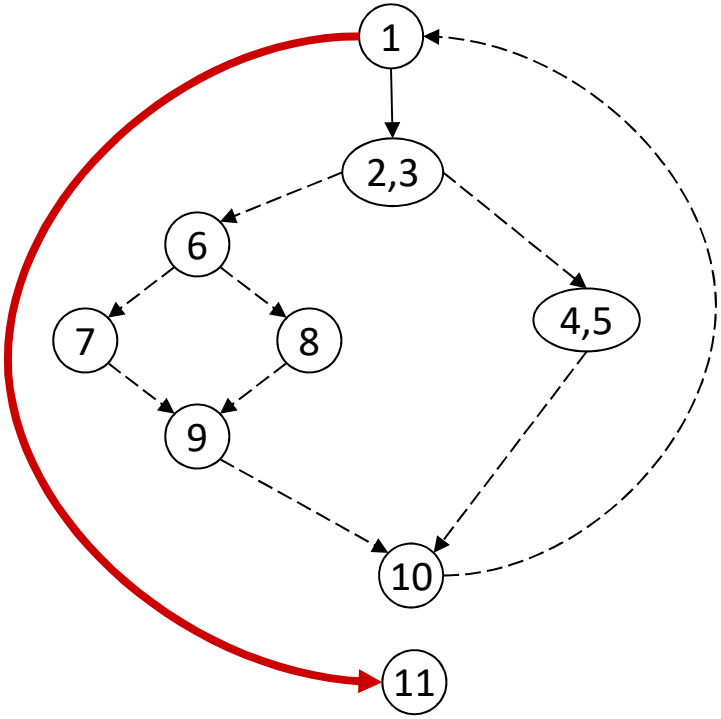
对应的逻辑如右图：



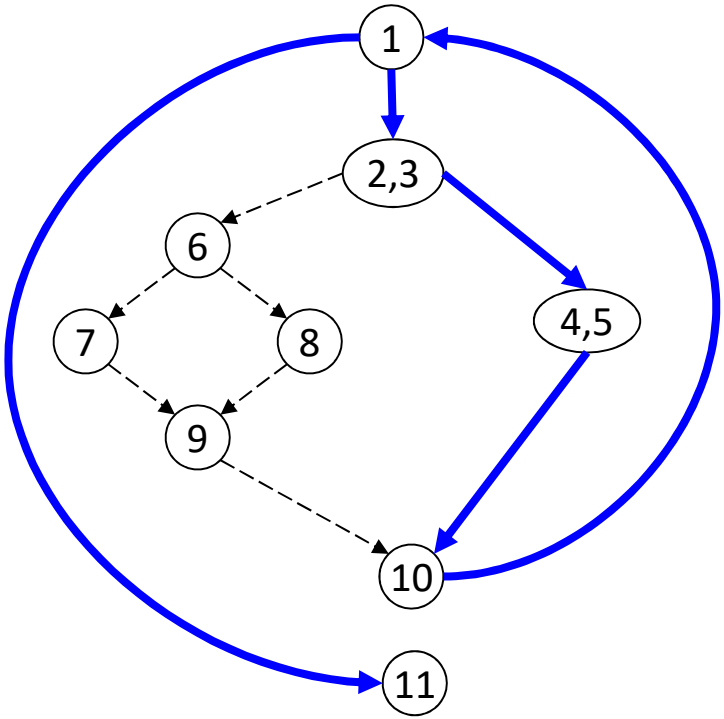
4.1.8 实例：基本路径测试

– 程序控制流图 (续)

✧ 独立路径：至少沿一条新的边移动的路径。如例6中：



路径1：1-11

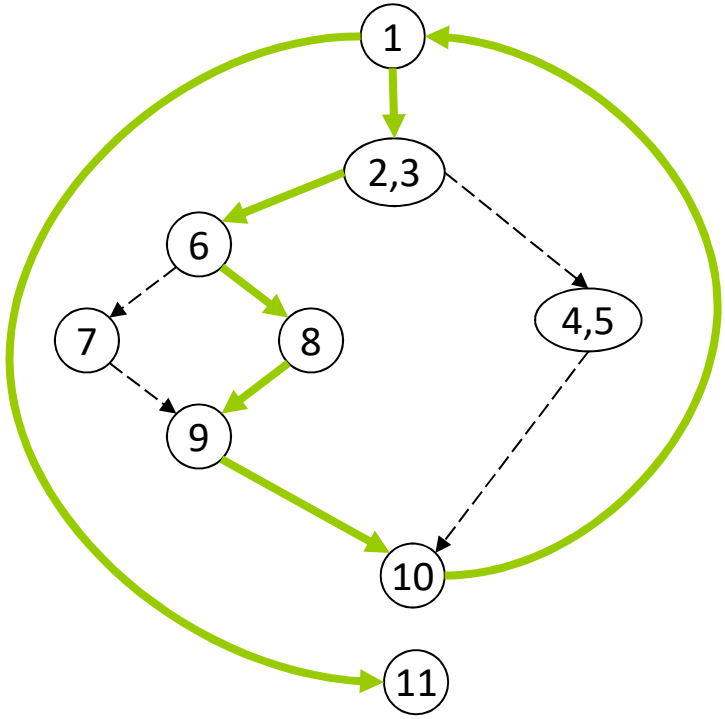


路径2：1-2,3-4,5-10-1-11

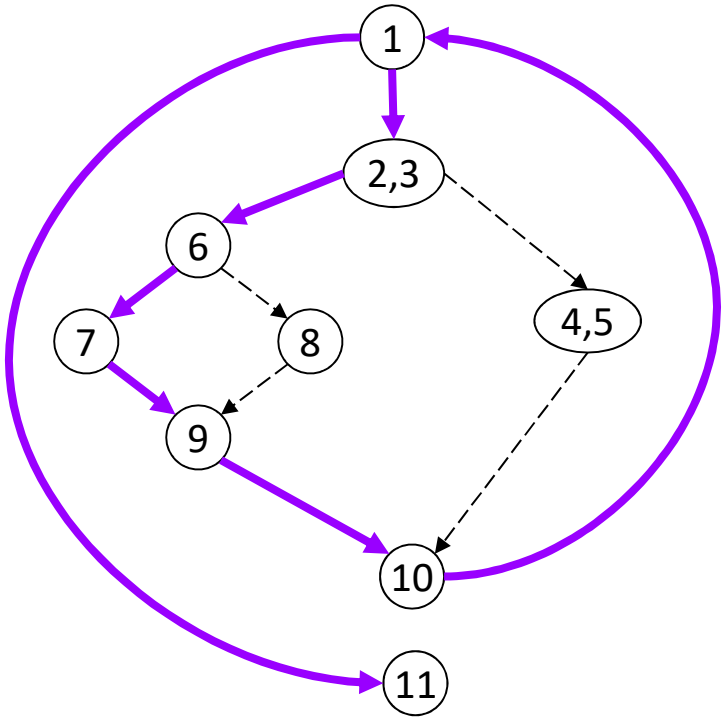
4.1.8 实例：基本路径测试

– 程序控制流图 (续)

✧ 对路径1-4的遍历使得程序中的所有语句至少被执行了一次。



路径3：1-2,3-6-8-9-10-1-11



路径4：1-2,3-6-7-9-10-1-11

4.1.8 实例：基本路径测试

- 基本路径测试的具体步骤
  - ✧ 第一步：画出控制流图
    - ◌ 程序流程图用来描述程序控制结构。将程序流程图映射到一个相应的程序控制流图。控制流图中的圆称为流图的结点，代表一个或多个语句。流程图一个处理方框序列或一个菱形判决框都可被映射为流图的一个结点 (简化起见假设程序流程图的菱形判决框中不包含复合条件)。流图中的箭头称为边或连接，代表控制流向。流图中的一条边必须终止于一个结点。由边和结点限定的封闭范围称为区域。计算区域时应包括外部域。

4.1 白盒测试

4.1.8 实例：基本路径测试

✧ 例8：有 C 函数源代码如下，画出其程序控制流图。

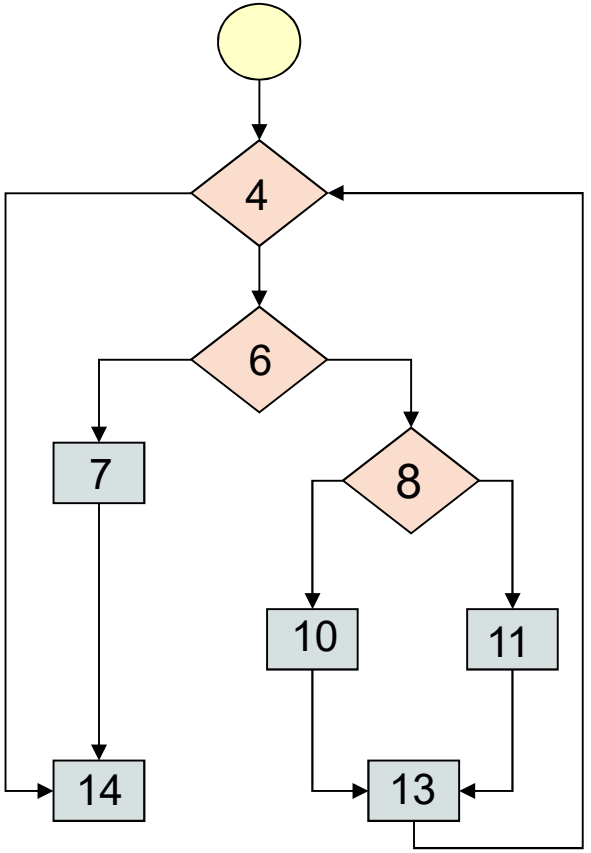
```
      void  Sort( int iRecordNum, int iType )
1.    {
2.       int x = 0;
3.       int y = 0;
4.       while ( iRecordNum - - >= 0 )
5.       {
6.          if ( 0 == iType )
7.             { x = y + 2; break; }
8.          else
9.             if ( 1 == iType )
10.               x = y + 10;
11.            else
12.               x = y + 20;
13.      }
14.   }
```

思考：例8的 C 源代码有没有不符合静态规范的地方？

4.1.8 实例：基本路径测试

✧ 解：该函数的程序流程图如右图。

```
      void  Sort( int iRecordNum, int iType )
1.    {
2.       int x = 0;
3.       int y = 0;
4.       while ( iRecordNum - - >= 0 )
5.       {
6.          if ( 0 == iType )
7.             { x = y + 2; break; }
8.          else
9.             if ( 1 == iType )
10.               x = y + 10;
11.            else
12.               x = y + 20;
13.      }
14.   }
```

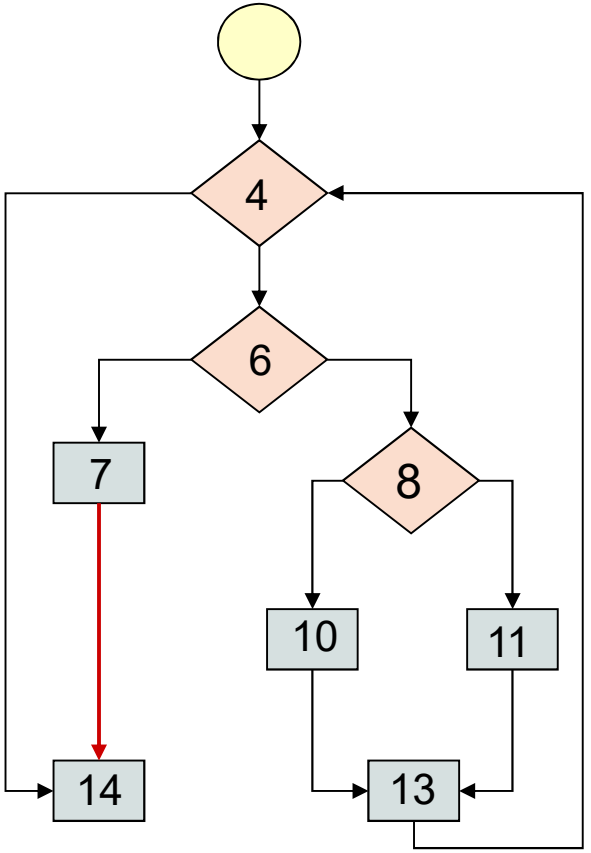


例8的程序流程图

4.1.8 实例：基本路径测试

✧ 解：该函数的程序流程图如右图。

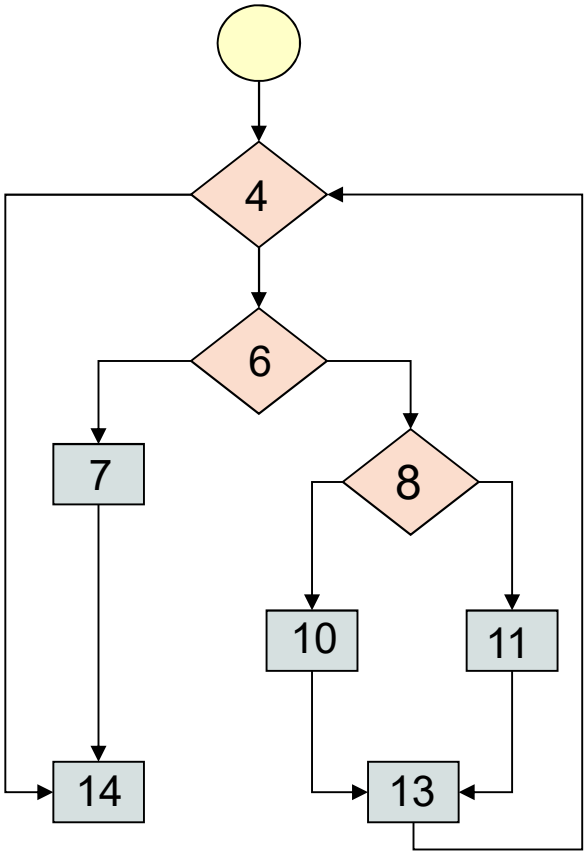
```
      void  Sort( int iRecordNum, int iType )
1.    {
2.       int x = 0;
3.       int y = 0;
4.       while ( iRecordNum - - >= 0 )
5.       {
6.          if ( 0 == iType )
7.             { x = y + 2; break; }  //直接跳出循环
8.          else
9.             if ( 1 == iType )
10.               x = y + 10;
11.            else
12.               x = y + 20;
13.      }
14.   }
```



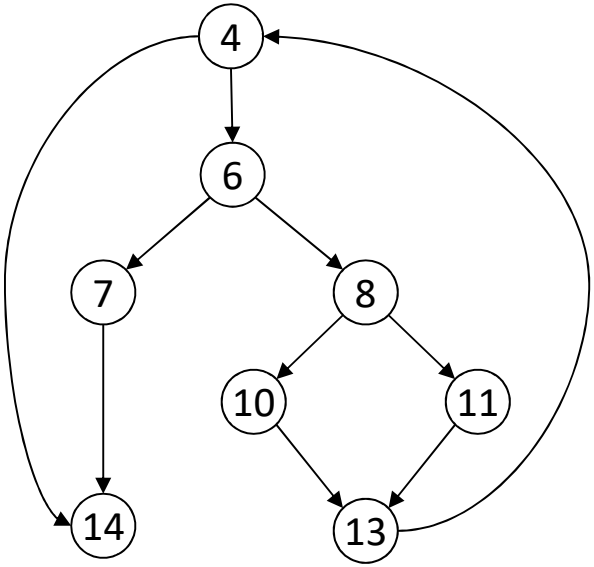
例8的程序流程图

4.1.8 实例：基本路径测试

✧ 解：该函数的控制流图如右图。



例8的流程图



例8的控制流图

4.1.8 实例：基本路径测试

- 基本路径测试的具体步骤
  - ✧ 第二步：计算环路复杂度
    - ◌ *McCabe* 环路复杂度为程序逻辑复杂性提供定量测度。该度量用于计算程序的基本独立路径数目，也即是确保所有语句至少执行一次的起码测试数量。
    - ◌ 有以下三种方法计算环路复杂度：
      - 给定流图 G 的环路复杂度 V(G)，定义为

        $$V(G) = m - n + 2.$$

        m 是流图中边的数量，n 是流图中结点的数量；
      - 平面流图中区域的数量对应于环路复杂度；
      - 给定流图 G 的环路复杂度 V(G)，定义为

        $$V(G) = d + 1.$$

        d 是流图 G 中单判定结点的数量。

4.1.8 实例：基本路径测试

– 基本路径测试的具体步骤

✧ 例9：计算例8的 *McCabe* 环路复杂度。

✧ 解：例8的程序控制流图的环路复杂度计算如下：

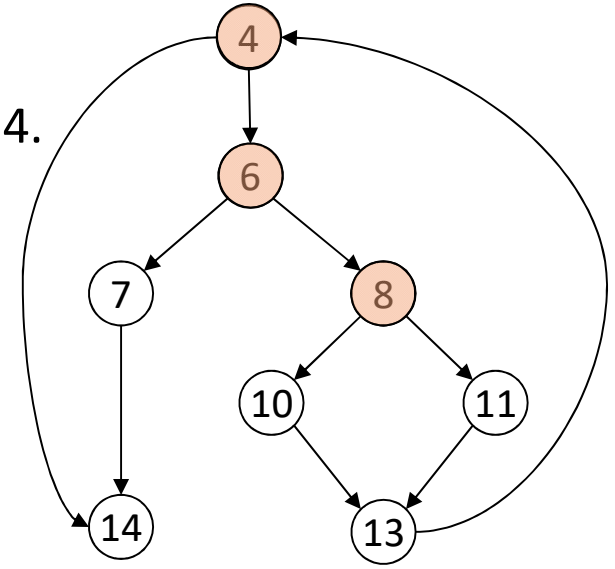
◌ 流图的边数 m=10，点数 n=8，故

V(G) = 10 – 8 + 2 = 4.

◌ 平面流图中有4个区域，故 V(G) = 4.

◌ 流图的单判定结点数 d=3，故

V(G) = 3 + 1 = 4.



例8的程序控制流图

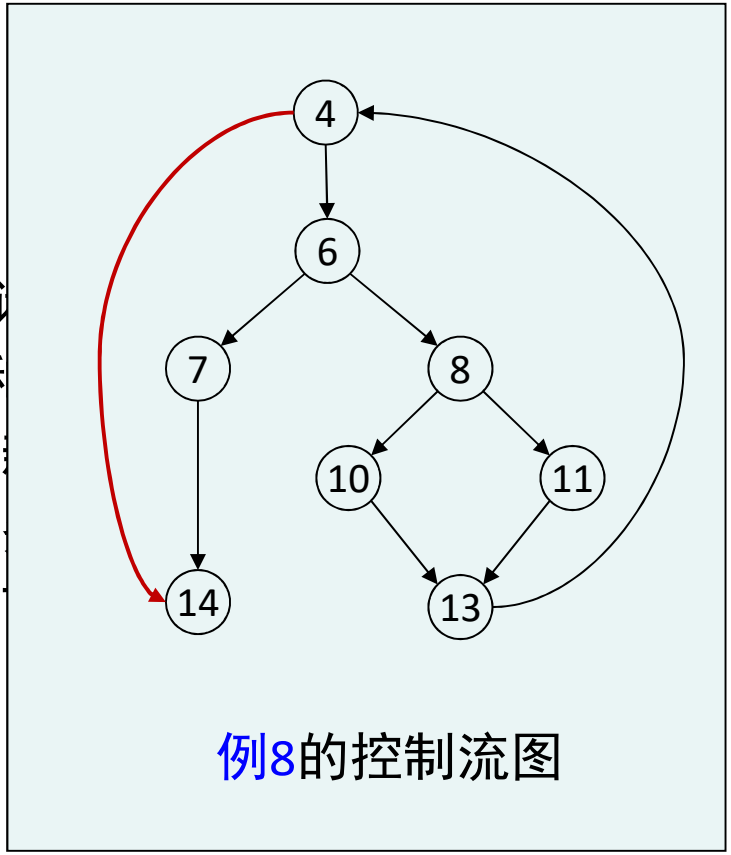
4.1 白盒测试

4.1.8 实例：基本路径测试

- 基本路径测试的具体步骤
  - ✧ 第三步：准备测试用例
    - ◌ 根据 *McCabe* 环路复杂度的计算结果，图中存在4条独立的路径 (一条独立路径是指，和其他的独立路径相比，至少引入一个新处理语句或一个新判断的程序通路。V(G) 值正好等于该程序的独立路径的条数)。
    - ◌ 给出一个独立的路径集合如下：
      - 路径1：4-14
      - 路径2：4-6-7-14
      - 路径3：4-6-8-10-13-4-14
      - 路径4：4-6-8-11-13-4-14
    - ◌ 根据上面的独立路径集合，设计输入用例数据，迫使程序分别执行上面4条路径。

4.1.8 实例：基本路径测试

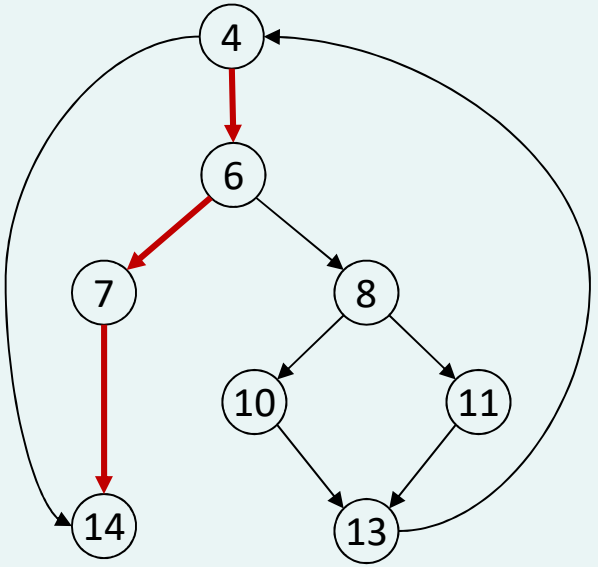
- 基本路径测试的具体步骤
  - ✧ 第三步：准备测试用例
    - ◌ 根据 *McCabe* 环路复杂度的
      路径 (一条独立路径是指，
      引入一个新处理语句或一个
      好等于该程序的独立路径的
    - ◌ 给出一个独立的路径集合如
      - 路径1：4-14
      - 路径2：4-6-7-14
      - 路径3：4-6-8-10-13-4-14
      - 路径4：4-6-8-11-13-4-14
    - ◌ 根据上面的独立路径集合，设计输入用例数据，迫使程序
      分别执行上面4条路径。



例8的控制流图

4.1.8 实例：基本路径测试

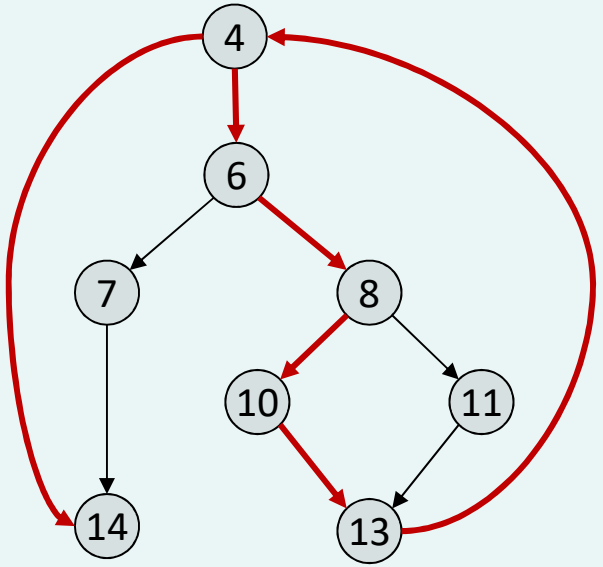
- 基本路径测试的具体步骤
  - ✧ 第三步：准备测试用例
    - ◌ 根据 *McCabe* 环路复杂度的
      路径 (一条独立路径是指，
      引入一个新处理语句或一个
      好等于该程序的独立路径的
    - ◌ 给出一个独立的路径集合如
      - 路径1：4-14
      - 路径2：4-6-7-14
      - 路径3：4-6-8-10-13-4-14
      - 路径4：4-6-8-11-13-4-14
    - ◌ 根据上面的独立路径集合，设计输入用例数据，迫使程序
      分别执行上面4条路径。



例8的控制流图

4.1.8 实例：基本路径测试

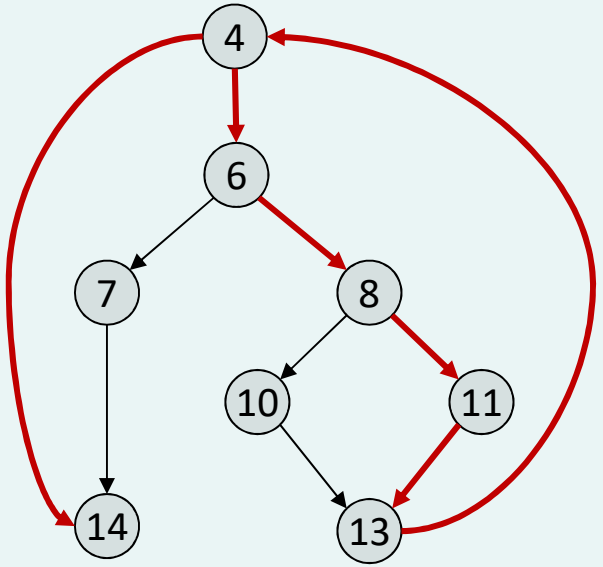
- 基本路径测试的具体步骤
  - ✧ 第三步：准备测试用例
    - ◌ 根据 *McCabe* 环路复杂度的
      路径 (一条独立路径是指，
      引入一个新处理语句或一个
      好等于该程序的独立路径的
    - ◌ 给出一个独立的路径集合如
      - 路径1：4-14
      - 路径2：4-6-7-14
      - 路径3：4-6-8-10-13-4-14
      - 路径4：4-6-8-11-13-4-14
    - ◌ 根据上面的独立路径集合，设计输入用例数据，迫使程序
      分别执行上面4条路径。



例8的控制流图

4.1.8 实例：基本路径测试

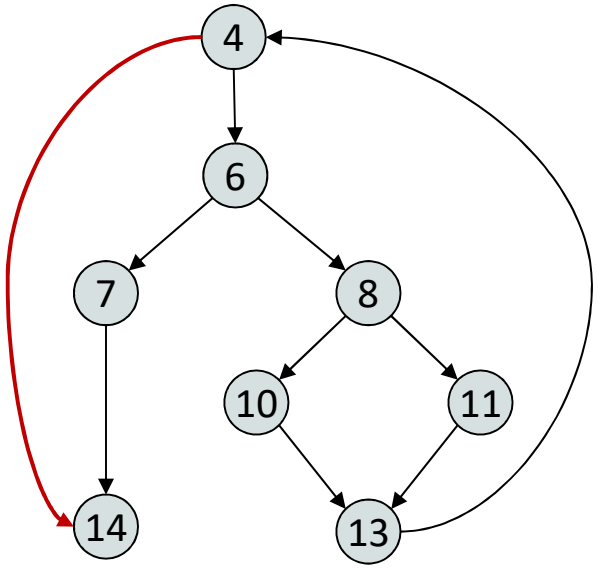
- 基本路径测试的具体步骤
  - ✧ 第三步：准备测试用例
    - ◌ 根据 *McCabe* 环路复杂度的
      路径 (一条独立路径是指，
      引入一个新处理语句或一个
      好等于该程序的独立路径的
    - ◌ 给出一个独立的路径集合如
      - 路径1：4-14
      - 路径2：4-6-7-14
      - 路径3：4-6-8-10-13-4-14
      - 路径4：4-6-8-11-13-4-14
    - ◌ 根据上面的独立路径集合，设计输入用例数据，迫使程序
      分别执行上面4条路径。



例8的控制流图

4.1.8 实例：基本路径测试

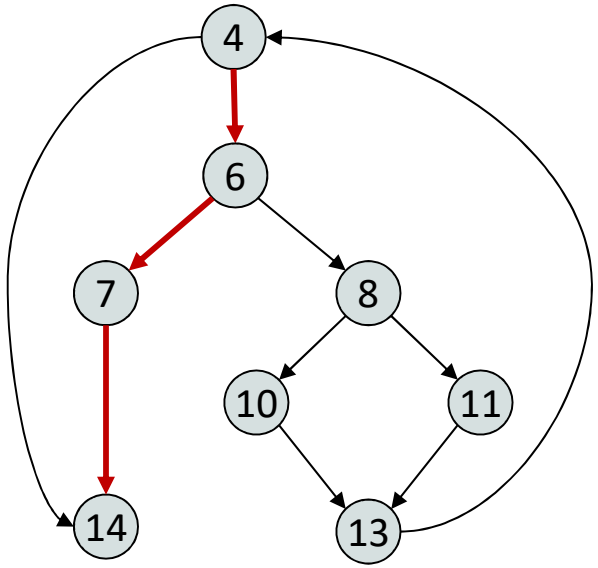
- 基本路径测试的具体步骤
  - ✧ 第四步：应用测试用例。满足上述基本路径集的测试用例是：
    - ◌ 路径1：4-14
      - 输入：iRecordNum=0，或者 iRecordNum<0 的某一个值
      - 预期结果：x=0




```
    void  Sort ( int iRecordNum, int iType )
1.  {
2.      int x = 0;
3.      int y = 0;
4.      while ( iRecordNum - - >= 0 )
5.      {
6.          if ( 0 == iType )
7.          { x = y + 2; break; }
8.          else
9.              if ( 1 == iType )
10.                 x = y + 10;
11.             else
12.                 x = y + 20;
13.         }
14. }
```

4.1.8 实例：基本路径测试

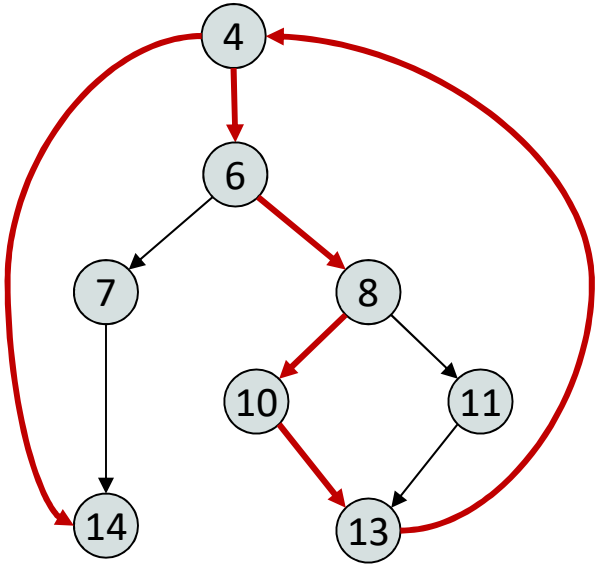
- 基本路径测试的具体步骤
  - ✧ 第四步：应用测试用例。满足上述基本路径集的测试用例是：
    - ◌ 路径2：4-6-7-14
      - 输入数据：iRecordNum=1, iType=0
      - 预期结果：x=2




```
    void  Sort ( int iRecordNum, int iType )
1.  {
2.      int x = 0;
3.      int y = 0;
4.      while ( iRecordNum - - >= 0 )
5.      {
6.          if ( 0 == iType )
7.          { x = y + 2; break; }
8.          else
9.              if ( 1 == iType )
10.                 x = y + 10;
11.             else
12.                 x = y + 20;
13.         }
14. }
```

4.1.8 实例：基本路径测试

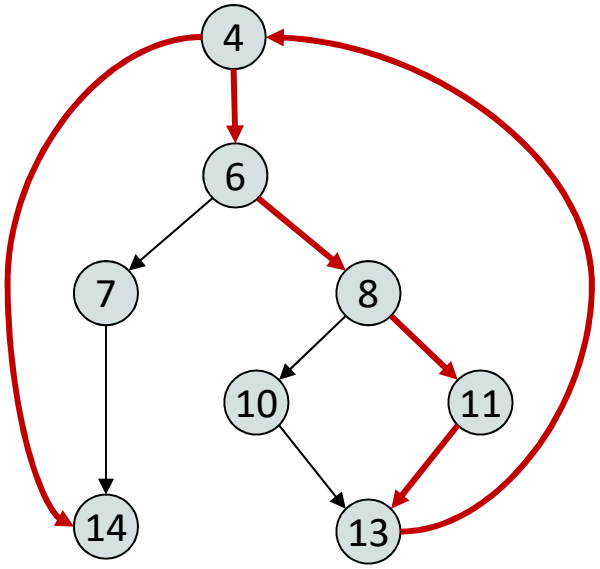
- 基本路径测试的具体步骤
  - ✧ 第四步：应用测试用例。满足上述基本路径集的测试用例是：
    - ◌ 路径3：4-6-8-10-13-4-14
      - 输入数据：iRecordNum=1, iType=1
      - 预期结果：x=10




```
    void  Sort ( int iRecordNum, int iType )
1.  {
2.      int x = 0;
3.      int y = 0;
4.      while ( iRecordNum - - >= 0 )
5.      {
6.          if ( 0 == iType )
7.          { x = y + 2; break; }
8.          else
9.              if ( 1 == iType )
10.                 x = y + 10;
11.             else
12.                 x = y + 20;
13.         }
14. }
```

4.1.8 实例：基本路径测试

- 基本路径测试的具体步骤
  - ✧ 第四步：应用测试用例。满足上述基本路径集的测试用例是：
    - ◌ 路径4：4-6-8-11-13-4-14
      - 输入数据：iRecordNum=1, iType=2
      - 预期结果：x=20



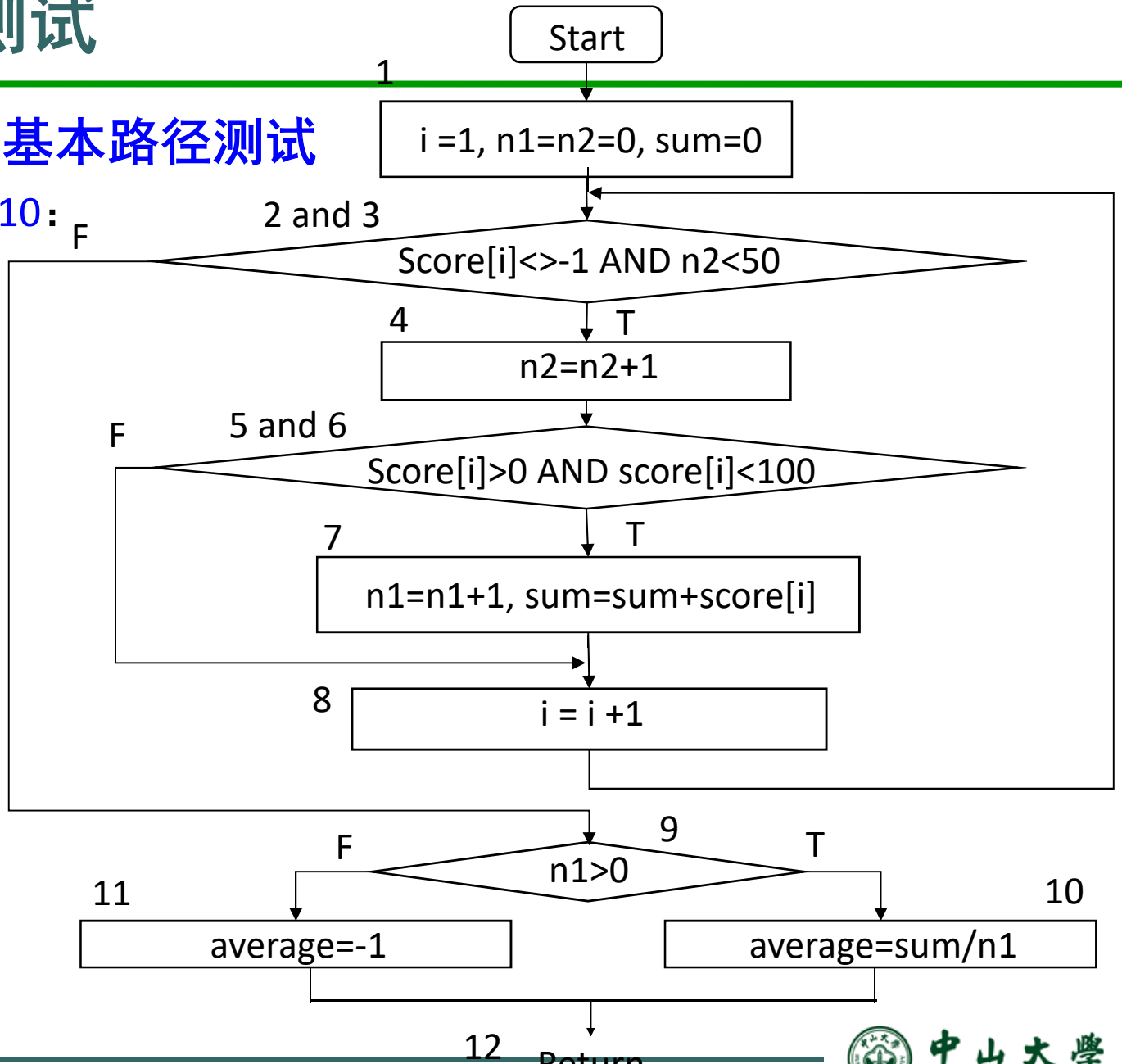

```
    void  Sort ( int iRecordNum, int iType )
1.  {
2.      int x = 0;
3.      int y = 0;
4.      while ( iRecordNum - - >= 0 )
5.      {
6.          if ( 0 == iType )
7.          { x = y + 2; break; }
8.          else
9.              if ( 1 == iType )
10.                 x = y + 10;
11.             else
12.                 x = y + 20;
13.         }
14. }
```

4.1.8 实例：基本路径测试

- 基本路径测试的具体步骤
  - 例10：下例程序流程图描述了最多输入50个值 (以–1作为输入结束标志) 作为学生成绩的分数，计算其中有效的学生成绩 (0-100分) 的个数、总分数和分数平均值的一个程序。数据类型作如下定义：
    - Score[i]: 存储学生成绩的整型数组
    - n2: 总输入数目计数器
    - n1: 有效的输入数目计数器
    - sum: 有效的输入分数累计
    - average: 结束时有效的输入分数平均值

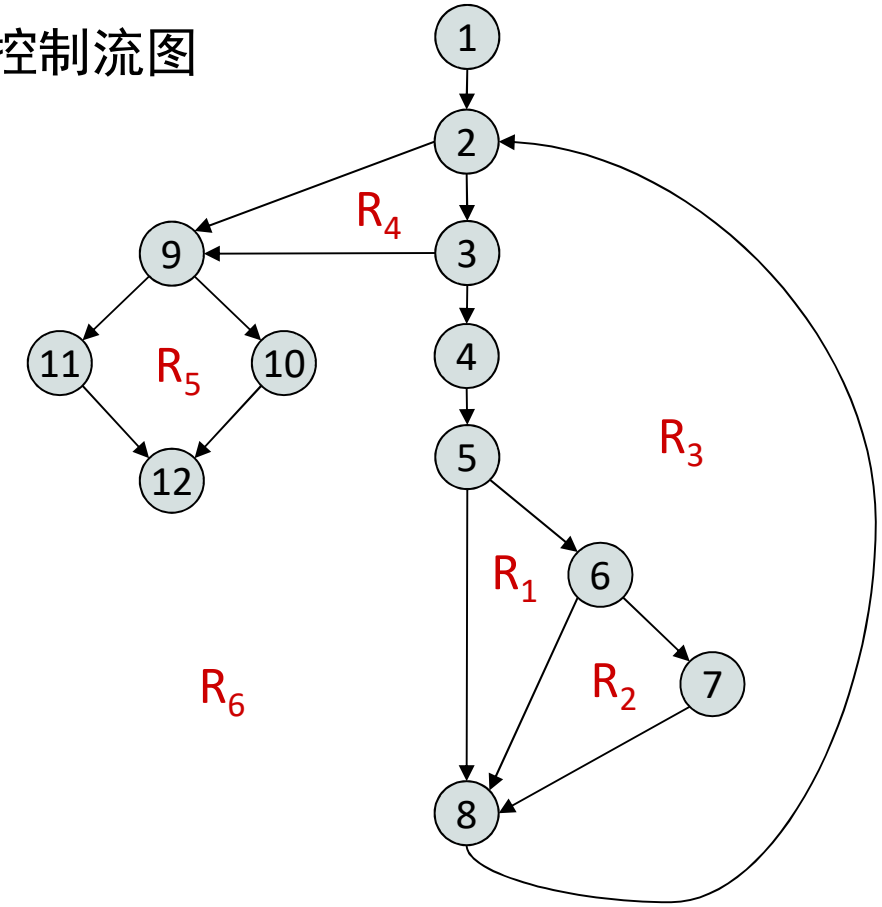
4.1.8 实例：基本路径测试

✧ 例10：



4.1.8 实例：基本路径测试

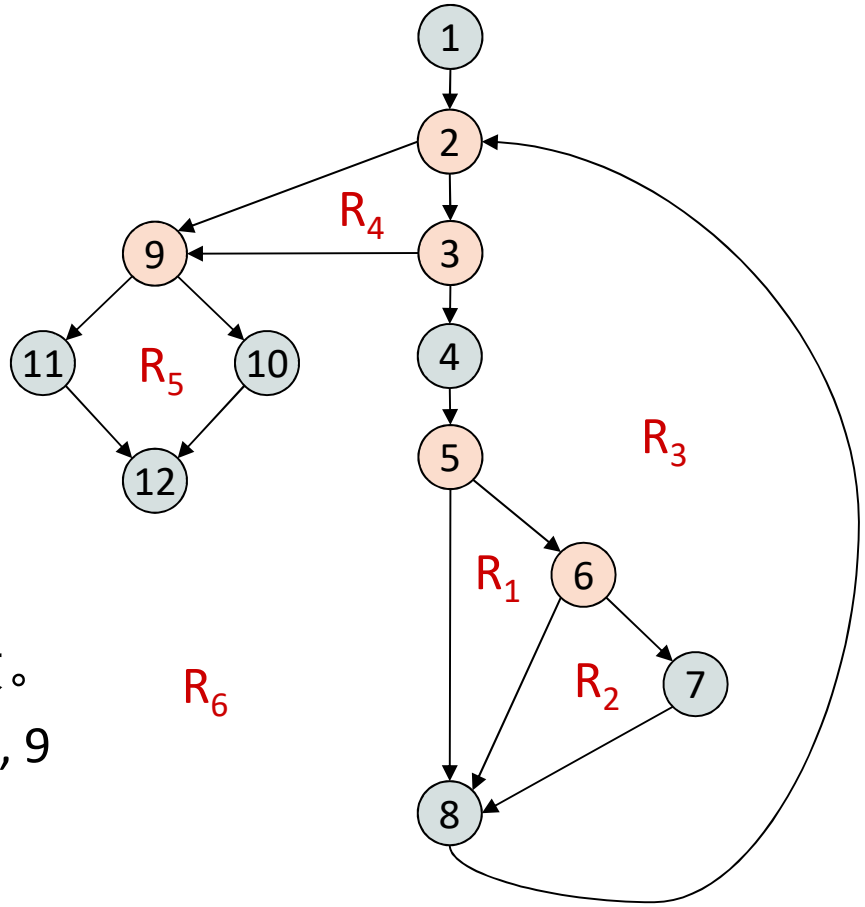
- 解：
  - 画出程序的控制流图



4.1.8 实例：基本路径测试

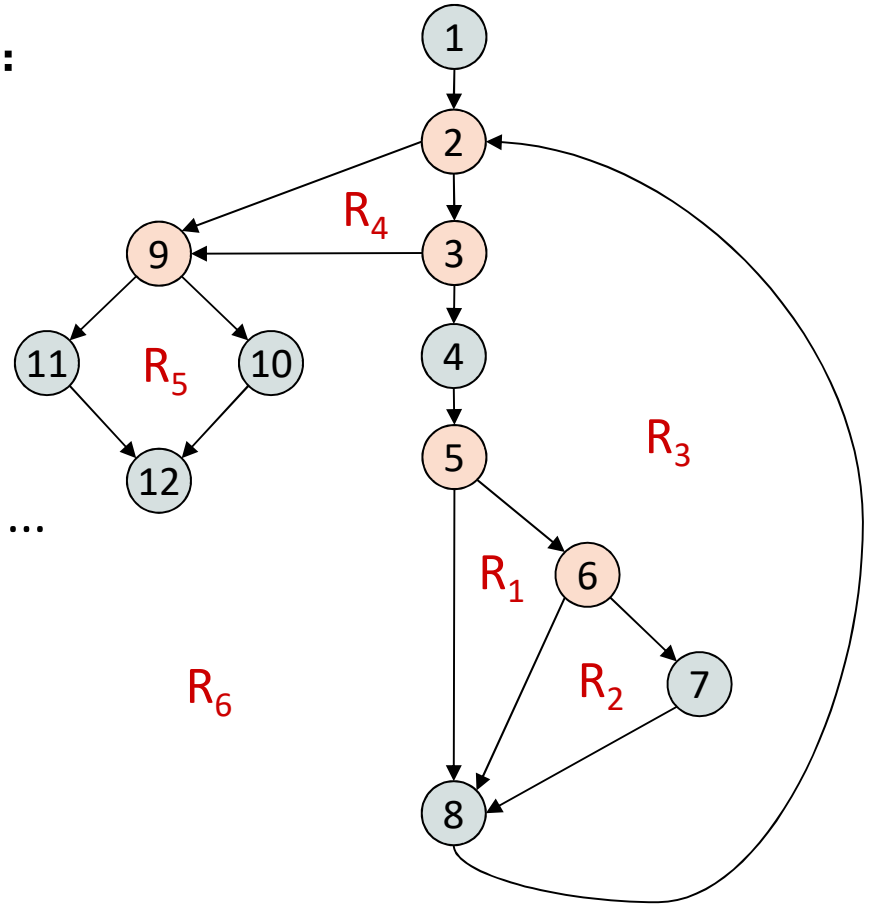
- 解：
  - 确定环路复杂度：
    - V(G) = m – n + 2
      = 16 – 12 + 2
      = 6
    - V(G) = 6 (区域数)
    - V(G) = d + 1
      = 5 + 1
      = 6

    d 为谓词结点的个数。
  - 控制流图中结点 2, 3, 5, 6, 9 是谓词结点。



4.1.8 实例：基本路径测试

- 解：
  - 确定6条独立的基本路径：
  - 路径1：1-2-9-10-12
  - 路径2：1-2-9-11-12
  - 路径3：1-2-3-9-10-12
  - 路径4：1-2-3-4-5-8-2 ...
  - 路径5：1-2-3-4-5-6-8-2 ...
  - 路径6：1-2-3-4-5-6-7-8-2 ...



4.1.8 实例：基本路径测试

- 解：
  - 为每一条独立路径各设计一组测试用例，以便强迫程序沿着该路径至少执行一次。

    (1) 路径1 (1-2-9-10-12) 的测试用例：
    - score[k]=有效分数值，当 k<i；
    - score[i]=-1, $2 \le i \le 50$；
    - 期望结果：根据输入的有效分数算出正确的分数个数 n1、总分 sum 和平均分 average。

    (2) 路径2 (1-2-9-11-12) 的测试用例：
    - score[1]=-1；
    - 期望结果：average = -1，其他量保持初值。

4.1.8 实例：基本路径测试

- 解：
  - 为每一条独立路径各设计一组测试用例，以便强迫程序沿着该路径至少执行一次。

    (3) 路径3 (1-2-3-9-10-12) 的测试用例：
    - 输入多于50个有效分数，即试图处理51个分数，要求前51个为有效分数；
    - 期望结果：n1=50，且算出正确的总分 sum 和平均分 average。

    (4) 路径4 (1-2-3-4-5-8-2 …) 的测试用例：
    - score[i]=有效分数，当 i<50；
    - score[k]<0，k<i；
    - 期望结果：根据输入的有效分数算出正确的分数个数 n1、总分 sum 和平均分 average。

4.1.8 实例：基本路径测试

- 解：
  - 为每一条独立路径各设计一组测试用例，以便强迫程序沿着该路径至少执行一次。

    (5) 路径5 (1-2-3-4-5-6-8-2 ...) 的测试用例：
    - score[i]=有效分数， 当 i<50；
    - score[k]>100， k<i；
    - 期望结果：根据输入的有效分数算出正确的分数个数 n1、总分 sum 和平均分 average。

    (6) 路径6 (1-2-3-4-5-6-7-8-2 ...) 的测试用例：
    - score[i]=有效分数， 当 i<50；
    - 期望结果：根据输入的有效分数算出正确的分数个数 n1、总分 sum 和平均分 average。

4.1 白盒测试

4.1.8 实例：基本路径测试

- 小结：
  - 白盒测试方法是在测试人员全面了解程序内部逻辑结构的基础上，对程序的所有逻辑路径进行测试，是穷举路径测试。但是贯穿程序的独立路径数目庞大，而且即使每条路径都测试了仍然可能有错误。
    - 穷举路径测试无法查出程序违反了设计规范，即程序本身是个错误的程序；
    - 穷举路径测试无法查出程序中因遗漏路径而发生的错误；
    - 穷举路径测试可能发现不了一些与数据相关的错误。

Thank you!