软件质量保证与测试

第4章 白盒测试

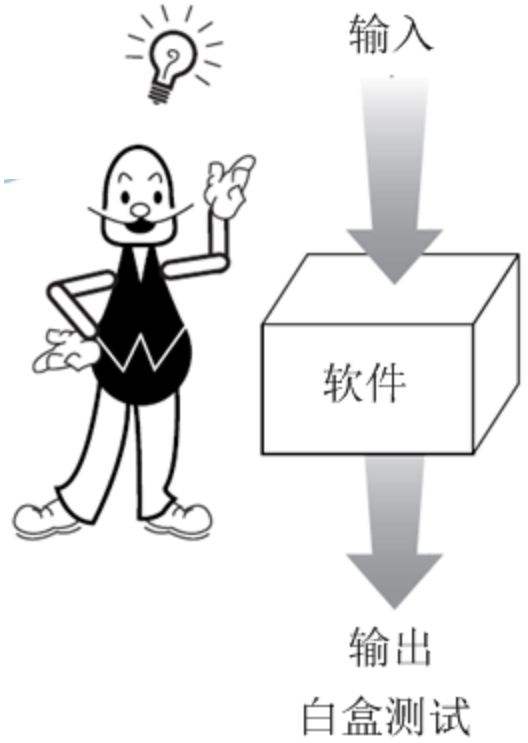
内容提要

- 4.1 白盒测试概述
- 4.2 控制流测试
- 4.3 数据流测试
- 4.4 程序插装
- 4.5 程序变异测试
- 4.6 小结

4.1 白盒测试概述

- 白盒测试：利用程序内部的逻辑结构及有关信息，根据不同的测试需求，结合不同的测试对象，使用合适的方法设计或选择测试用例，进行测试

- 白盒测试发现、定位和解决问题不仅是彻底的，也是直接的，效率很高。



白盒测试概述(续)

- 源代码

  阅读源代码，检验代码的规范性，并对照函数功能查找代码的逻辑缺陷、内存管理缺陷、数据定义和使用缺陷等

- 程序结构

  使用与程序设计相关的图表，找到程序设计的缺陷，或评价程序的执行效率

白盒测试概述(续)

- **白盒测试方法**

静态测试

- ✓ 代码评审
- ✓ 控制流分析
- ✓ 数据流分析
- ✓ 信息流分析

动态测试

- ✓ 语句覆盖
- ✓ 分支覆盖
- ✓ 条件覆盖
- ✓ 分支-条件覆盖
- ✓ 条件组合覆盖
- ✓ 路径覆盖
- ✓ 数据流测试
- ✓ 插桩
- ✓ 变异测试

白盒测试概述(续)

- **优势：**
  - 针对性强，便于快速定位缺陷
  - 在函数级别开始测试工作，缺陷修复的成本低
  - 有助于了解测试的覆盖程度
  - 有助于代码优化和缺陷预防

白盒测试概述(续)

- 不足：
  - 对测试人员要求高

    测试人员需要具备一定的编程经验

    测试人员需要具备广博的知识面
  - 成本高

    白盒测试准备时间较长

白盒测试概述(续)

穷尽路径测试可能吗？

- 软件测试是不完备的
- 软件测试是有风险的
- 测试设计应达到的目标
  - 提高效率
  - 降低风险

内容提要

- 4.1 白盒测试概述
- 4.2 控制流测试
- 4.3 数据流测试
- 4.4 程序插装
- 4.5 程序变异测试
- 4.6 小结

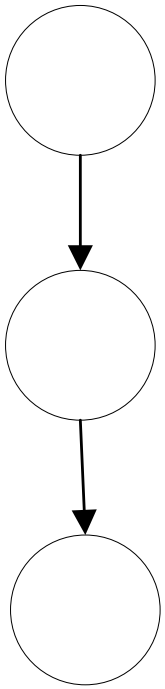
4.2 控制流测试

- 什么因素导致程序结构变得复杂？
- 如何衡量程序结构的复杂程度？
- 控制程序执行流程发生变化的主要因素是什么？
- 如何测试这些因素，并确保测试的效率？

控制流测试（续）

线性结构

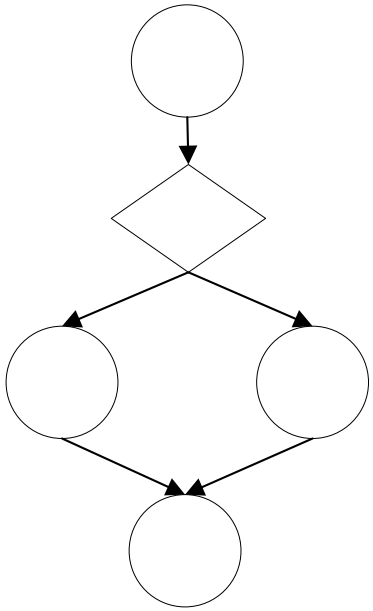
```
int Func1(int a)
{
        int b;
        b = a + 1;
        return b;
}
```



控制流测试（续）

条件判定结构

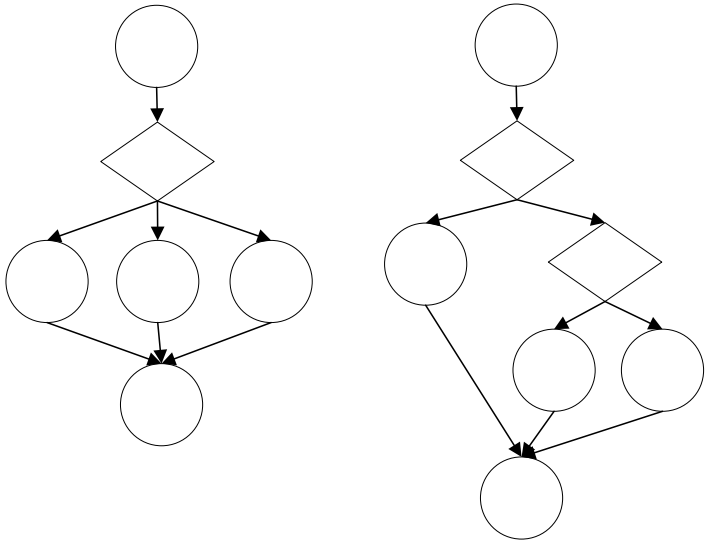
```
int Func2(int a)
{
      int b = 0;
      if(a > 1)
        b = a + 1;
      else
        b = a – 1;
      return b;
}
```



控制流测试（续）

条件判定结构

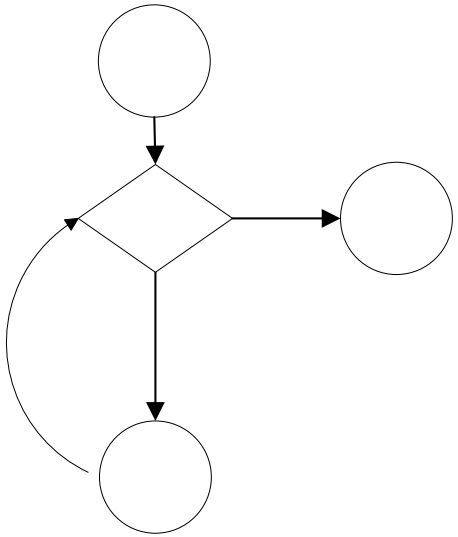
```
int Func3(int a)
{
      int b = 0;
      switch(a){
         case 0: b = a; break;
         case 1: b = a * 2; break;
         case 2: b = a * 3; break;
         default: break;
       }
       return b;
}
```



控制流测试（续）

while-do循环结构

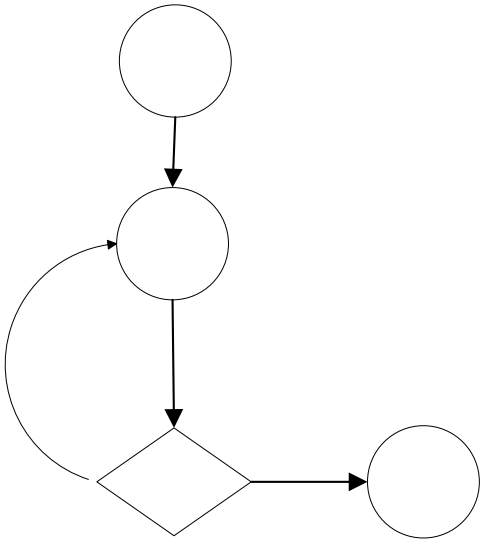
```
int Func4(int a)
{
      int b = 0;
      int i =1;
      while(i < 10){
        b = b + a * i;
        i ++;
       }
       return b;
}
```



控制流测试（续）

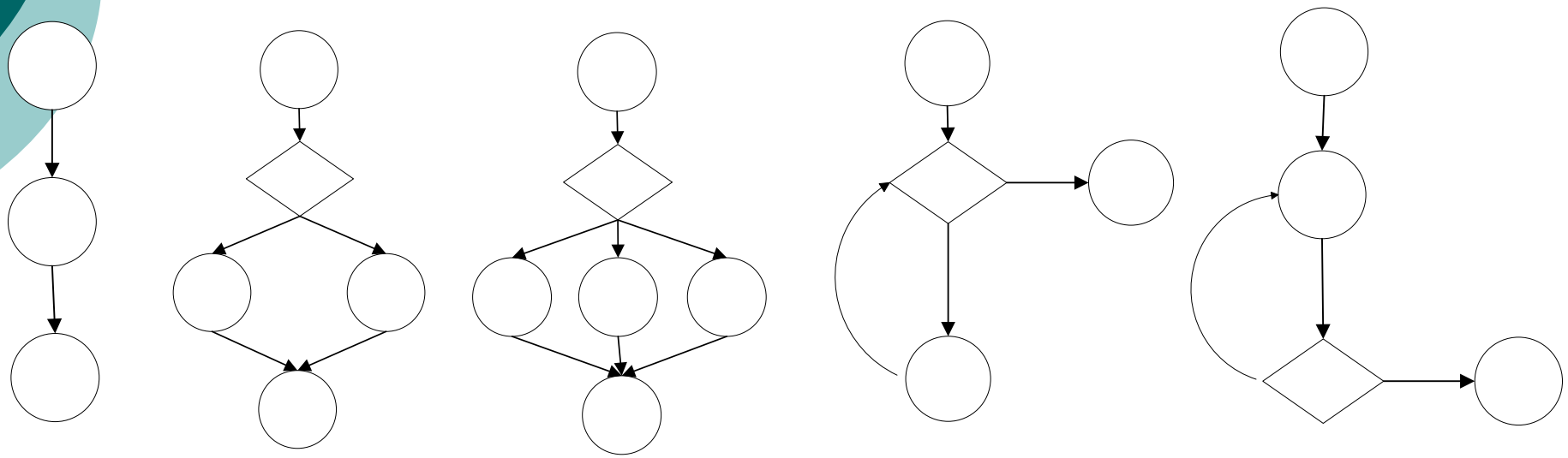
do-while循环结构

```
int Func5(int a)
{
      int b = 0;
      int i =1;
      do{
        b = b + a * i;
        i ++;
       }while(i < 10)
       return b;
}
```



控制流测试（续）

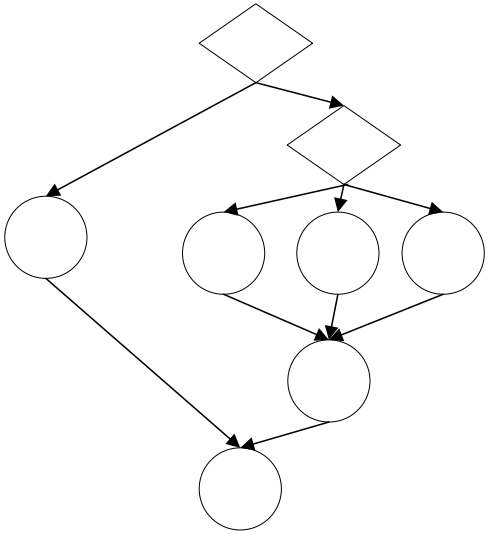
判定节点导致结构复杂



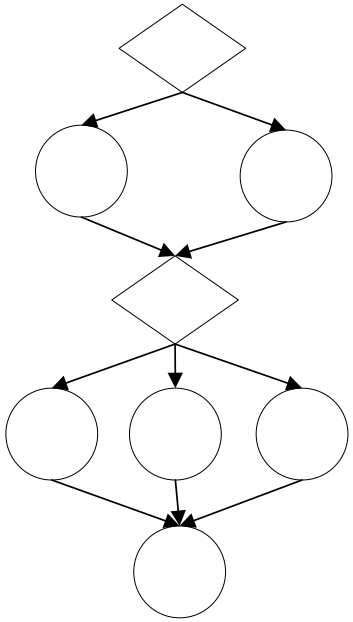
风险：小 → **大**

控制流测试（续）

嵌套



串联



控制流测试（续）

- 什么因素导致程序结构变得复杂？
  判定节点
- 控制程序执行流程发生变化的主要因素是什么？
  判定节点
- 如何衡量程序结构的复杂程度？
- 如何测试这些因素，并确保测试的效率？

控制流测试（续）

- 关注判定节点固有的复杂性
  - 焦点：判定表达式
  - 方法：逻辑覆盖测试
- 关注判定结构与循环结构对执行路径产生的影响
  - 焦点：路径
  - 方法：独立路径测试
- 关注循环结构本身的复杂性
  - 焦点：循环体
  - 方法：基于数据的静态分析

控制流测试（续）

逻辑覆盖：对判定的测试

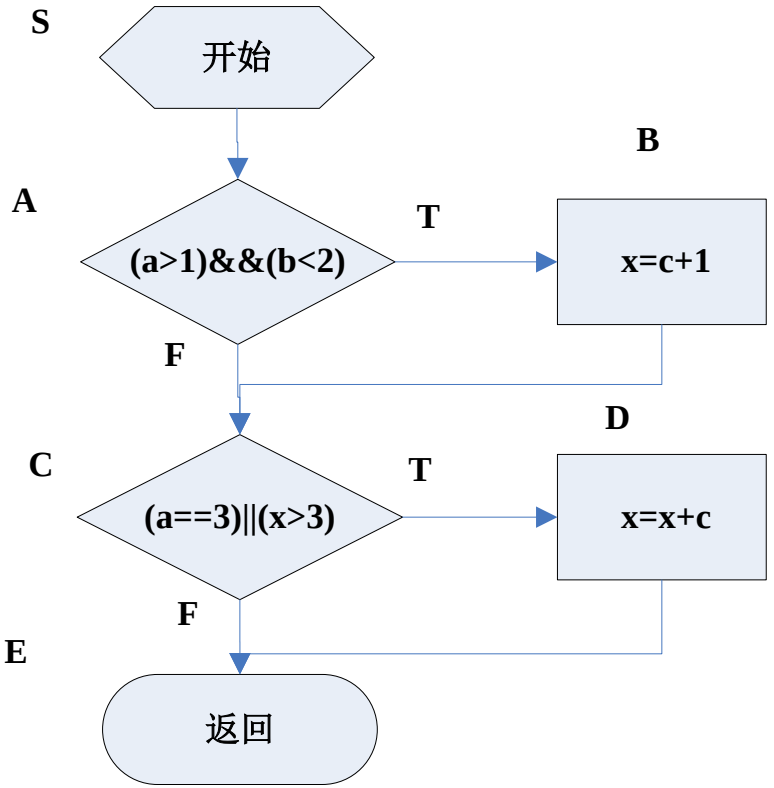
- 关注点：判定表达式本身的复杂度
- 原理：通过对程序逻辑结构的遍历，来实现测试对程序的覆盖
- 原则：对程序代码中所有的逻辑值，都需要测试真值(True)和假值(False)的情况

控制流测试（续）

- 对于选定的覆盖目标，如何控制测试用例规模，提高测试用例典型性，避免测试漏洞？
- 如何选择合适的覆盖指标？

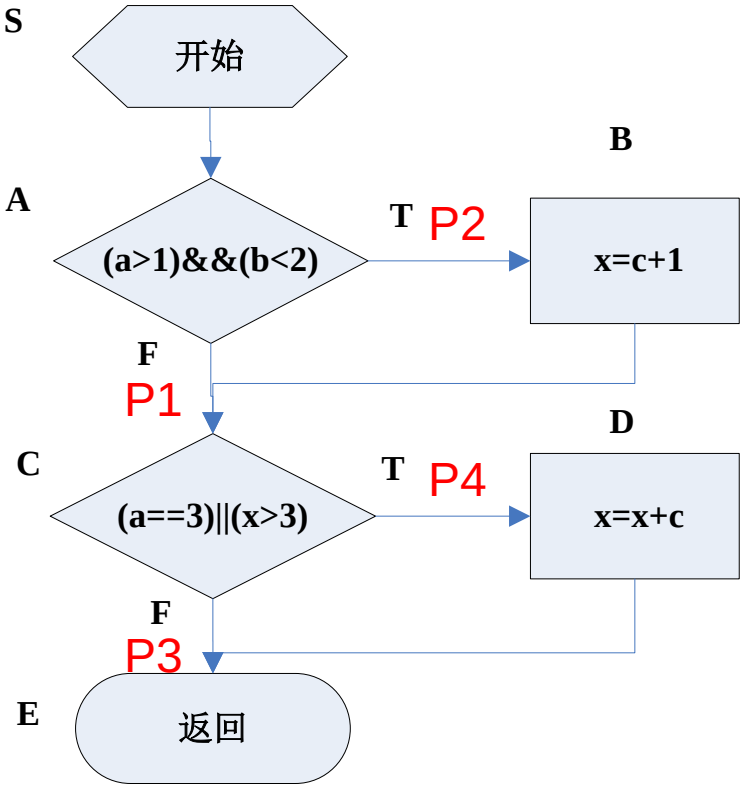
控制流测试(续)

```
S  int Func6(int a, int b, int c,
int x)
   {
A     if ((a > 1) && (b < 2))
B         x = c + 1;
C    if ((a == 3) || (x > 3))
D        x = x + c;
E    return x;
}
```



控制流测试(续)

- 四个基本逻辑判定条件
  - T1：a>1
  - T2：b<2
  - T3：a==3
  - T4：x>3
- 4条执行路径
  - L13: SACE (P1->P3)
  - L14: SACDE (P1->P4)
  - L23: SABCE (P2->P3)
  - L24: SABCDE (P2->P4)



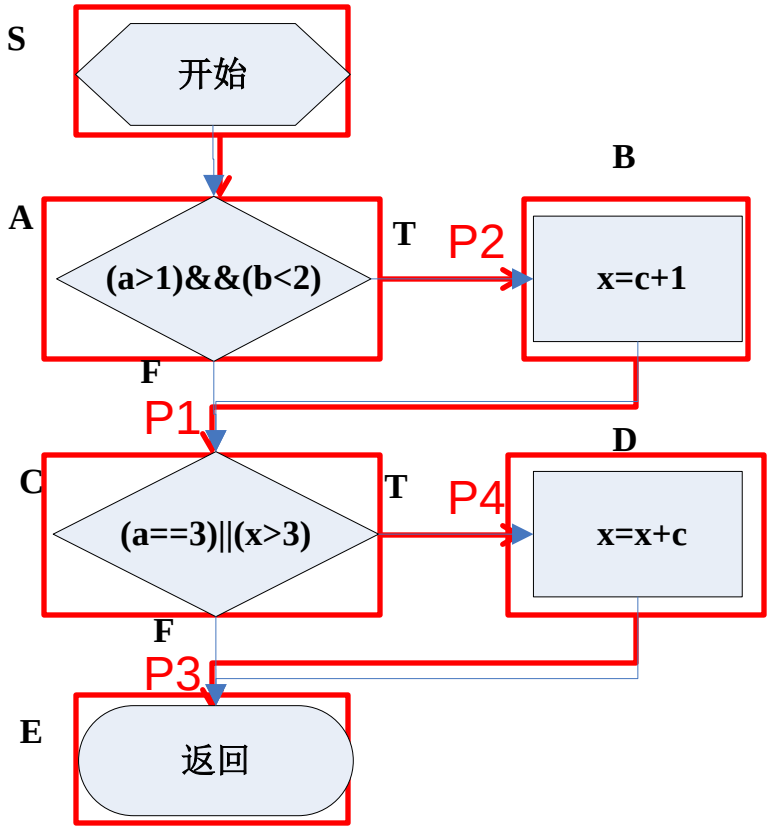
控制流测试（续）

- 语句覆盖
- 判定覆盖
- 条件覆盖
- 判定-条件覆盖
- 条件组合覆盖
- 修正的判定-条件覆盖

语句覆盖

- 基本思想：设计若干个测试用例，然后运行被测程序，使程序中的**每个可执行语句至少执行一次**。
- 点覆盖

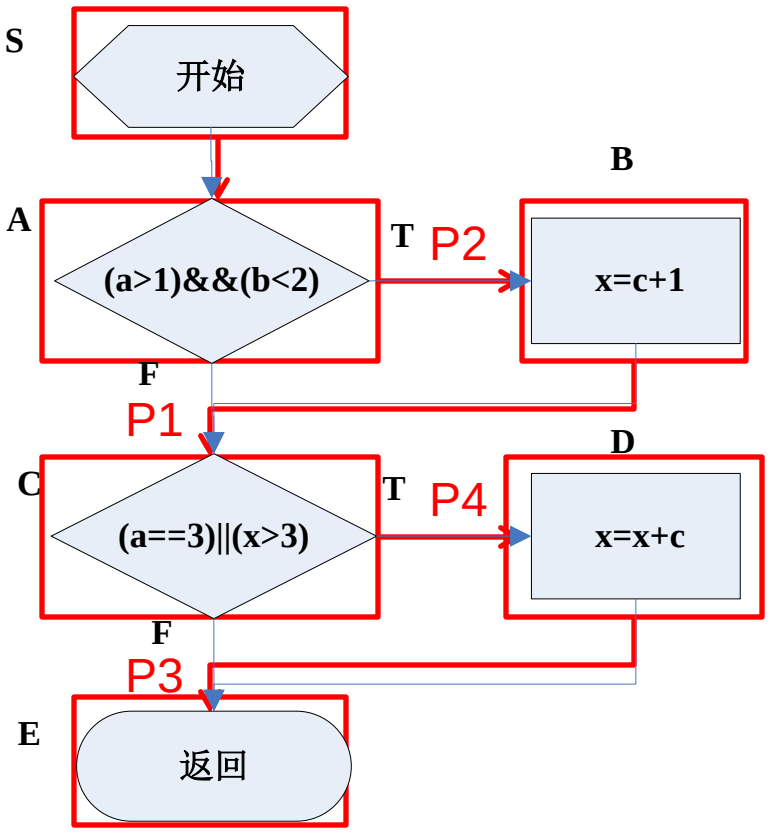
如何设计测试用例？



语句覆盖（续）

输入条件	测试用例	测试用例
T1:a>1	T	T
T2:b<2	T	T
T3:a==3	T	T
T4:x>3	T	F
(a>1)&&(b<2)	T	T
(a==3)||(x>3)	T	T
覆盖率	100%	100%

测试用例	输入				预期输出
	a	b	c	x	
TC1	3	1	3	0	7
TC2	3	1	2	0	5

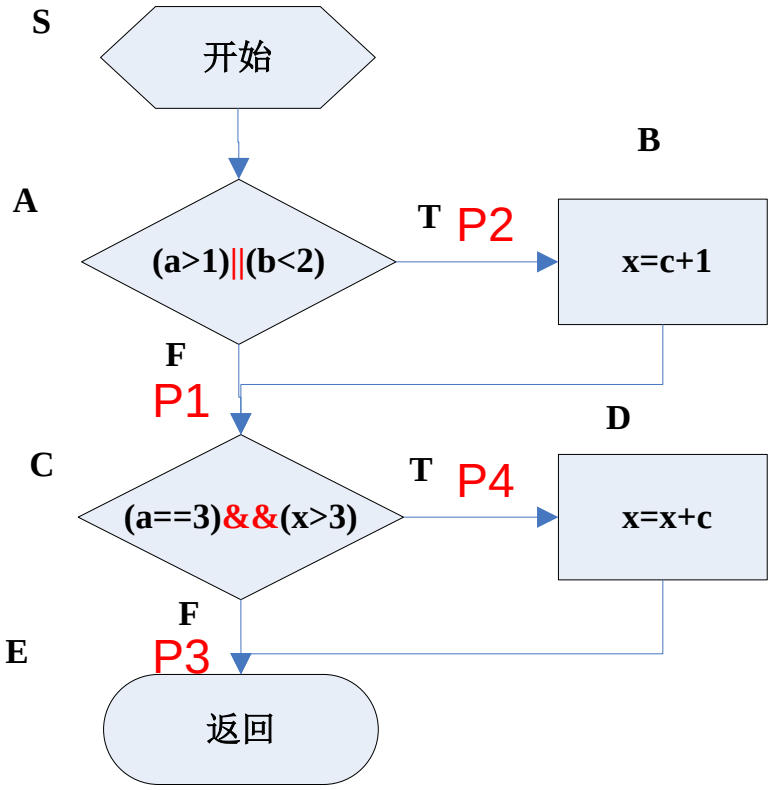


语句覆盖（续）

测试用例	输入				预期输出
	a	b	c	x	
TC1	3	1	3	0	7
TC2	3	1	2	0	5

存在缺陷的程序

- **TC1:不能发现缺陷**
- **TC2:可以部分发现缺陷**



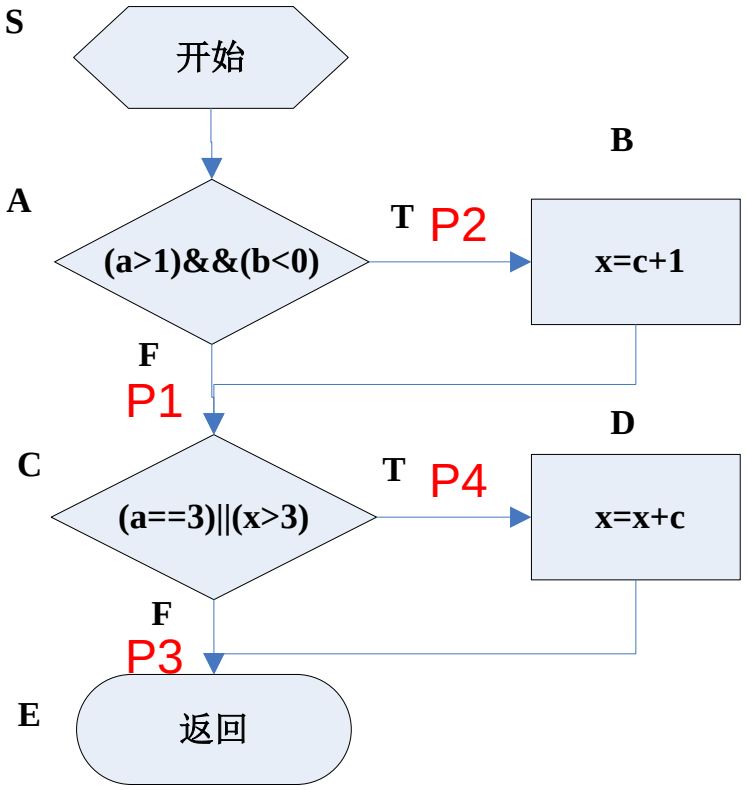
语句覆盖(续)

语句覆盖是最弱的覆盖准则

- **局限性**
  - 关注语句，而非关注判定结点
  - 对隐式分支无效
- **对策**
  - 优选测试数据
  - 更强的覆盖准则：判定覆盖

判定覆盖

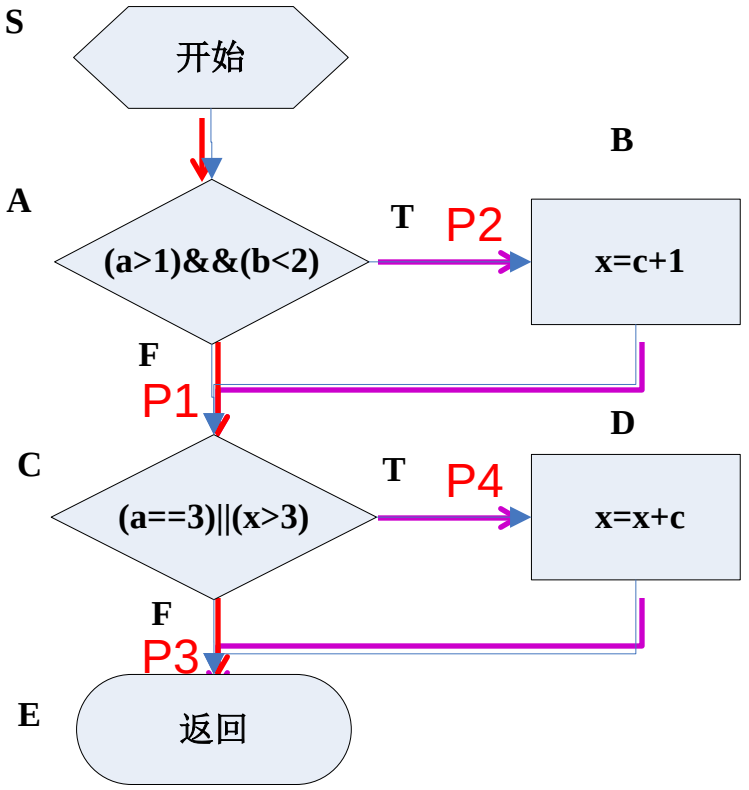
- 基本思想：设计若干测试用例，运行被测程序，使得程序中每个判断的取真分支和取假分支至少经历一次，即判断的真假值均曾被满足。
- 边覆盖



判定覆盖（续）

测试用例设计满足判定覆盖

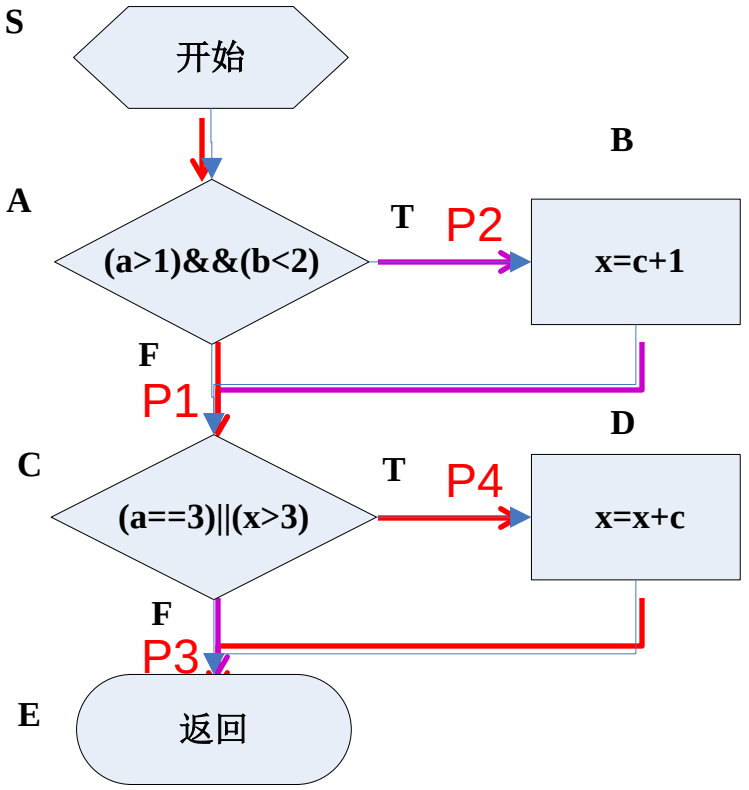
- **用例需执行路径L13**
- **并执行路径L24**



判定覆盖（续）

测试用例设计满足判定覆盖

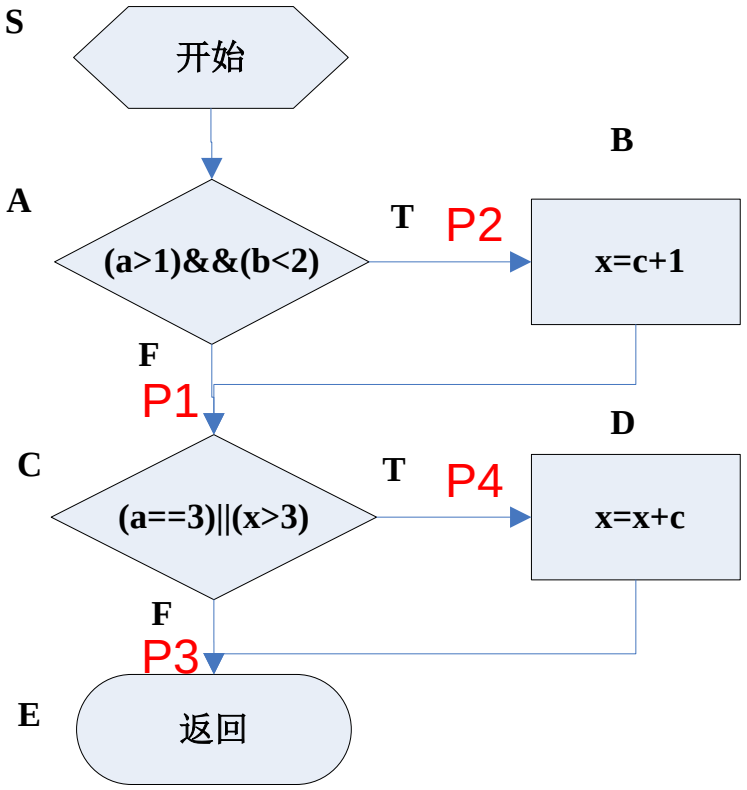
- **用例需执行路径L14**
- **并执行路径L23**



判定覆盖（续）

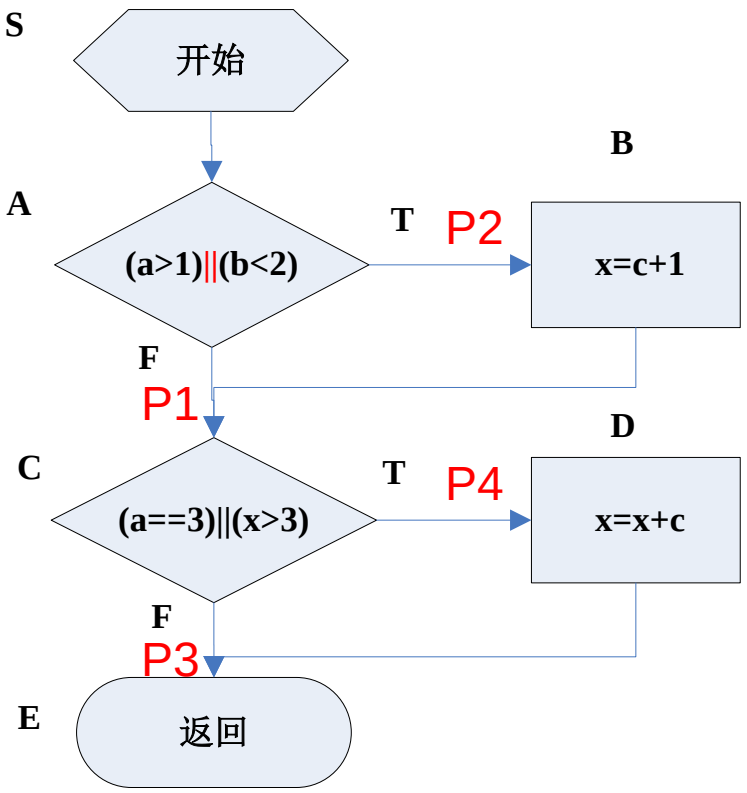
测试用例设计

测试用例	输入				预期输出	通过路径	判定覆盖	语句覆盖
	a	b	c	x				
TC3	2	1	3	0	7	L24	100%	100%
TC4	4	2	2	0	0	L13		
TC5	1	2	2	4	6	L14	100%	100%
TC6	2	1	1	2	2	L23		



判定覆盖（续）

测试用例	输入				预期输出	通过路径	判定覆盖	语句覆盖
	a	b	c	x				
TC3	2	1	3	0	7	L24	100%	100%
TC4	4	2	2	0	0	L13		
TC5	1	2	2	4	6	L14	100%	100%
TC6	2	1	1	2	2	L23		



存在缺陷的程序

- **TC3＋TC4:可以发现缺陷**
- **TC5＋TC6:不能发现缺陷**

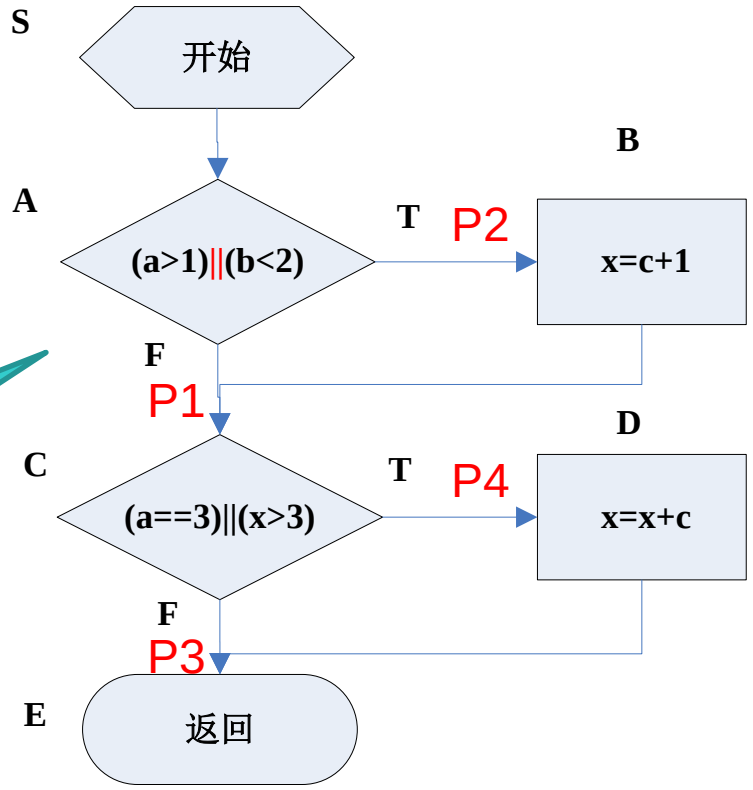
判定覆盖(续)

- **优点**
  - 分支覆盖是比语句覆盖更强的测试能力：实现了分支覆盖，就实现了语句覆盖
  - 它无需细分每个判定就可以得到测试用例。
- **缺点**
  - 往往大部分的判定语句是由多个逻辑条件组合而成，若仅仅判断其最终结果，不能判断条件错误，不能判断逻辑错误
  - 仍是弱的逻辑覆盖

条件覆盖

- **基本思想：设计若干测试用例，运行被测程序，使得每个判断中每个条件的可能取值至少满足一次。**

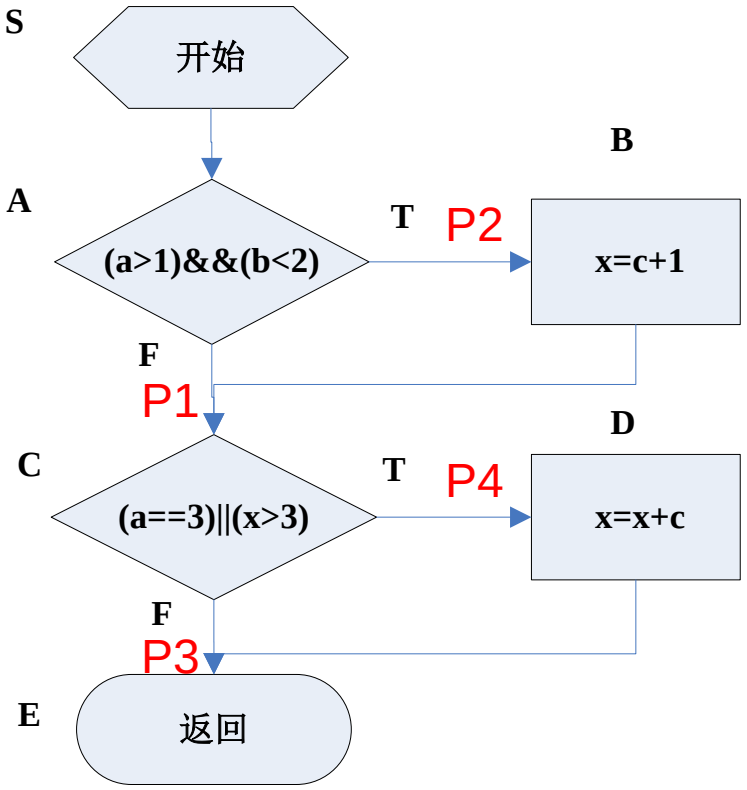
条件覆盖一定能满足判定覆盖吗？



条件覆盖（续）

条件	取值	取值
T1:a>1	T	F
T2:b<2	F	T
T3:a==3	T	F
T4:x>3	F	T

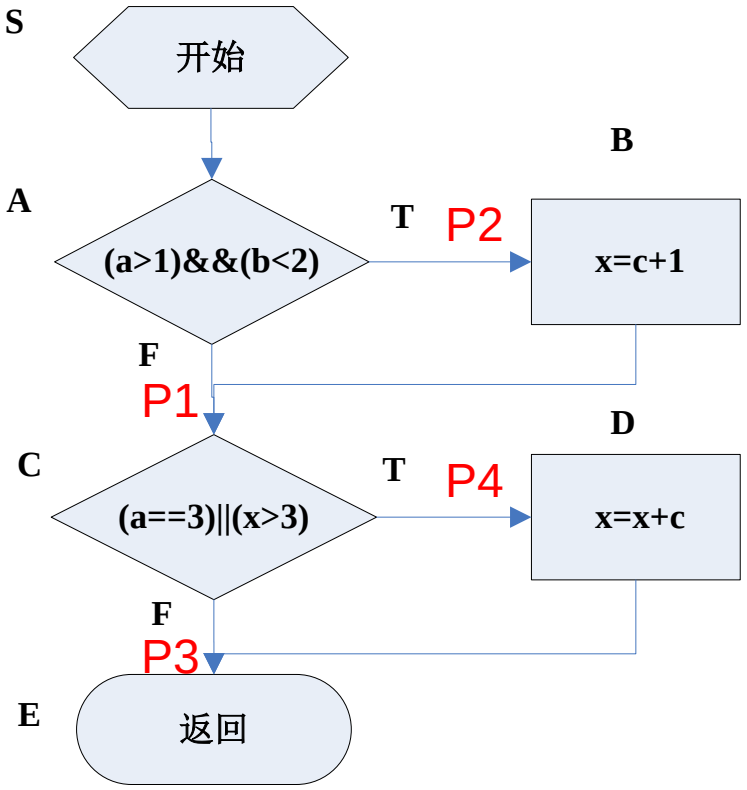
测试用例	输入				预期输出	通过路径	条件覆盖	判定覆盖
	a	b	c	x				
TC7	3	2	1	0	1	L14	100%	50%
TC8	1	1	1	4	5	L14		



条件覆盖（续）

条件	取值	取值
T1:a>1	T	F
T2:b<2	T	F
T3:a==3	T	F
T4:x>3	T	F

测试用例	输入				预期输出	通过路径	条件覆盖	判定覆盖
	a	b	c	x				
TC9	3	1	3	0	7	L24	100%	100%
TC10	1	2	2	0	0	L13		

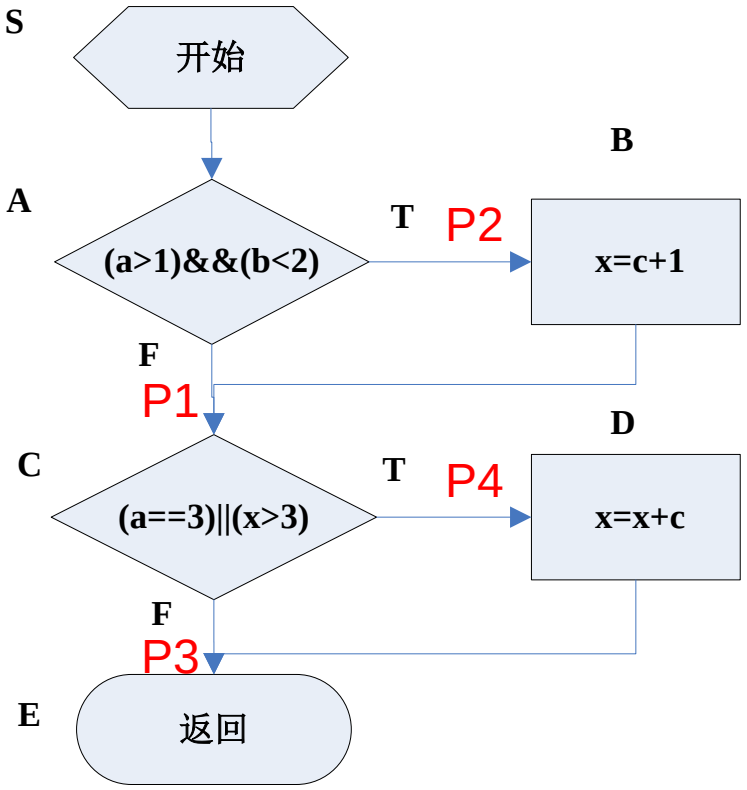


条件覆盖(续)

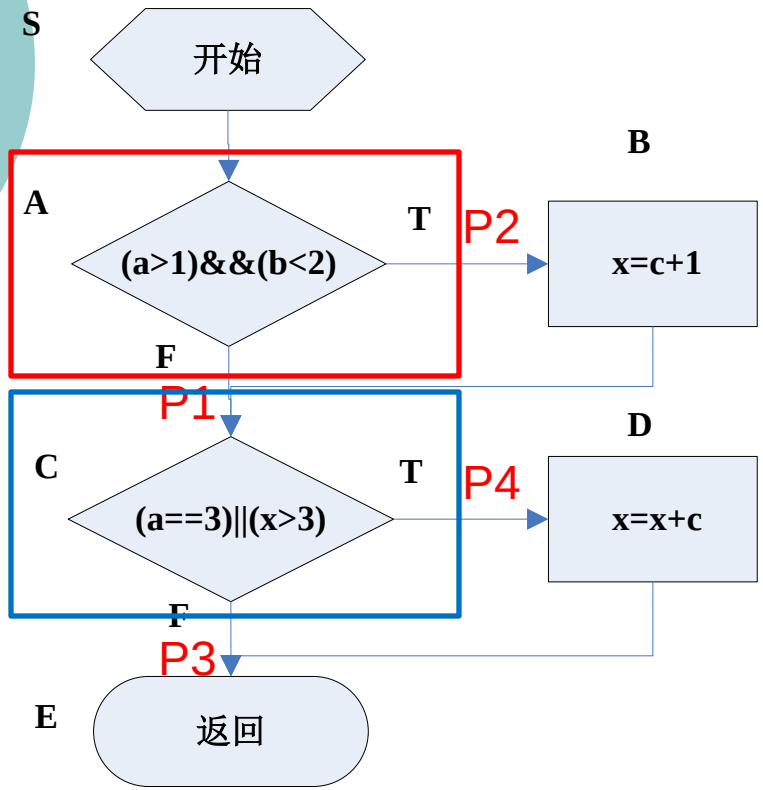
- **优点**
  - 能够检查所有的条件错误
- **缺点**
  - 不能保证判定覆盖

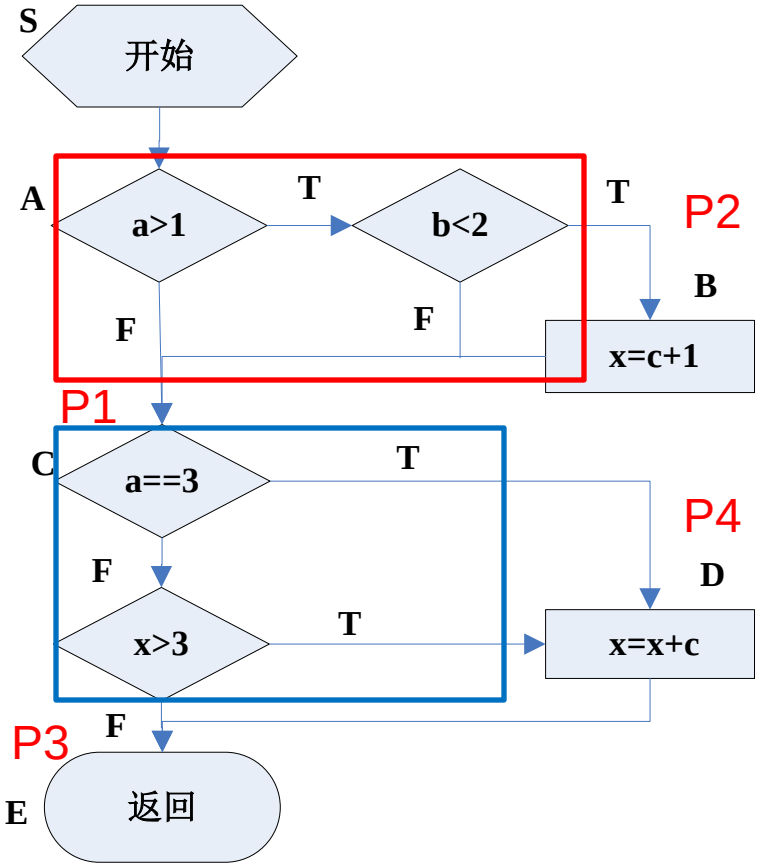
判定-条件覆盖

- **基本思想：设计若干测试用例，运行被测程序，使得**
  - 判断中每个条件的所有可能至少出现一次
  - 每个判断本身的判定结果也至少出现一次
- **判定覆盖+条件覆盖**



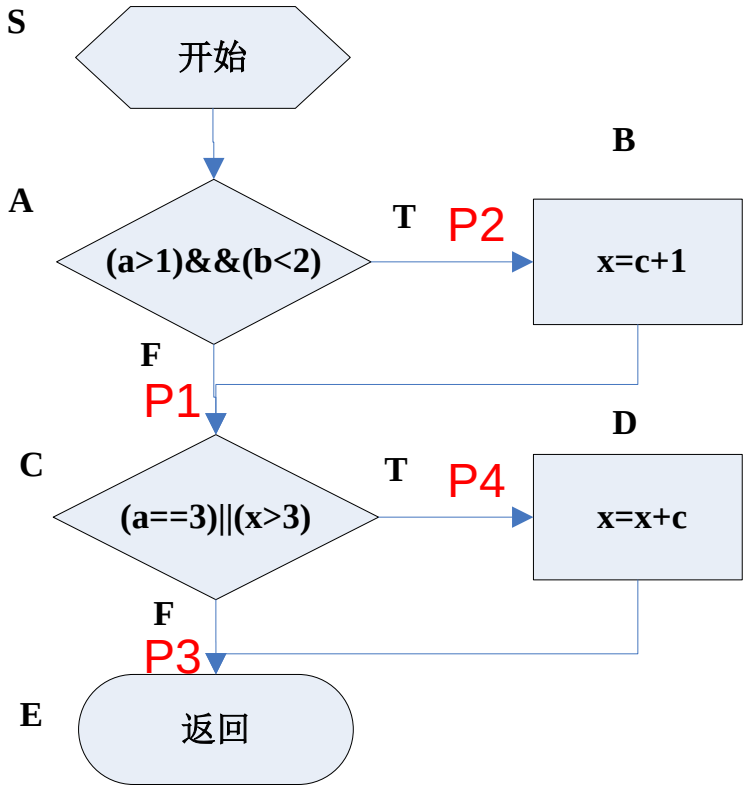
判定-条件覆盖（续）


S
开始
A
(a>1)&&(b<2)
T
P2
B
x=c+1
F
P1
C
(a==3)||(x>3)
T
P4
D
x=x+c
F
P3
E
返回


S
开始
A
a>1
T
b<2
T
P2
B
x=c+1
F
F
P1
C
a==3
T
P4
F
D
x>3
T
x=x+c
F
P3
E
返回

条件组合覆盖

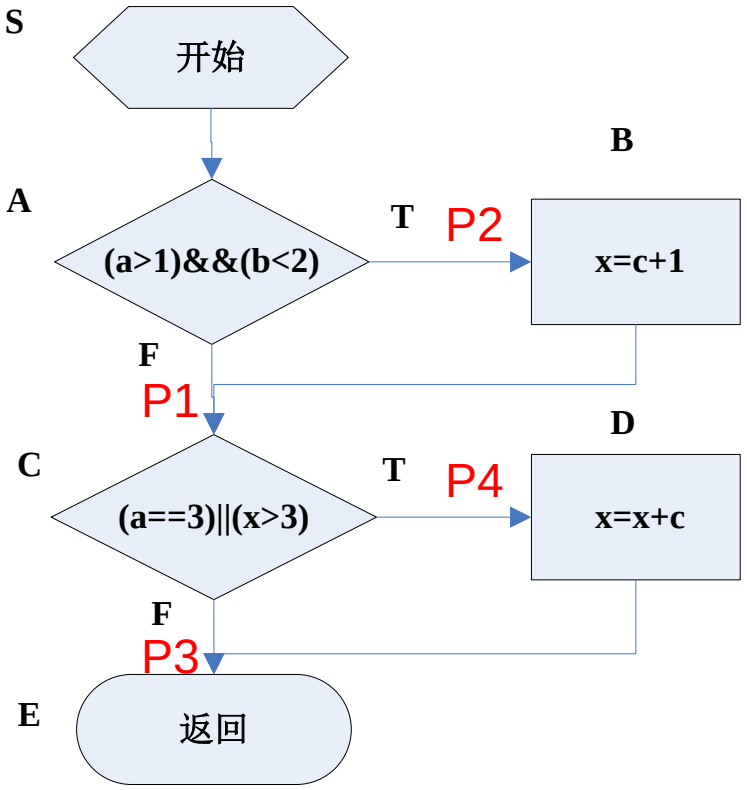
- 基本思想：设计若干测试用例，运行被测程序，使得所有可能的条件取值组合至少执行一次
- 本质：真值表



条件组合覆盖（续）

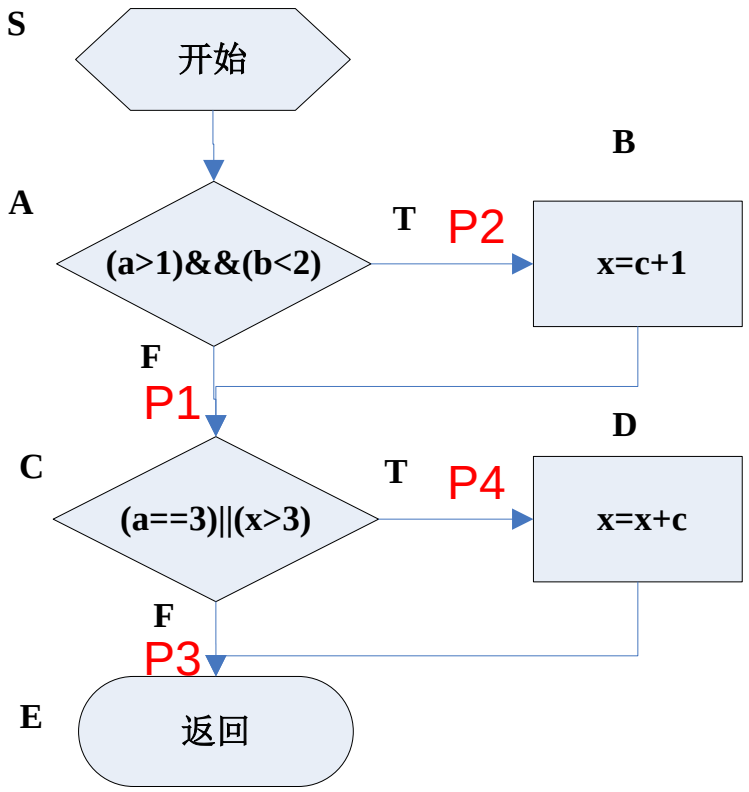
	简单判定条件	
取值	T1:a>1	T2:b<2
	T	T
	T	F
	F	T
	F	F

	简单判定条件	
取值	T3:a==3	T4:x>3
	T	T
	T	F
	F	T
	F	F



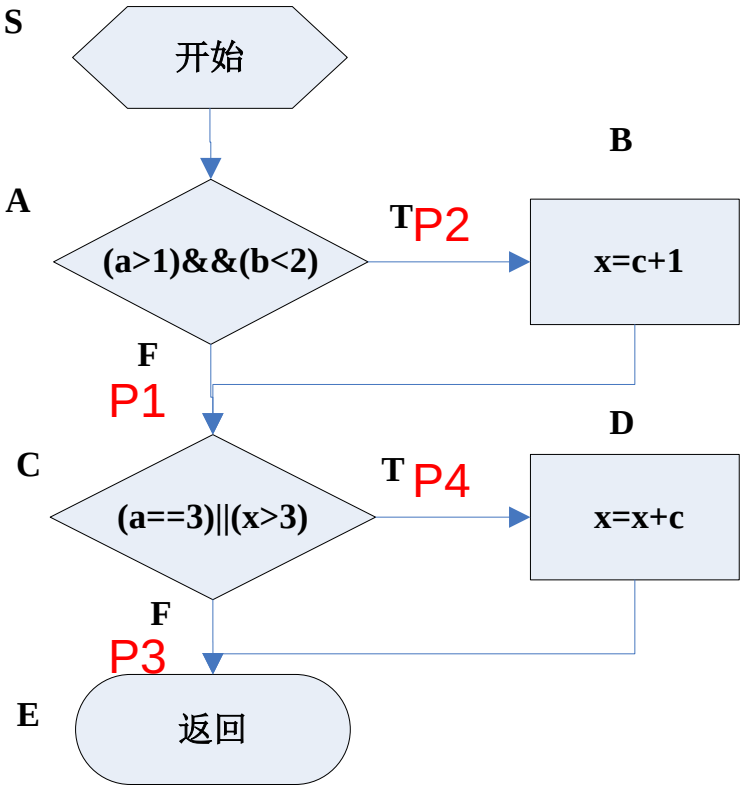
条件组合覆盖(续)

序号	输入				判定条件				路径
	a	b	c	x	T1	T2	T3	T4	
1	3	1	3	0	T	T	T	T	L24
2	3	1	2	0	T	T	T	F	L24
3	2	1	3	0	T	T	F	T	L24
4	2	1	2	0	T	T	F	F	L23
5	3	2	3	4	T	F	T	T	L14
6	3	2	1	3	T	F	T	F	L14
7	2	1	1	4	T	F	F	T	L14
8	2	2	1	3	T	F	F	F	L13



条件组合覆盖(续)

序号	输入				判定条件				路径	备注
	a	b	c	x	T1	T2	T3	T4		
					F	T	T	T		不可能
					F	T	T	F		不可能
9	1	1	1	4	F	T	F	T	L14	
10	2	2	1	3	F	F	F	F	L13	
					F	F	T	T		不可能
					F	F	T	F		不可能
11	1	2	1	4	F	F	F	T	L14	
12	1	2	1	3	F	F	F	F	L13	



条件组合覆盖(续)

- **优点**
  - 满足判定覆盖
  - 满足条件覆盖
  - 满足判定-条件覆盖
  - 方法简单
- **缺点**
  - 测试用例太多，冗余严重

修正的判定-条件覆盖

- 在满足判定-条件覆盖的基础上，每个简单判定条件都应独立地影响到整个判定表达式的取值。
- 判定覆盖+条件覆盖+独立影响性
- 实质：利用简单判定条件的独立影响性来消除测试用例的冗余。

修正的判定-条件覆盖（续）

- 判定表达式：A && B

取值	组合1	组合2	组合3	组合4
A	T	T	F	F
B	T	F	T	F
A && B	T	F	F	F

修正的判定-条件覆盖（续）

- 判定表达式：A && B

取值	组合1	组合2	组合3	组合4
A	T	T	F	F
B	T	F	T	F
A && B	T	F	F	F

修正的判定-条件覆盖（续）

- 判定表达式：A && B

取值	组合1	组合2	组合3	组合4
A	T	T	F	F
B	T	F	T	F
A && B	T	F	F	F

修正的判定-条件覆盖（续）

- 判定表达式：A && B

取值	组合1	组合2	组合3
A	T	T	F
B	T	F	T
A && B	T	F	F

修正的判定-条件覆盖（续）

测试用例设计的一般步骤

- 列出所有简单判定条件；
- 构建真值表；
- 对每个简单判定条件，找到能读整个判定结点产生独立影响的测试用例集合（简称独立影响对）；
- 抽取能体现所有简单判定条件独立影响性的最少独立影响对。

修正的判定-条件覆盖（续）

- 优势：综合具备条件组合覆盖的有点，有效控制了测试用例数量，消除了测试冗余。
- 不足：测试用例设计较为困难。

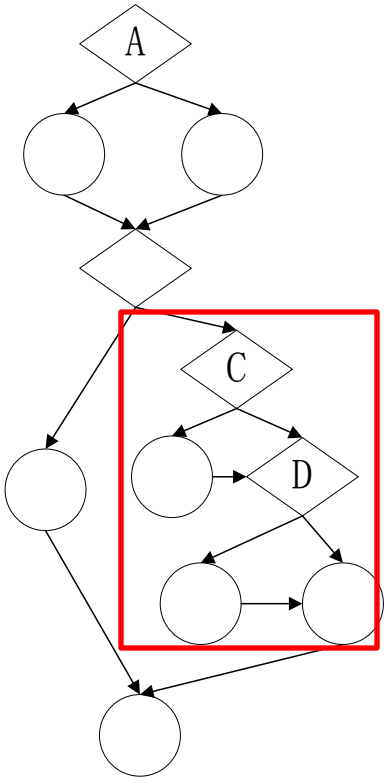
```
(year < 1800 || year > 2050)
```

常见的判定测试覆盖指标

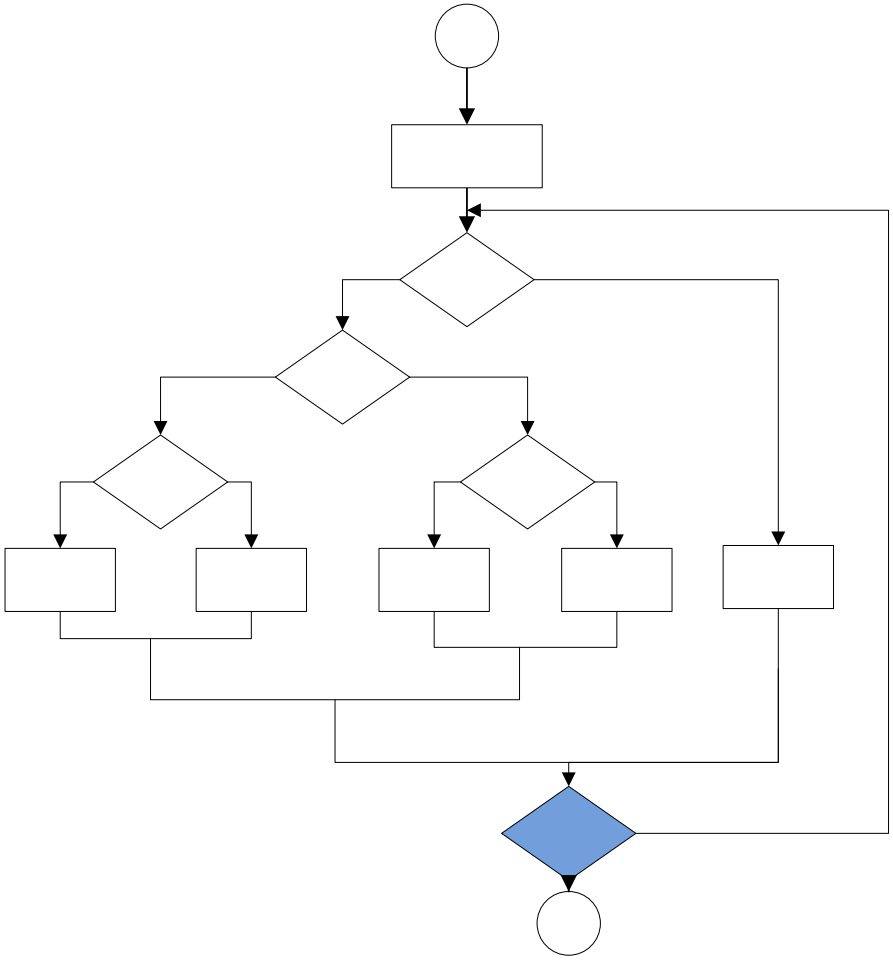
- 语句覆盖
- 判定覆盖
- 条件覆盖
- 判定-条件覆盖
- 条件组合覆盖
- 修正的判定-条件覆盖

路径覆盖

- 基本思想：设计足够多的测试用例要求覆盖程序中所有可能的路径。

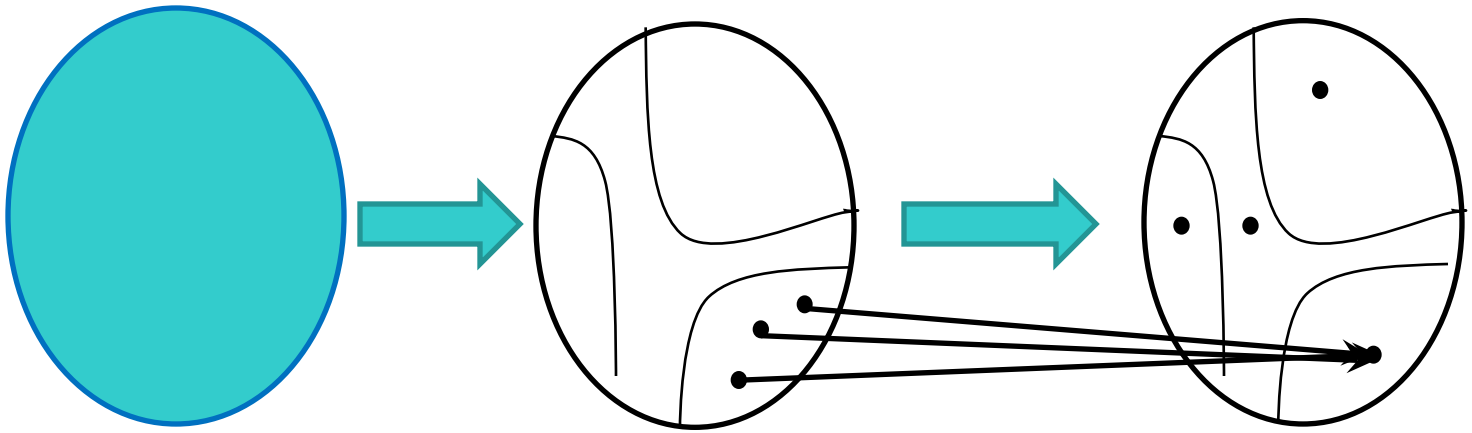


路径覆盖（续）



路径覆盖（续）

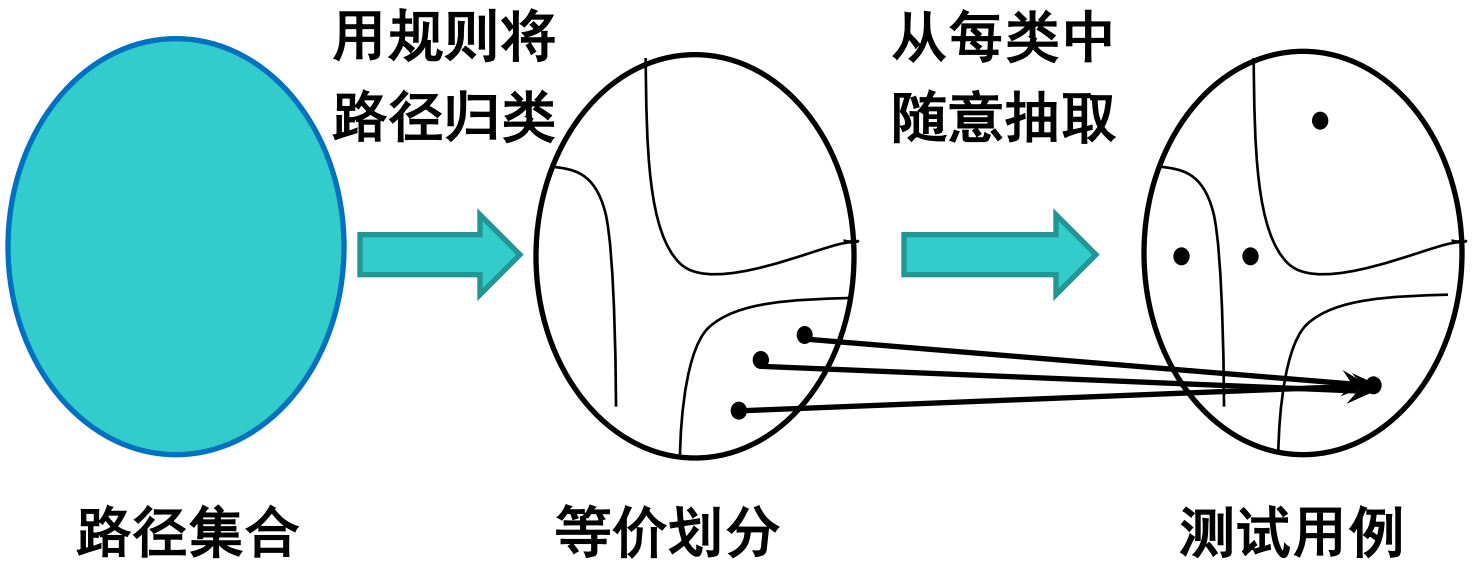
等价类测试的基本原理



输入域 **等价划分** **测试用例**

路径覆盖（续）

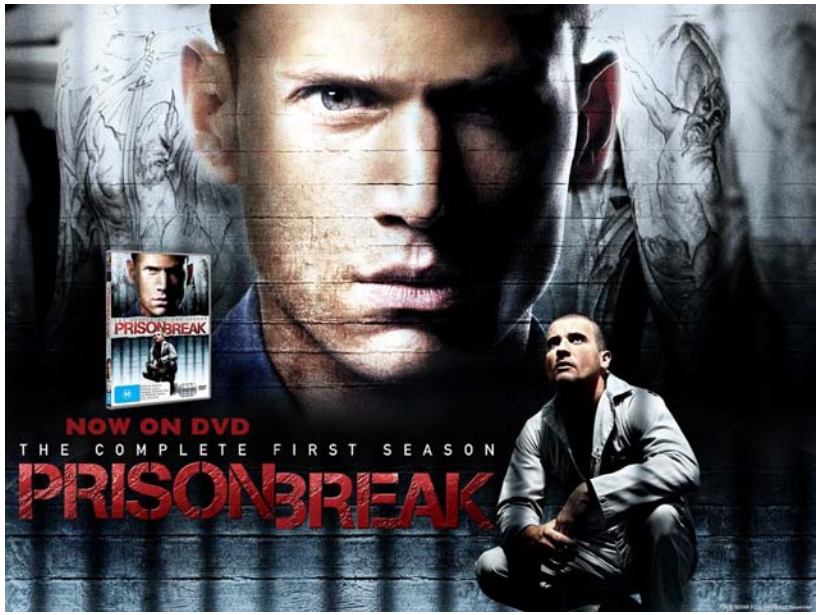
基于等价路径的测试



- 如何找到路径划分的规律？
- 如何定义路径的有效等价类和无效等价类

路径覆盖（续）

- 一张用于记录路线的地图
- 地图中的最少线性无关路径数
- 找到所有可能迅速逃离的最佳路线
- 其他……



路径覆盖（续）

- 用于记录程序路径的地图
- 最少线性无关路径数
- 所有可能迅速找到缺陷的最佳独立路径

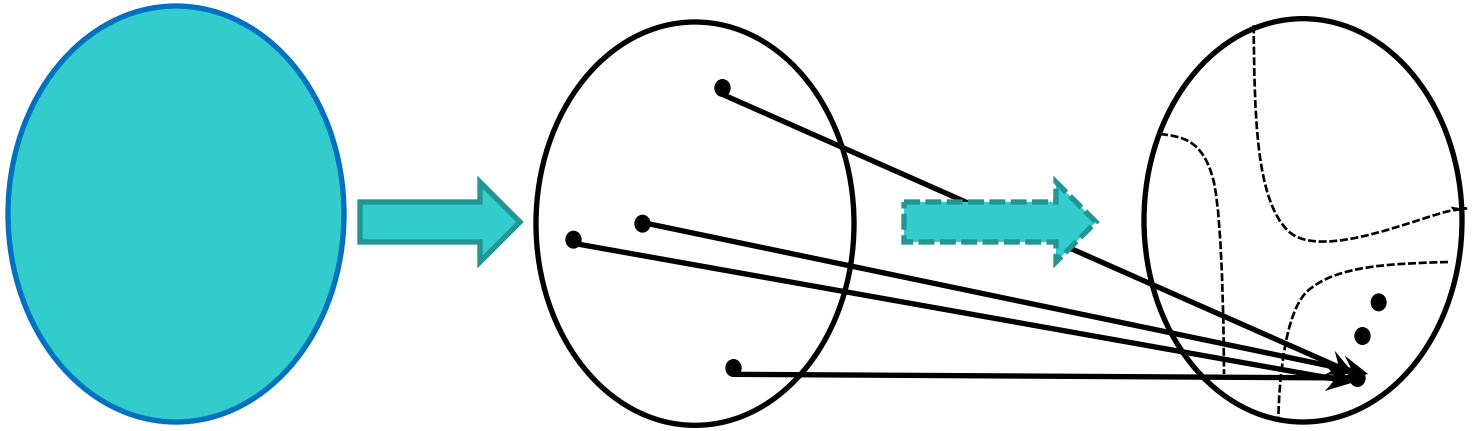
路径覆盖（续）

基于独立路径的测试

- 若将向量空间视作向量组，则基就是向量组的极大线性无关组，维数就是向量组的秩。
- 该向量空间内的任意向量，都可以用这组向量基来线性表示。

路径覆盖（续）

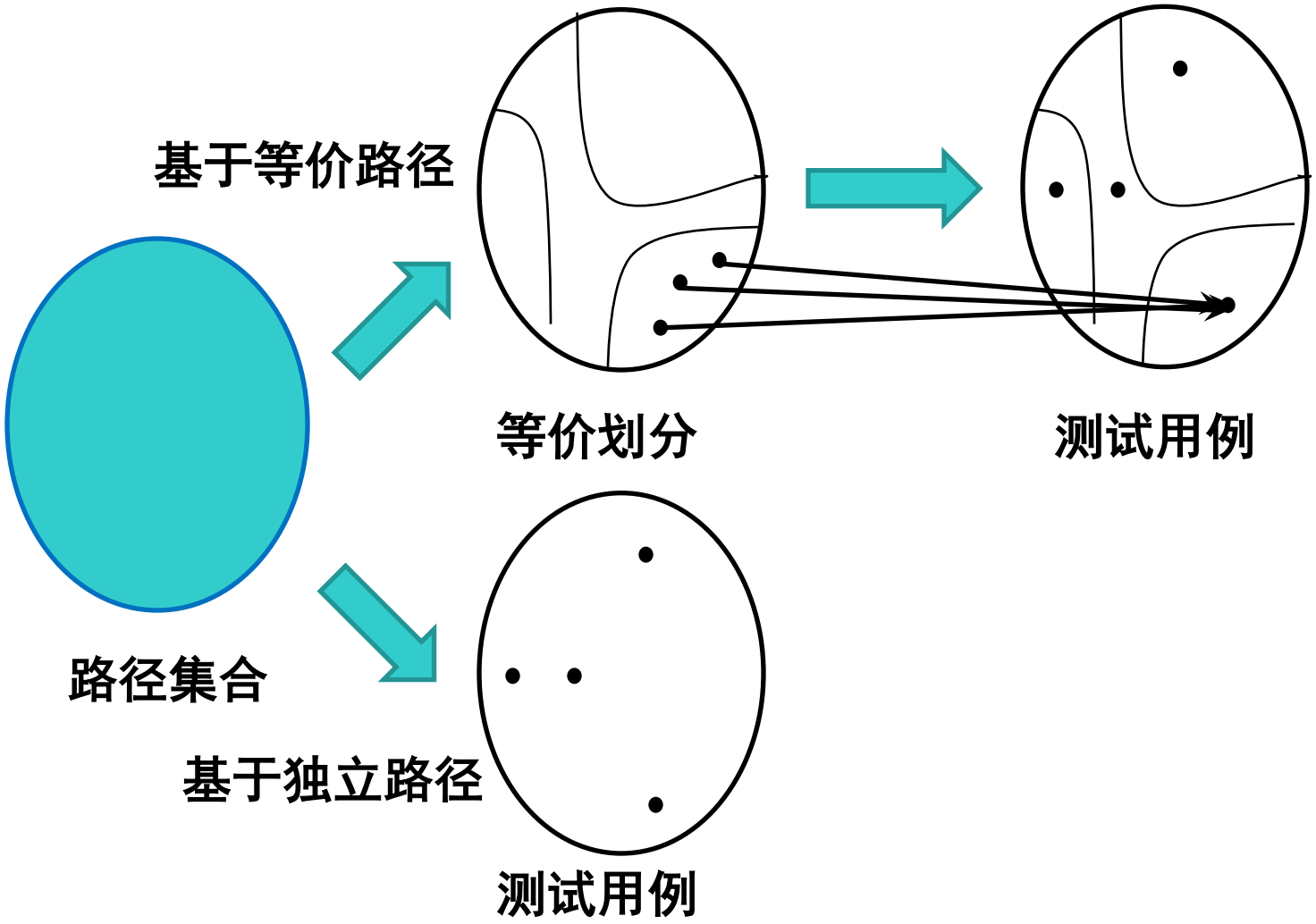
基于独立路径的测试



路径集合　　**测试用例**

- **1个约束：线性无关**
- **方法关键**
  - **如何确定向量空间的维数**
  - **如何确定这组独立路径**

路径覆盖（续）



路径覆盖（续）

核心问题

- 如何生成路径测试的地图
- 如何确定地图内最少线性无关路径数
- 如何找到最佳独立路径

路径覆盖（续）

重要术语

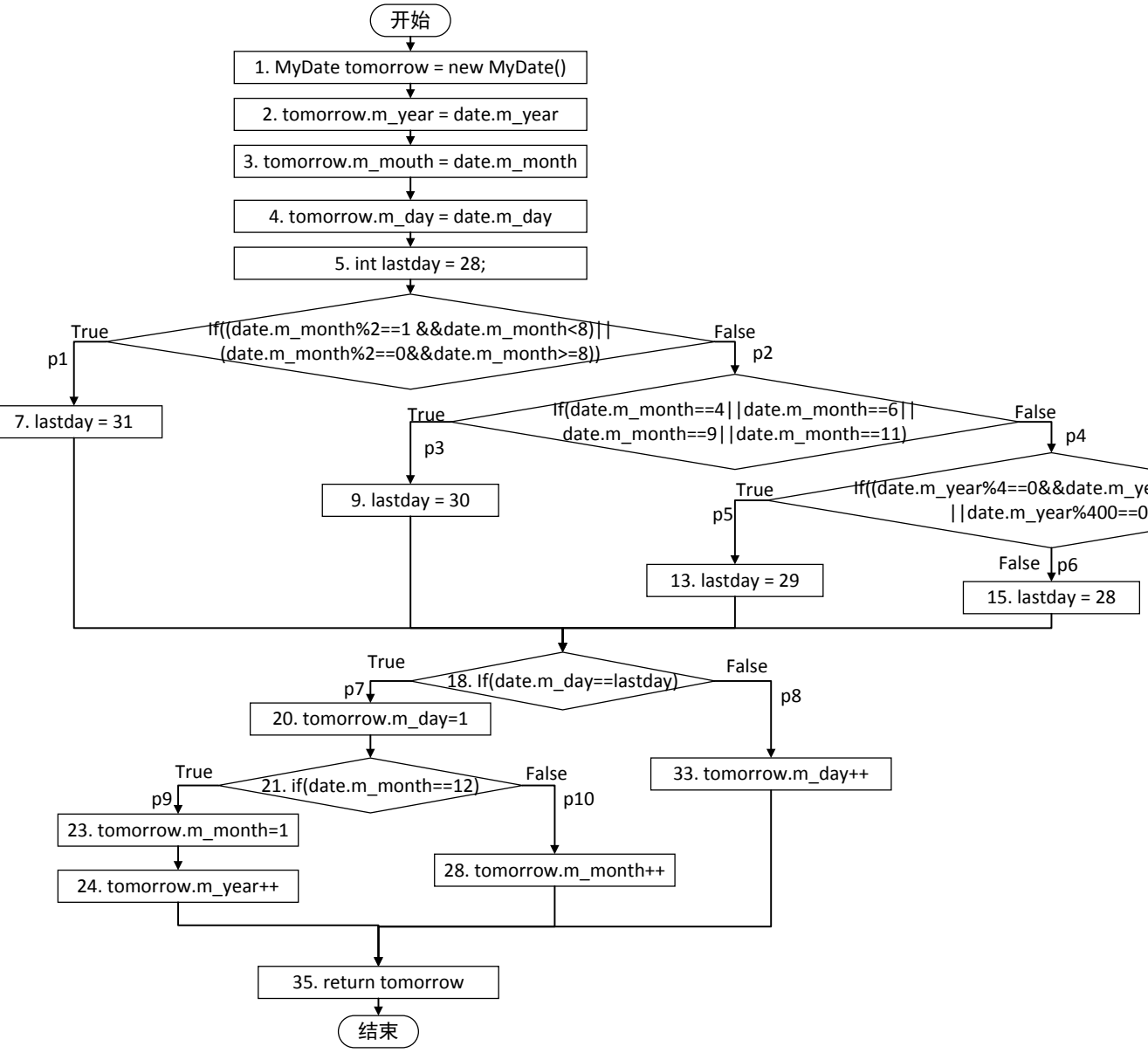
- **程序图**
- **环复杂度**

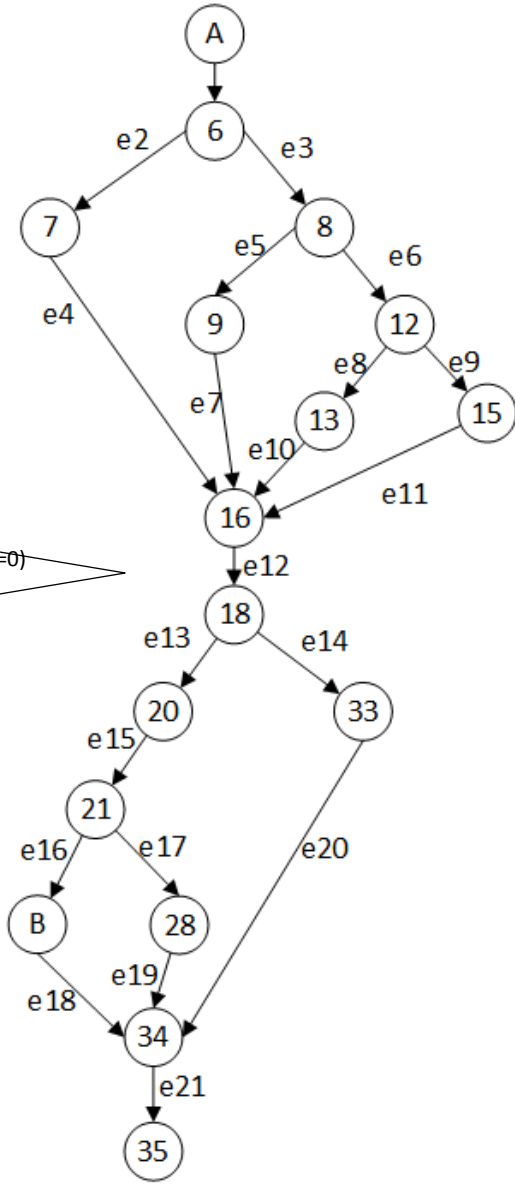
路径覆盖（续）

程序图

- 从源代码得到
- 用来表示程序结构的一种有向图
- 由节点和有向边构成
  - 节点：执行语句
  - 有向边：程序执行顺序

程序图是简化、压缩的流程图





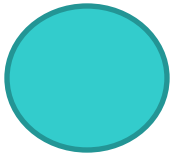
路径覆盖（续）

程序图：简化的流程图

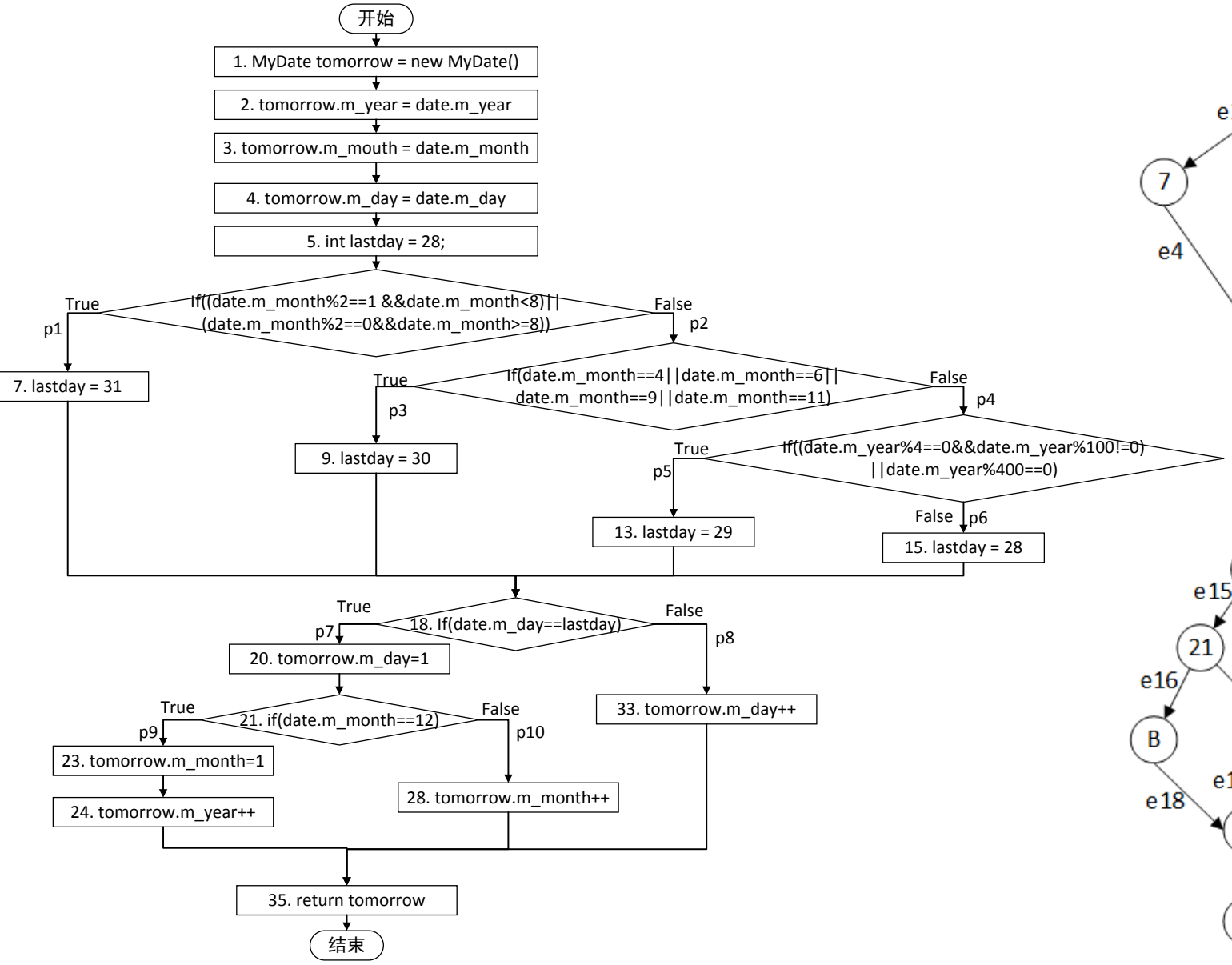
- **简化节点形状**

- **流程图**

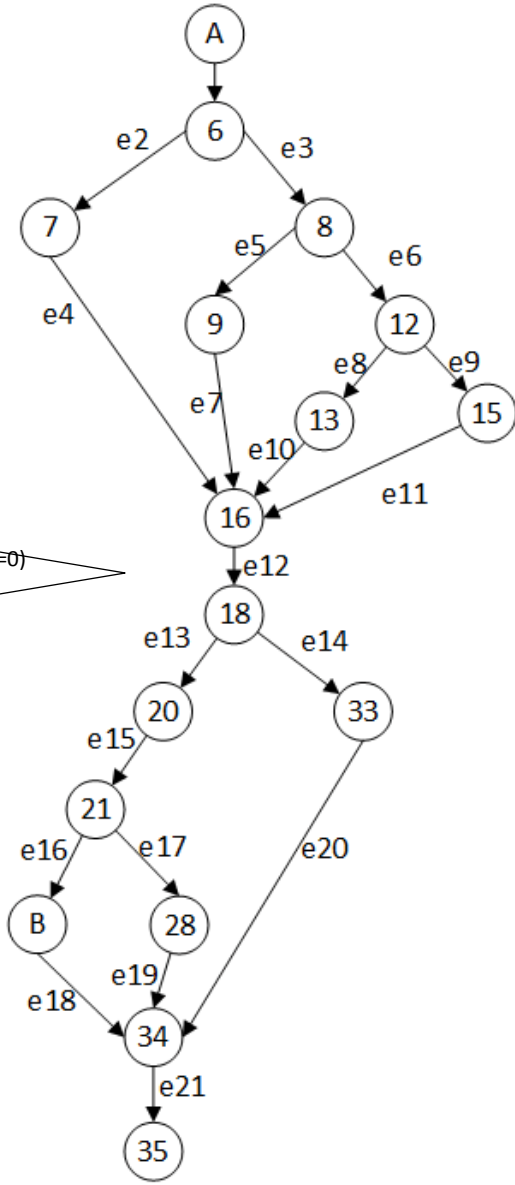
- **程序图**



流程图

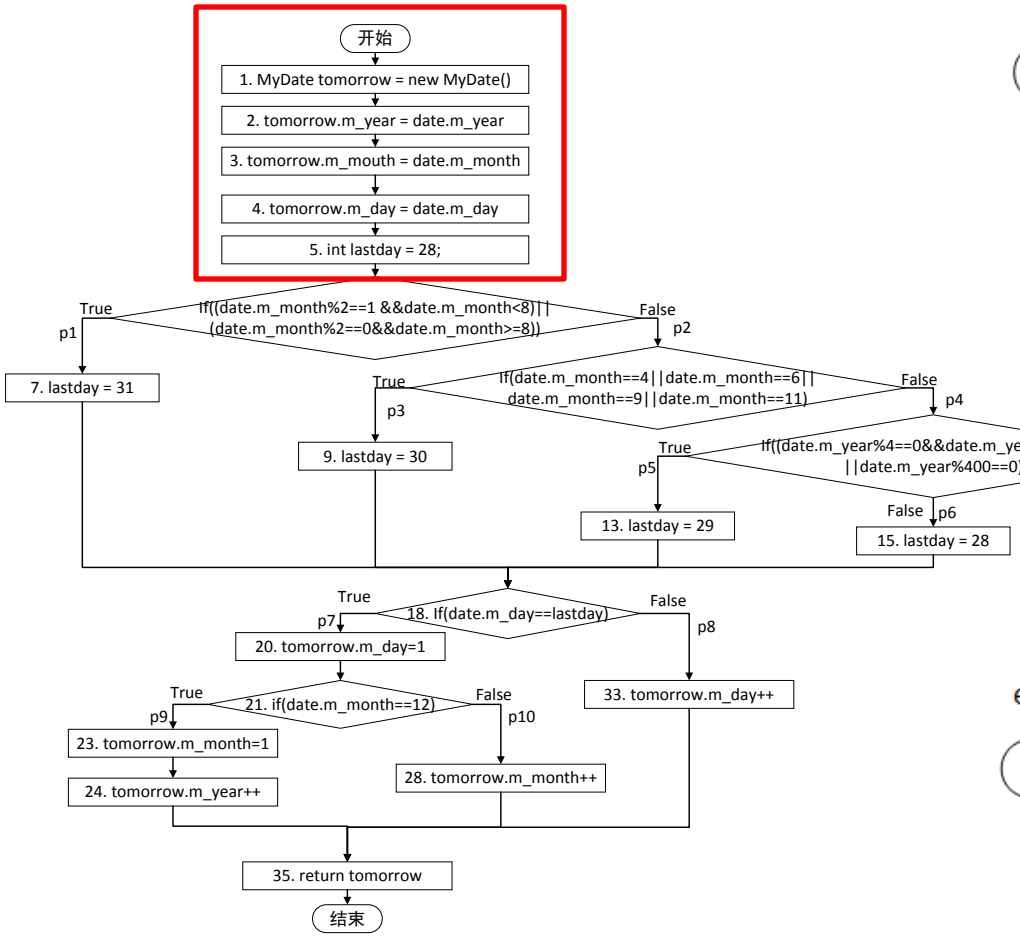

开始
1. MyDate tomorrow = new MyDate()
2. tomorrow.m_year = date.m_year
3. tomorrow.m_mouth = date.m_month
4. tomorrow.m_day = date.m_day
5. int lastday = 28;
If((date.m_month%2==1 &&date.m_month<8)||(date.m_month%2==0&&date.m_month>=8))
True
False
p1
p2
7. lastday = 31
If(date.m_month==4||date.m_month==6||date.m_month==9||date.m_month==11)
True
False
p3
p4
9. lastday = 30
If((date.m_year%4==0&&date.m_year%100!=0)||date.m_year%400==0)
True
False
p5
p6
13. lastday = 29
15. lastday = 28
18. If(date.m_day==lastday)
True
False
p7
p8
20. tomorrow.m_day=1
33. tomorrow.m_day++
21. if(date.m_month==12)
True
False
p9
p10
23. tomorrow.m_month=1
24. tomorrow.m_year++
28. tomorrow.m_month++
35. return tomorrow
结束

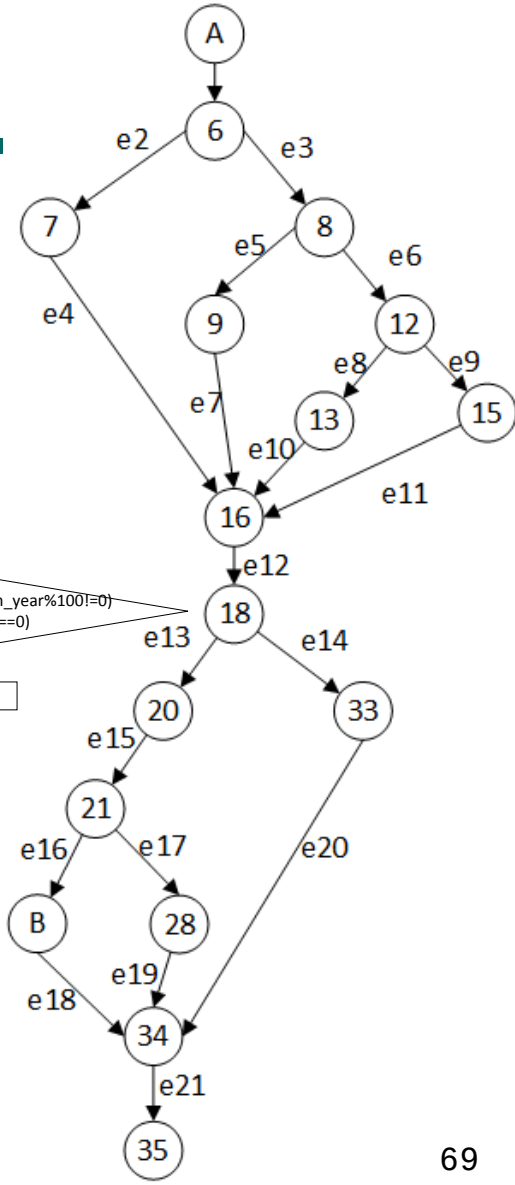
程序图


A
6
e2
e3
7
8
e5
e6
e4
9
12
e8
e9
e7
13
15
e10
e11
16
e12
18
e13
e14
20
33
e15
21
e16
e17
e20
B
28
e19
e18
34
e21
35

路径覆盖（续）

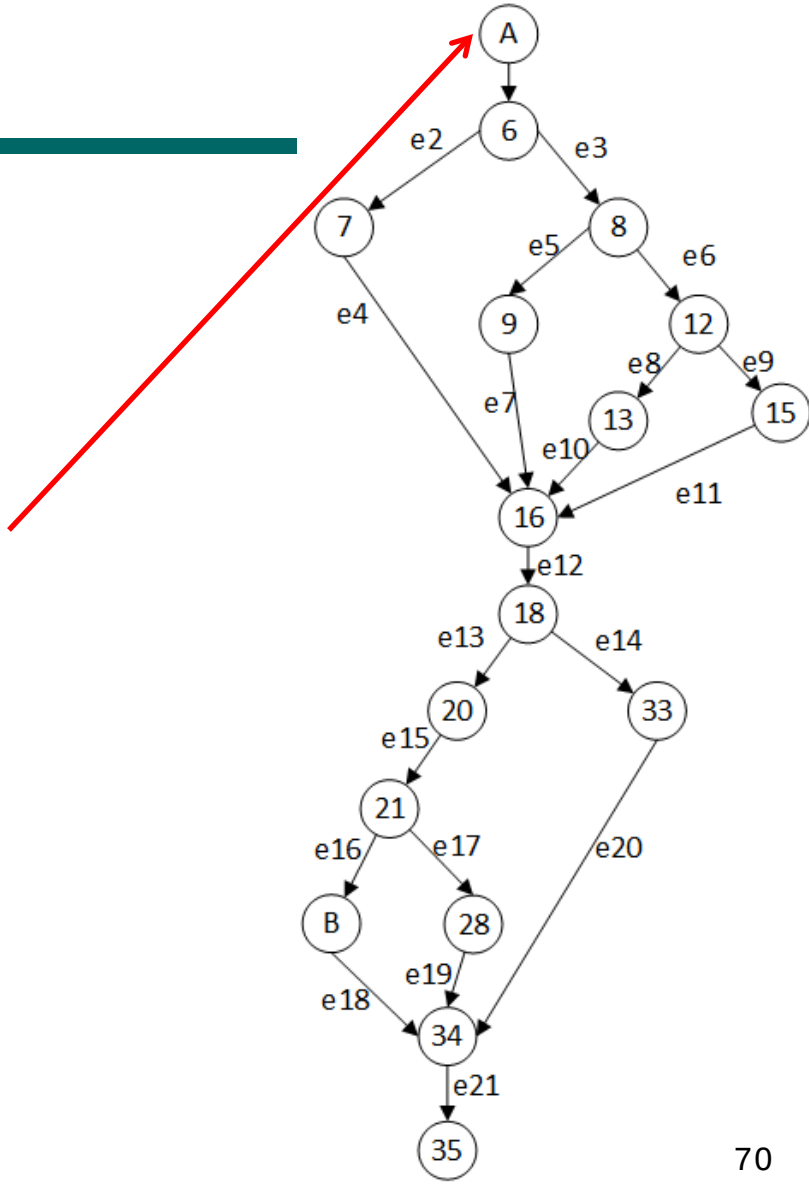
- 数据声明
- 注释
- 串行语句
- 循环结构





路径覆盖（续）


开始
1. MyDate tomorrow = new MyDate()
2. tomorrow.m_year = date.m_year
3. tomorrow.m_mouth = date.m_month
4. tomorrow.m_day = date.m_day
5. int lastday = 28;


A
6
e2
e3
7
8
e5
e6
e4
9
12
e8
e9
e7
13
15
e10
e11
16
e12
18
e13
e14
20
33
e15
21
e16
e17
e20
B
28
e19
e18
34
e21
35

路径覆盖（续）

环复杂度

- McCabe复杂性度量
  - 一种定量描述程序结构复杂度的度量模型
  - 能够反映判定节点和循环的引入对程序结构以及执行路径数目带来的不利影响

路径覆盖（续）

环复杂度的确定

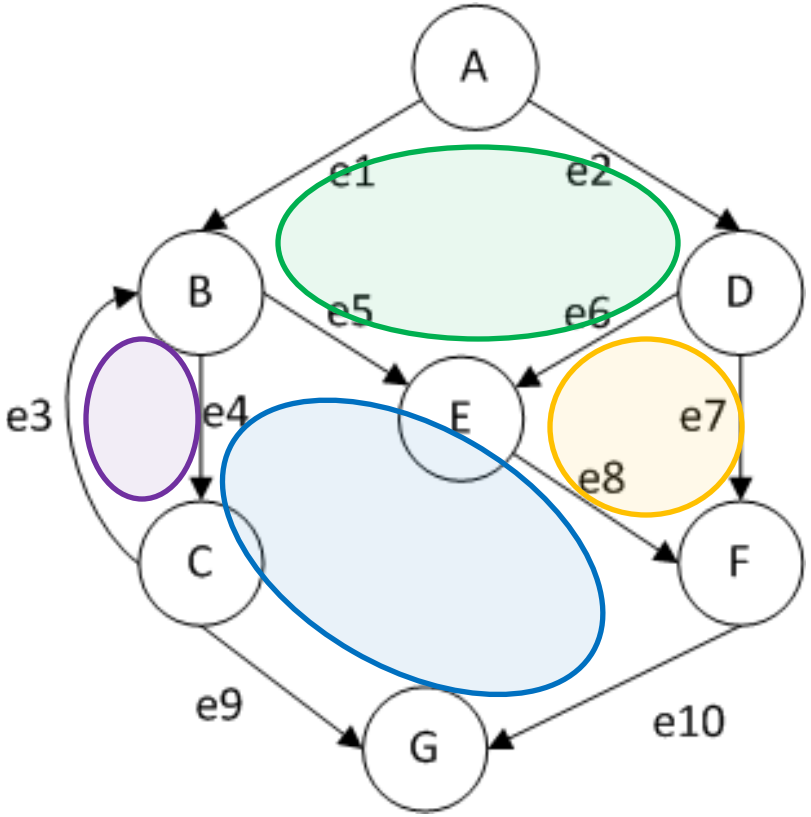
- **直观观察法**
- **公式计算法**
- **判定结点法**

路径覆盖（续）

环复杂度的确定

- **直观观察法**

  V＝5



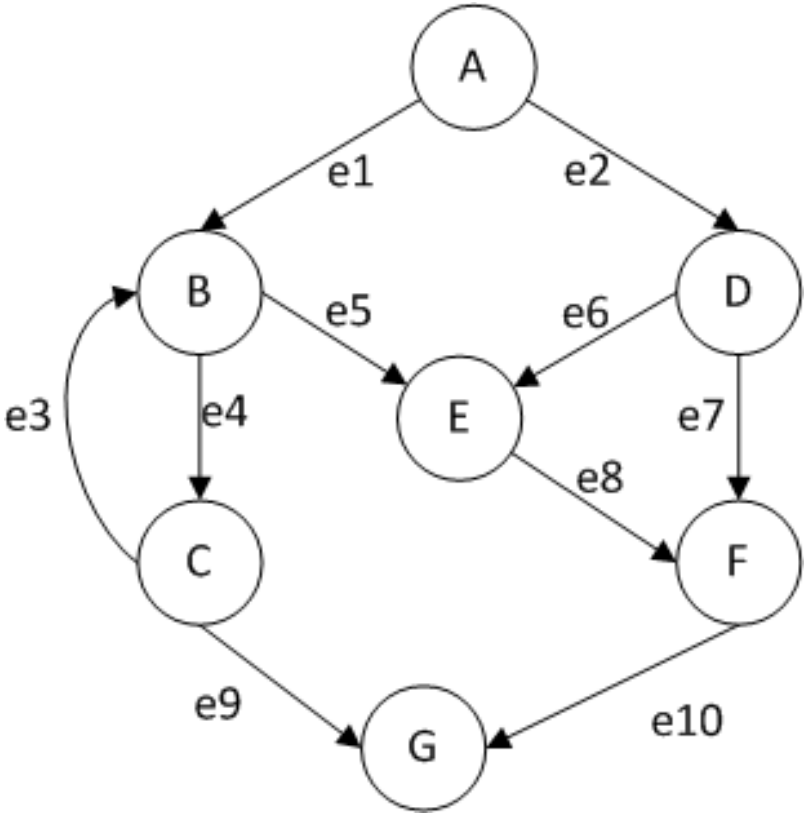
路径覆盖（续）

环复杂度的确定

- 公式计算法：

  $V(G)=e-n+1$

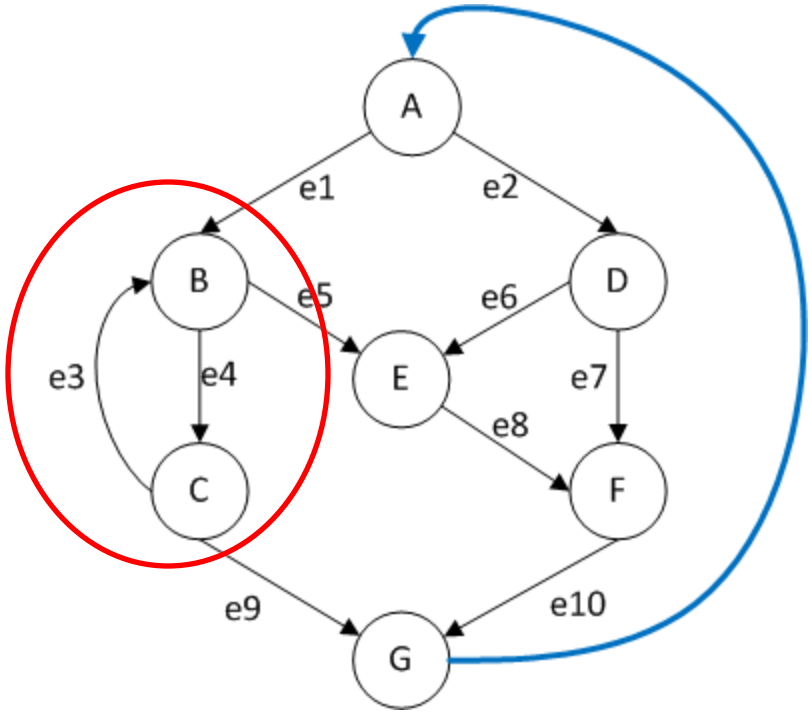
- 前提条件：
  - 程序图中无孤立节点
  - 程序图是强连通图



路径覆盖（续）

对程序图改造

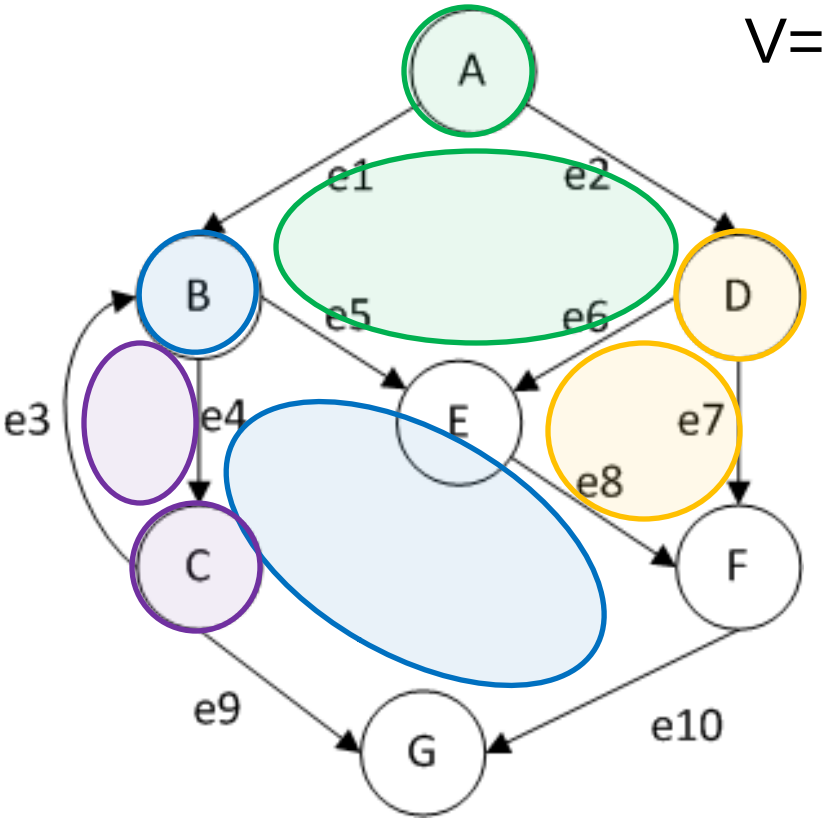
- 构建死循环
- V(G)=e-n+1
  =11-7+1
  =5



路径覆盖（续）

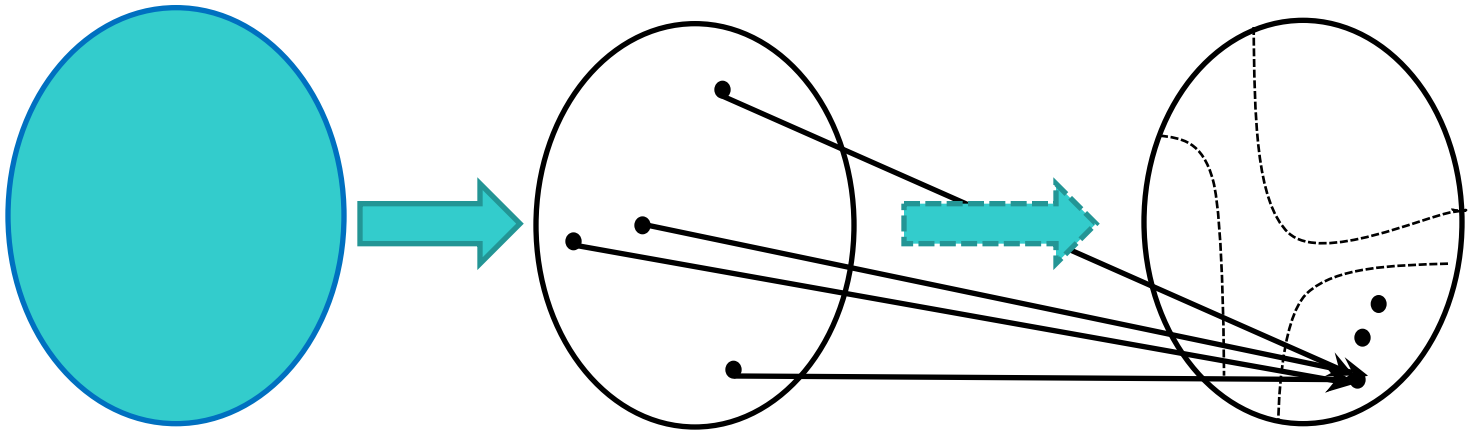
环复杂度的确定

- **判定结点法**
- **V＝P＋1**
- **前提条件：**
- **仅计算两分支的判定结点**



路径覆盖（续）

基于独立路径的测试



路径集合　　**测试用例**

路径覆盖（续）

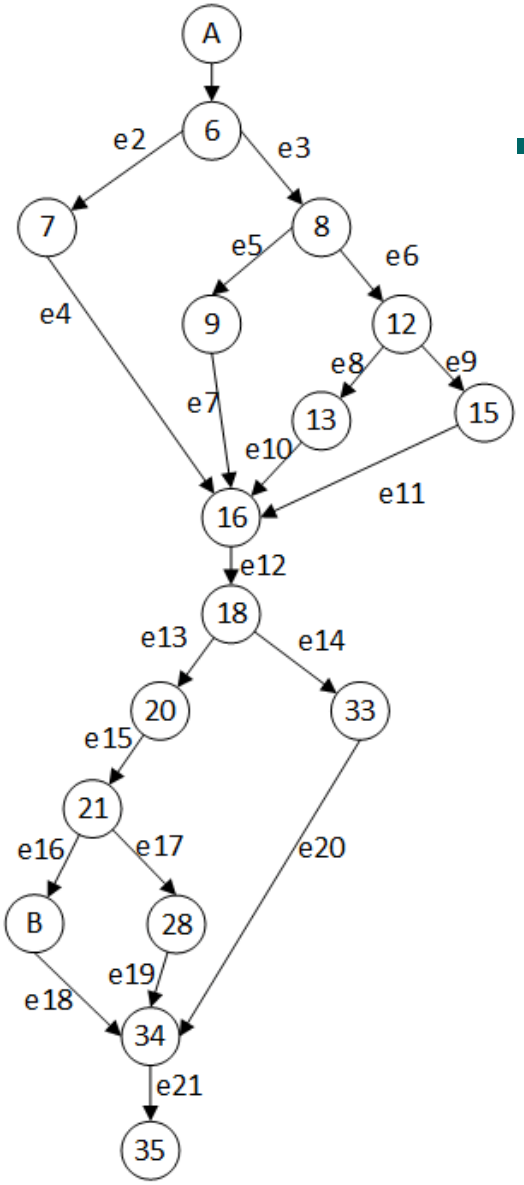
要解决的问题：

- 如何生成独立路径测试所需的地图？
- 如何确定独立路径集合的规模？
- 如何找到一组独立路径？

路径覆盖（续）

如何生成路径地图

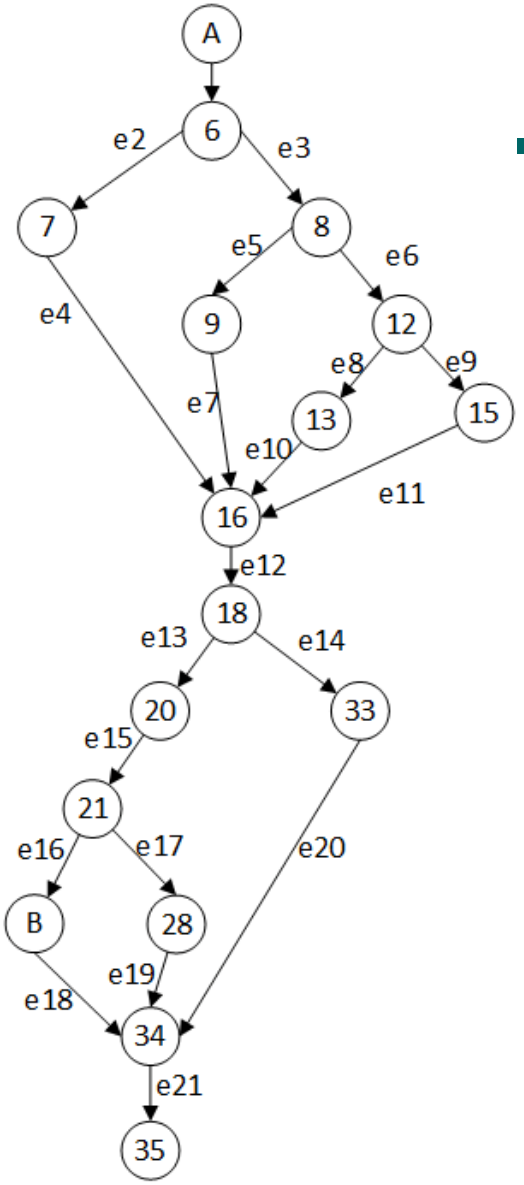
- ~~流程图~~
- ~~控制流图~~
- 程序图



路径覆盖（续）

如何确定独立路径集合规模

- 环复杂度

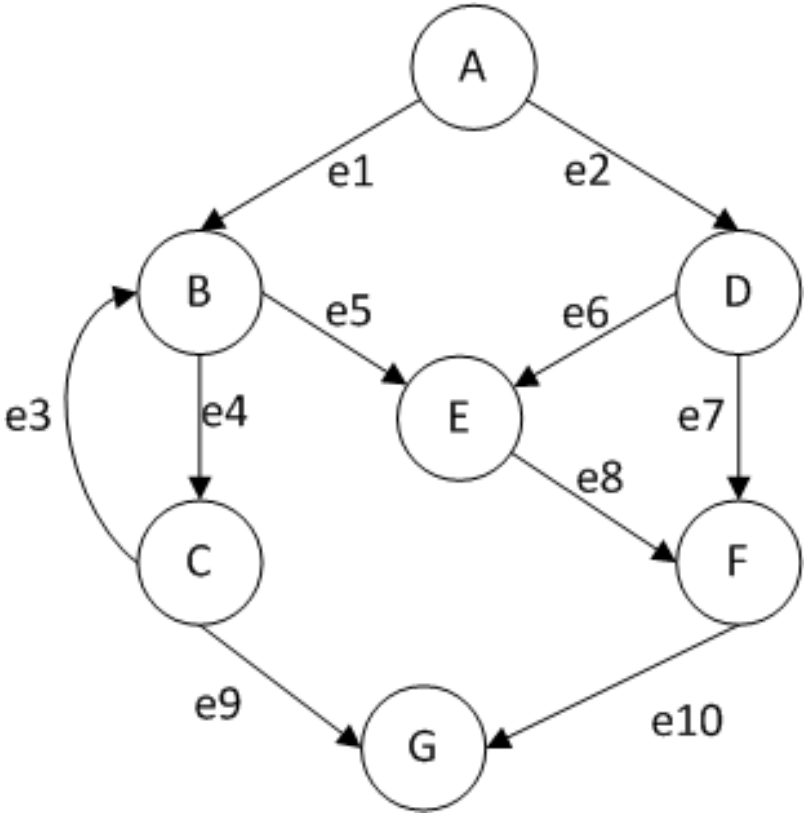


路径覆盖（续）

如何找到一组独立路径

- 确定主路径
- 根据主路径

  抽取其他独立路径



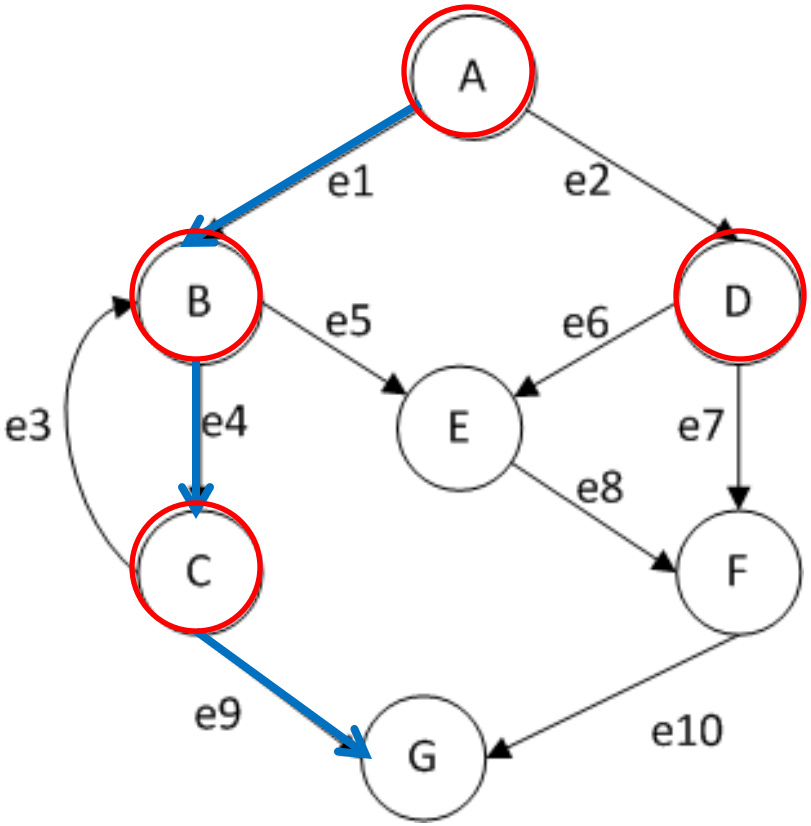
路径覆盖（续）

确定主路径

- 风险最高的路径
- 从结构复杂度看风险：

包含判定结点最多的路径

P1:A,B,C,G

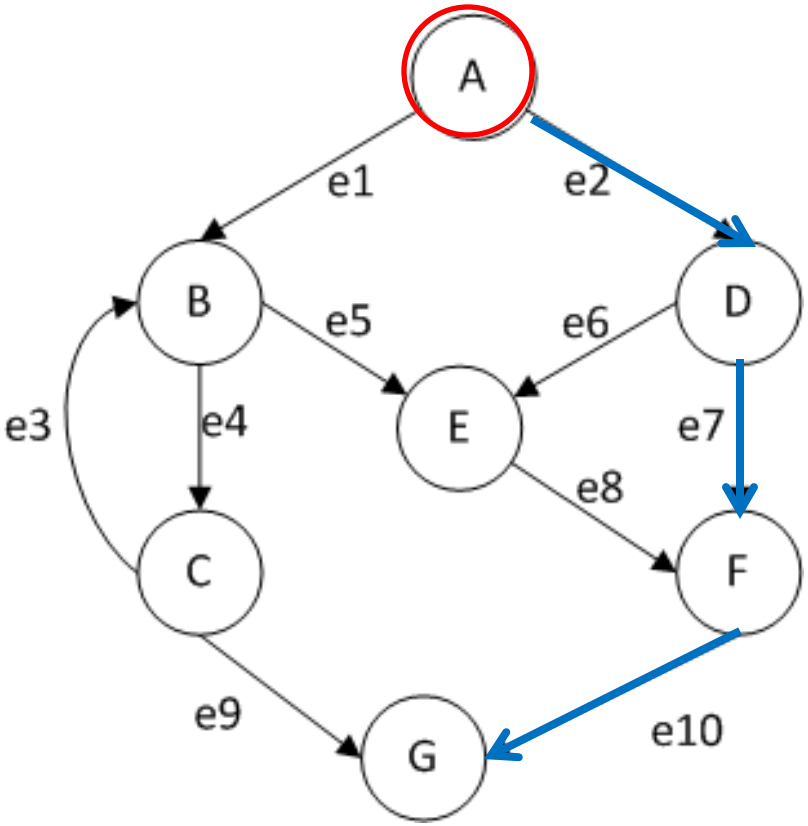


路径覆盖（续）

抽取其他独立路径

- P1:A,B,C,G
- 依次覆盖判定结点的不同执行分支

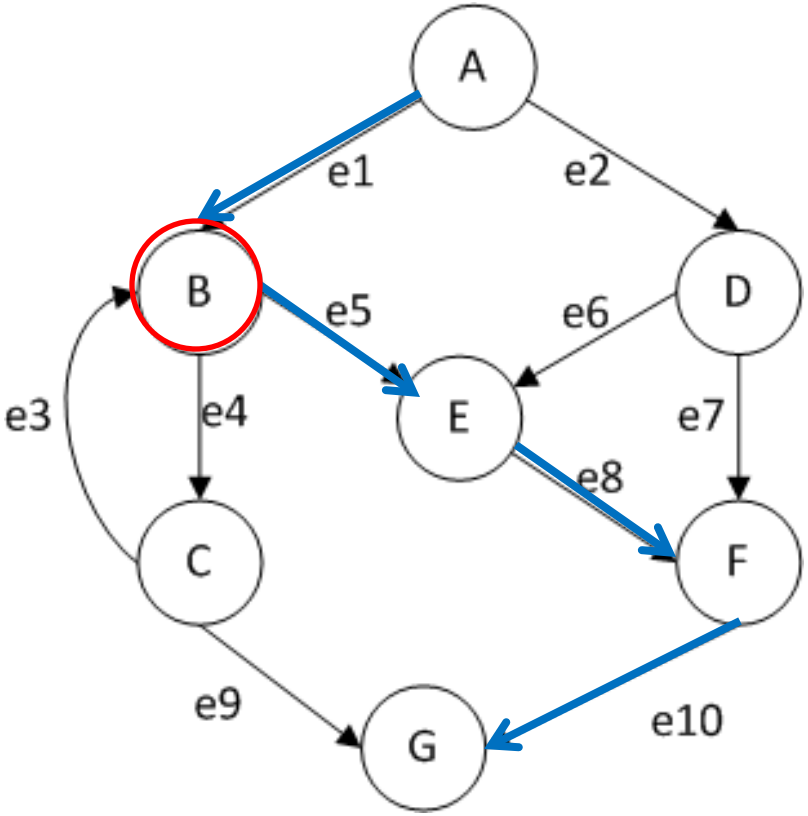
P2:A,D,F,G



路径覆盖（续）

抽取其他独立路径

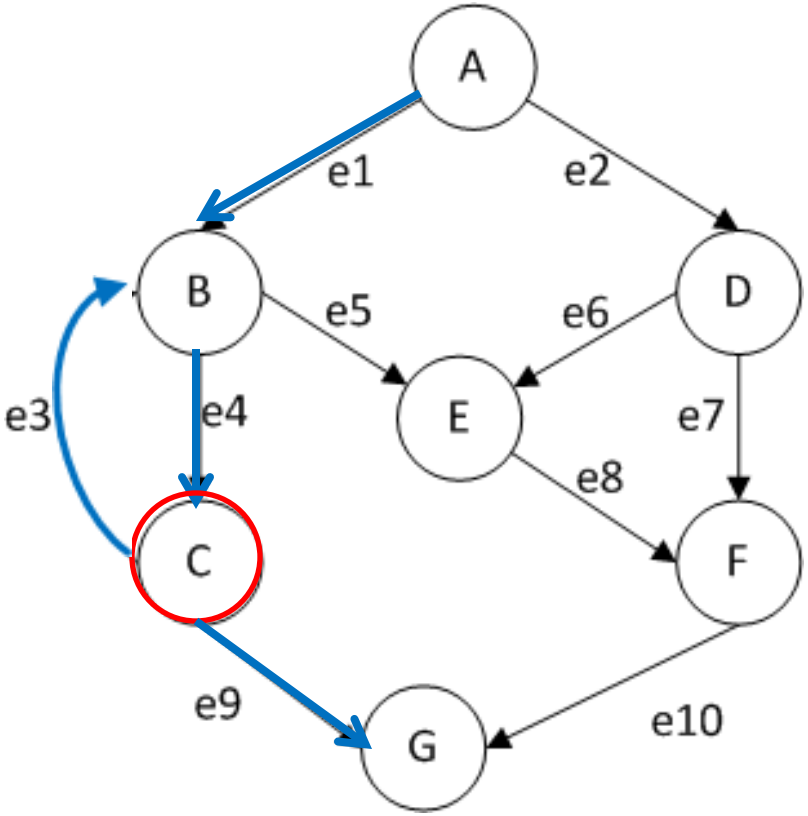
- P1:A,B,C,G
- P2:A,D,F,G
- P3:A,B,E,F,G



路径覆盖（续）

抽取其他独立路径

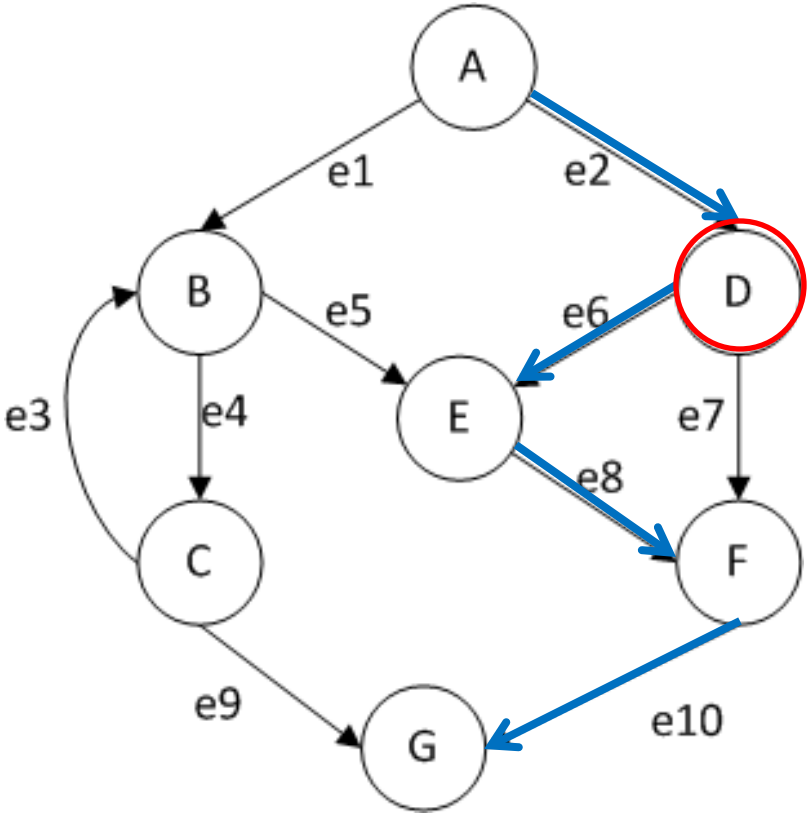
- P1:A,B,C,G
- P2:A,D,F,G
- P3:A,B,E,F,G
- P4:A,B,C,B,C,G



路径覆盖（续）

抽取其他独立路径

- P1:A,B,C,G
- P2:A,D,F,G
- P3:A,B,E,F,G
- P4:A,B,C,B,C,G
- P5:A,D,E,F,G



路径覆盖（续）

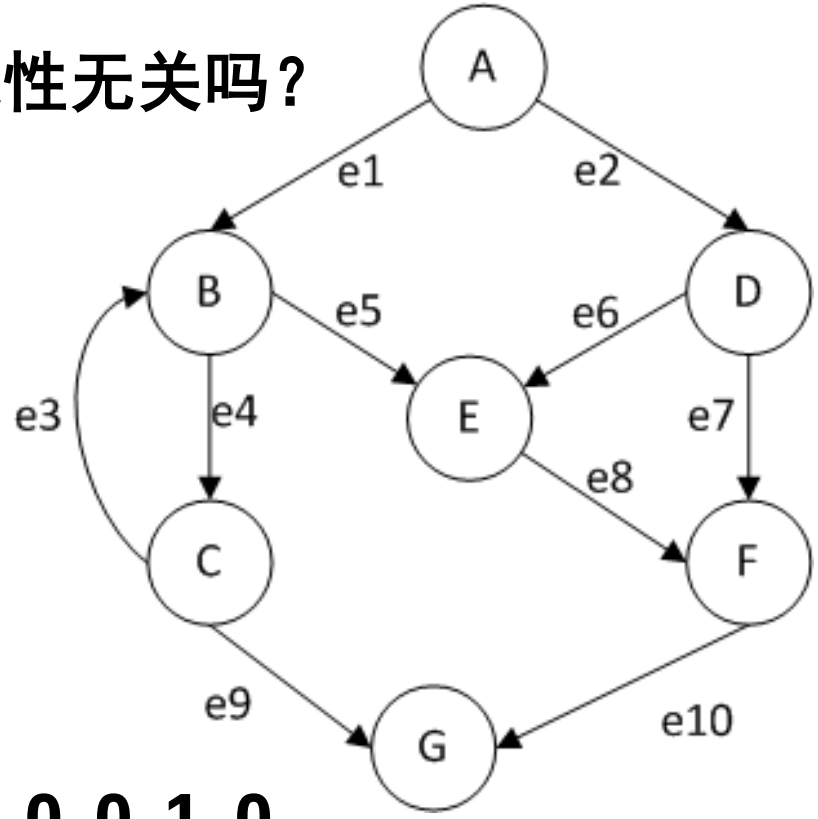
更多思考

- 这样得到的路径可以保证相互独立吗？
- 这样的路径集合是唯一的吗？

路径覆盖（续）

路径集合中的路径满足线性无关吗？

- P1:A,B,C,G
- P2:A,D,F,G
- P3:A,B,E,F,G
- P4:A,B,C,B,C,G
- P5:A,D,E,F,G

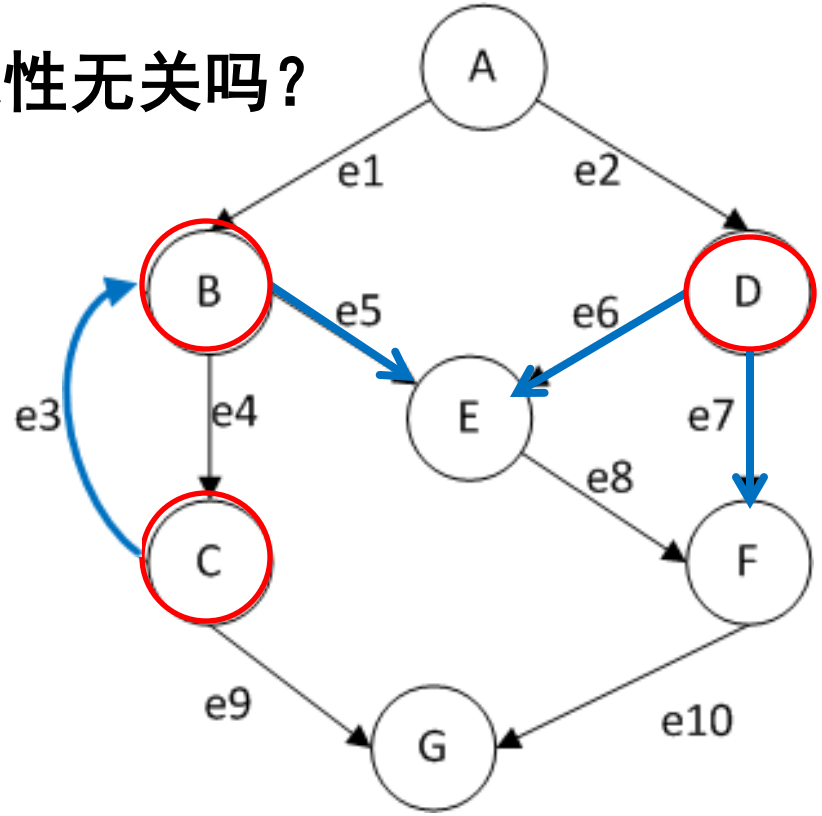


对应向量：1 0 0 1 0 0 0 0 1 0

路径覆盖（续）

路径集合中的路径满足线性无关吗？

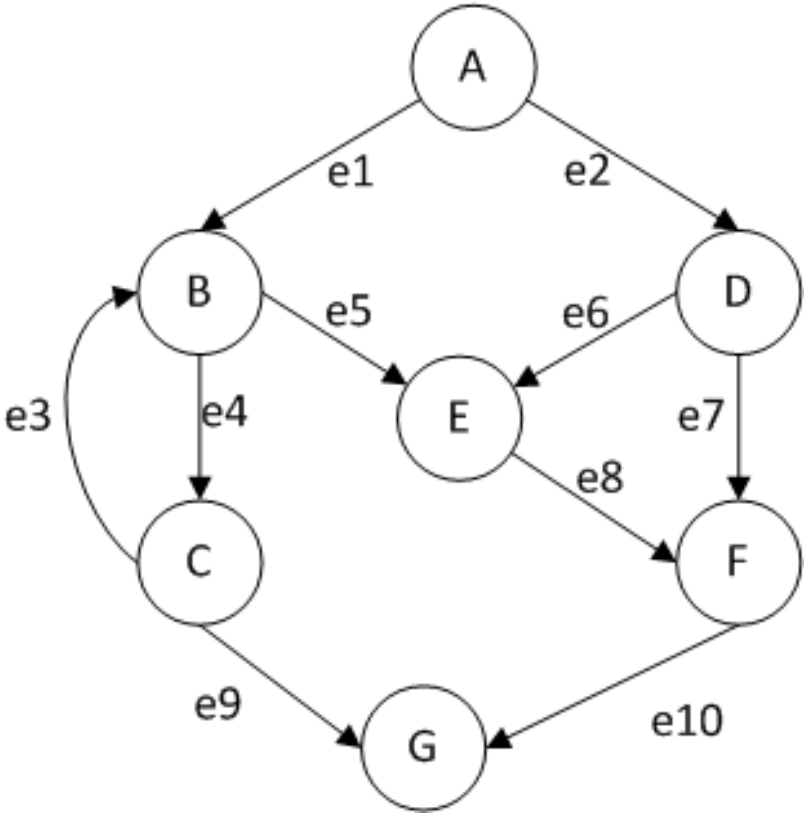
- P1:A,B,C,G
- P2:A,D,F,G
- P3:A,B,E,F,G
- P4:A,B,C,B,C,G
- P5:A,D,E,F,G



路径覆盖（续）

该路径集合是唯一的吗？

- P1:A,B,C,G
- P2:A,D,F,G
- P3:A,B,E,F,G
- P4:A,B,C,B,C,G
- P5:A,D,E,F,G



路径覆盖（续）

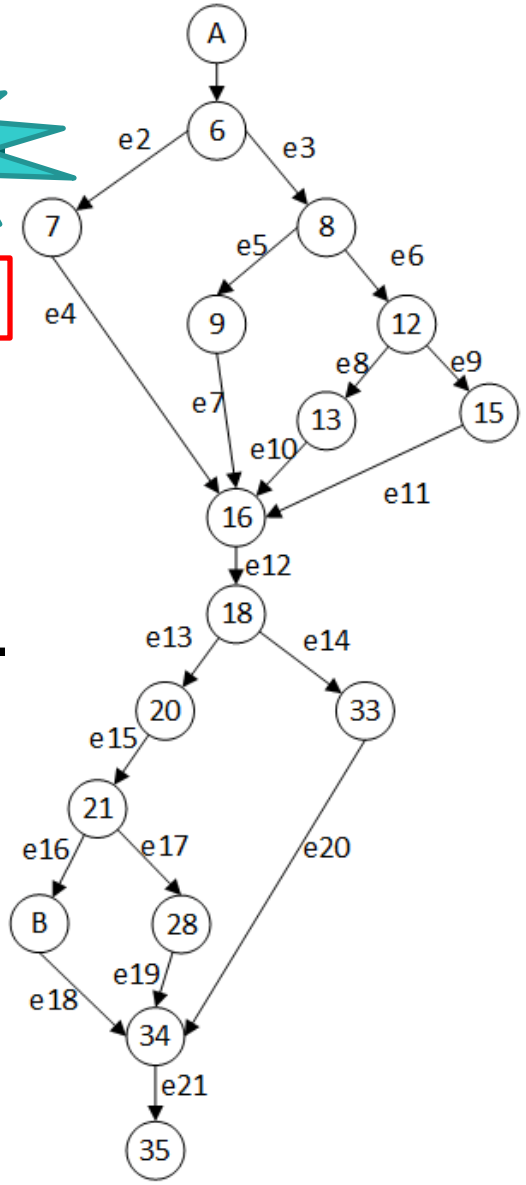
不可行？

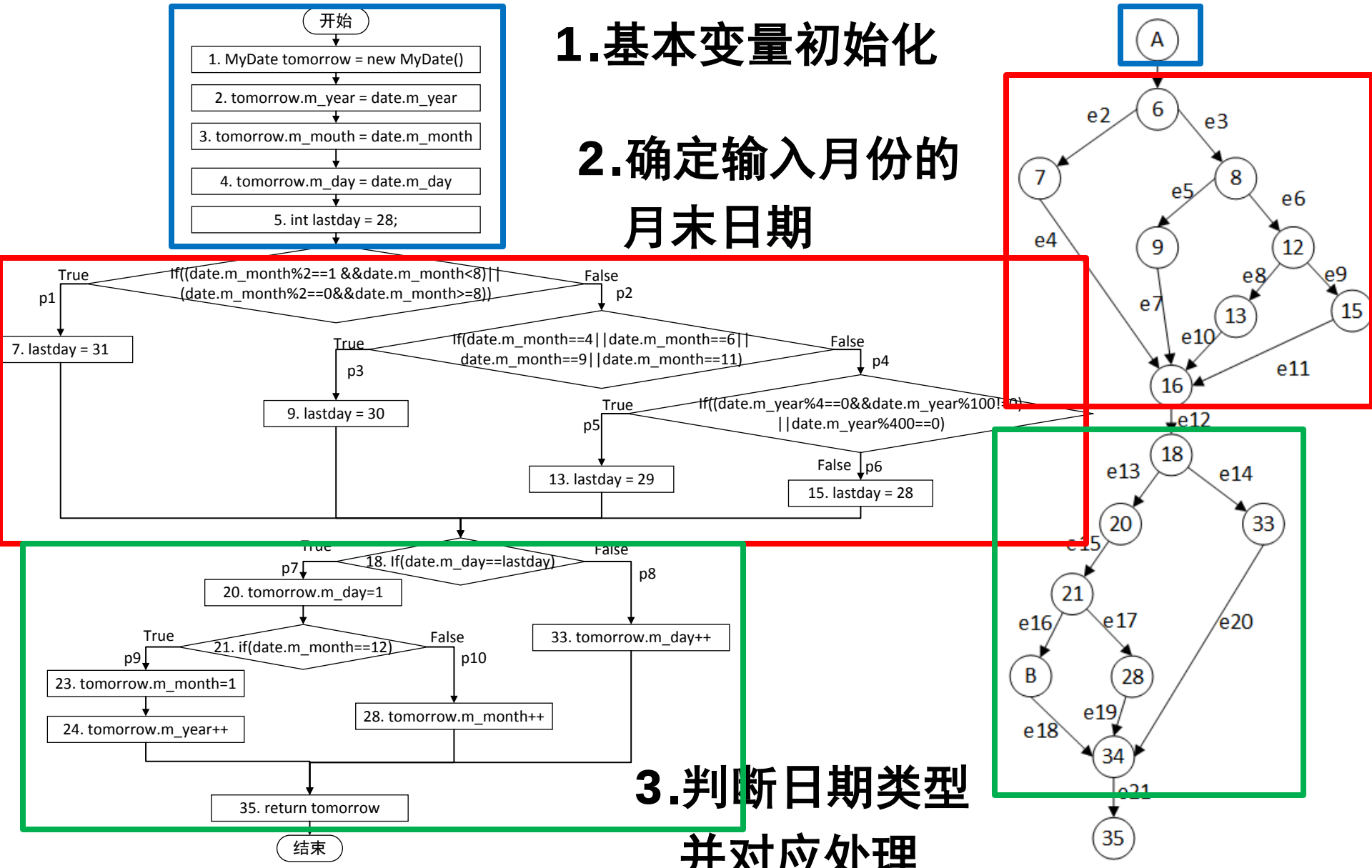
独立路径测试

- P1:A,6,8,12,13,16,18,20,21,B,34,35;
- P2:A,6,7,16,18,20,21,B,34,35;
- P3:A,6,8,9,16,18,20,21,B,34,35;
- P4:A,6,8,12,15,16,18,20,21,B,34,35;
- P5:A,6,8,12,13,16,18,33,34,35;
- P6:A,6,8,12,13,16,18,20,21,28,34,35.

独立路径:6条

完整路径:12条

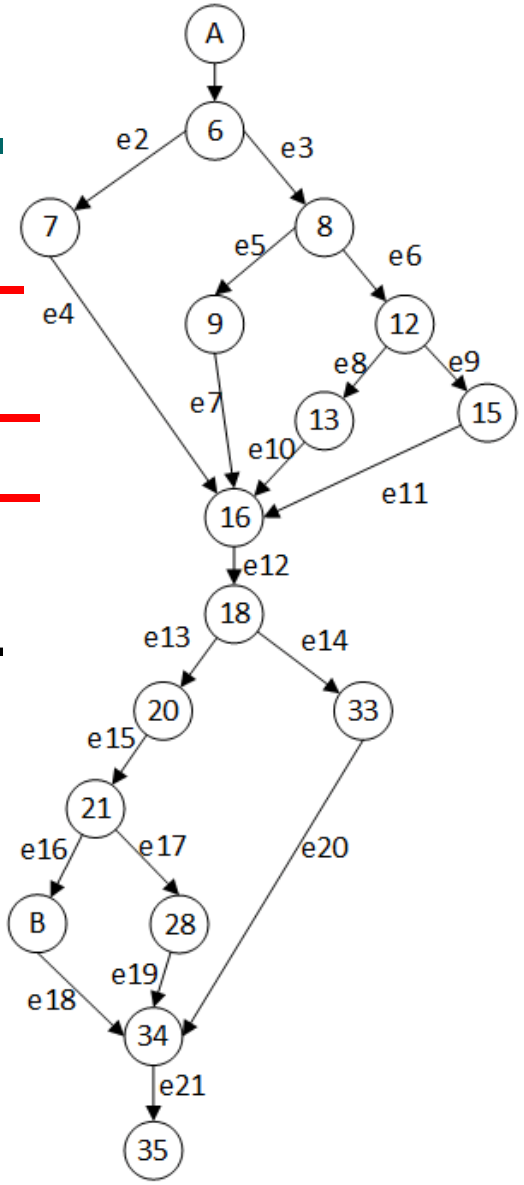



开始
1. MyDate tomorrow = new MyDate()
2. tomorrow.m_year = date.m_year
3. tomorrow.m_mouth = date.m_month
4. tomorrow.m_day = date.m_day
5. int lastday = 28;
1.基本变量初始化
2.确定输入月份的月末日期
3.判断日期类型并对应处理
If((date.m_month%2==1 &&date.m_month<8)||(date.m_month%2==0&&date.m_month>=8))
True
False
p1
p2
7. lastday = 31
If(date.m_month==4||date.m_month==6||date.m_month==9||date.m_month==11)
True
False
p3
p4
9. lastday = 30
If((date.m_year%4==0&&date.m_year%100!=0)||date.m_year%400==0)
True
False
p5
p6
13. lastday = 29
15. lastday = 28
18. If(date.m_day==lastday)
True
False
p7
p8
20. tomorrow.m_day=1
33. tomorrow.m_day++
21. if(date.m_month==12)
True
False
p9
p10
23. tomorrow.m_month=1
24. tomorrow.m_year++
28. tomorrow.m_month++
35. return tomorrow
结束
A
6
e2
e3
7
8
e5
e6
9
12
e4
e8
e9
e7
13
15
e10
16
e11
e12
18
e13
e14
20
33
e15
21
e16
e17
e20
B
28
e19
e18
34
e21
35

路径覆盖（续）

不可行路径

- ~~P1:A,6,8,12,13,16,18,20,21,B,34,35;~~
- P2:A,6,7,16,18,20,21,B,34,35;
- ~~P3:A,6,8,9,16,18,20,21,B,34,35;~~
- ~~P4:A,6,8,12,15,16,18,20,21,B,34,35;~~
- P5:A,6,8,12,13,16,18,33,34,35;
- P6:A,6,8,12,13,16,18,20,21,28,34,35.



路径覆盖（续）

不可行路径问题

- 不可行路径对测试造成什么影响？
- 如何处理不可行路径？
- 不可行路径如何产生的？
- 开发过程中如何避免不可行路径？

路径覆盖（续）

不可行路径带来的影响

- 破坏了独立路径测试的完备性和无冗余性
- 增大了测试用例设计的难度

路径覆盖（续）

如何处理不可行路径？

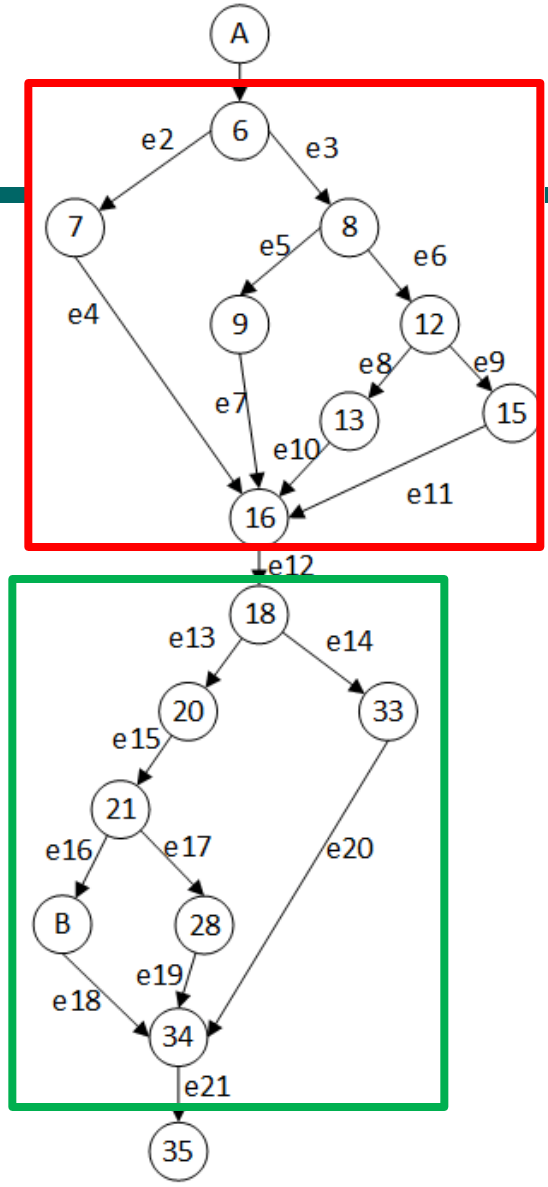
- 结合源代码寻找独立路径
- 补充其他具有较高风险的路径进行测试

路径覆盖（续）

不可行路径如何产生？

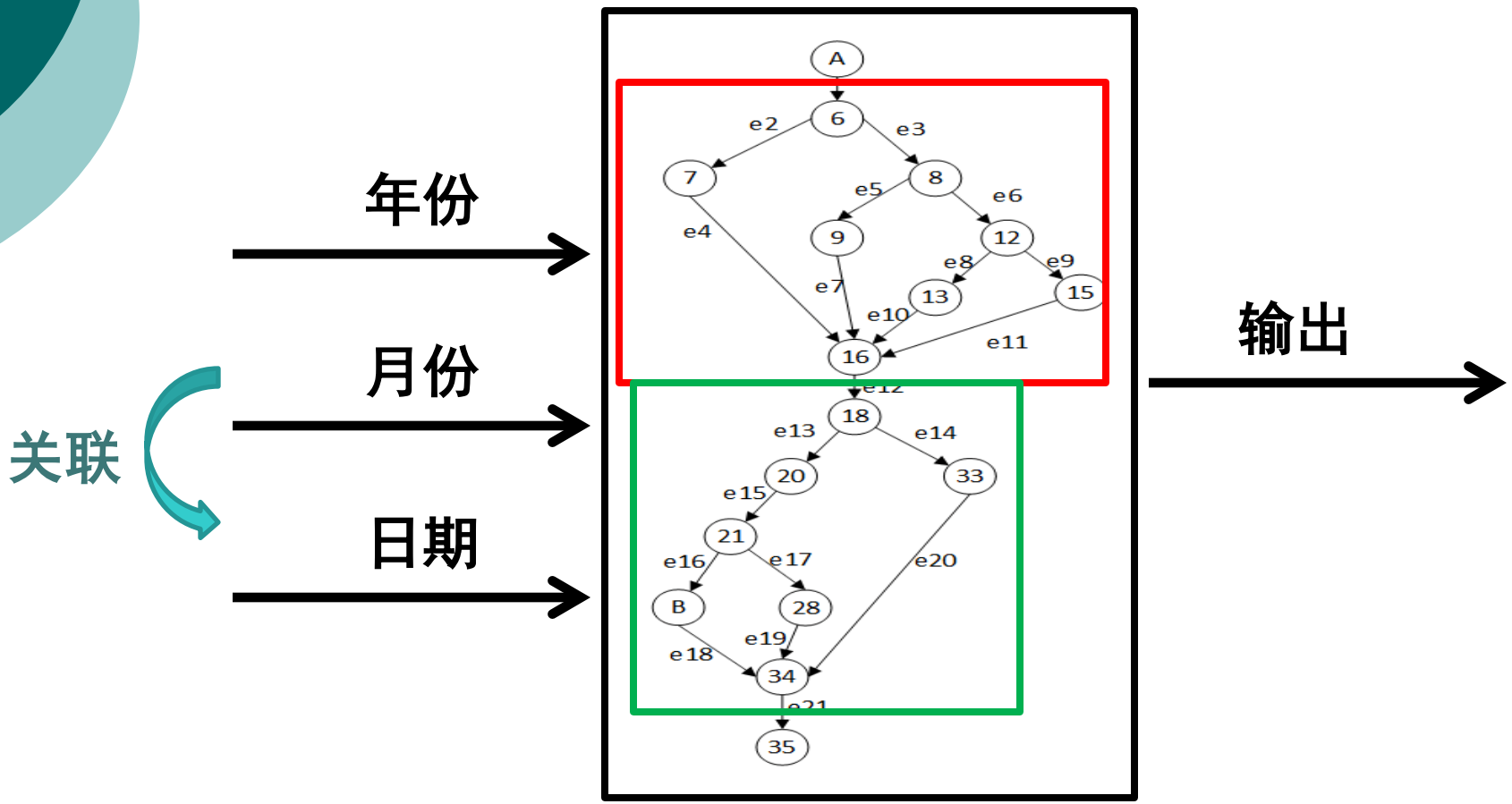
2.确定输入月份的
月末日期

3.判断日期类型
并对应处理



路径覆盖（续）

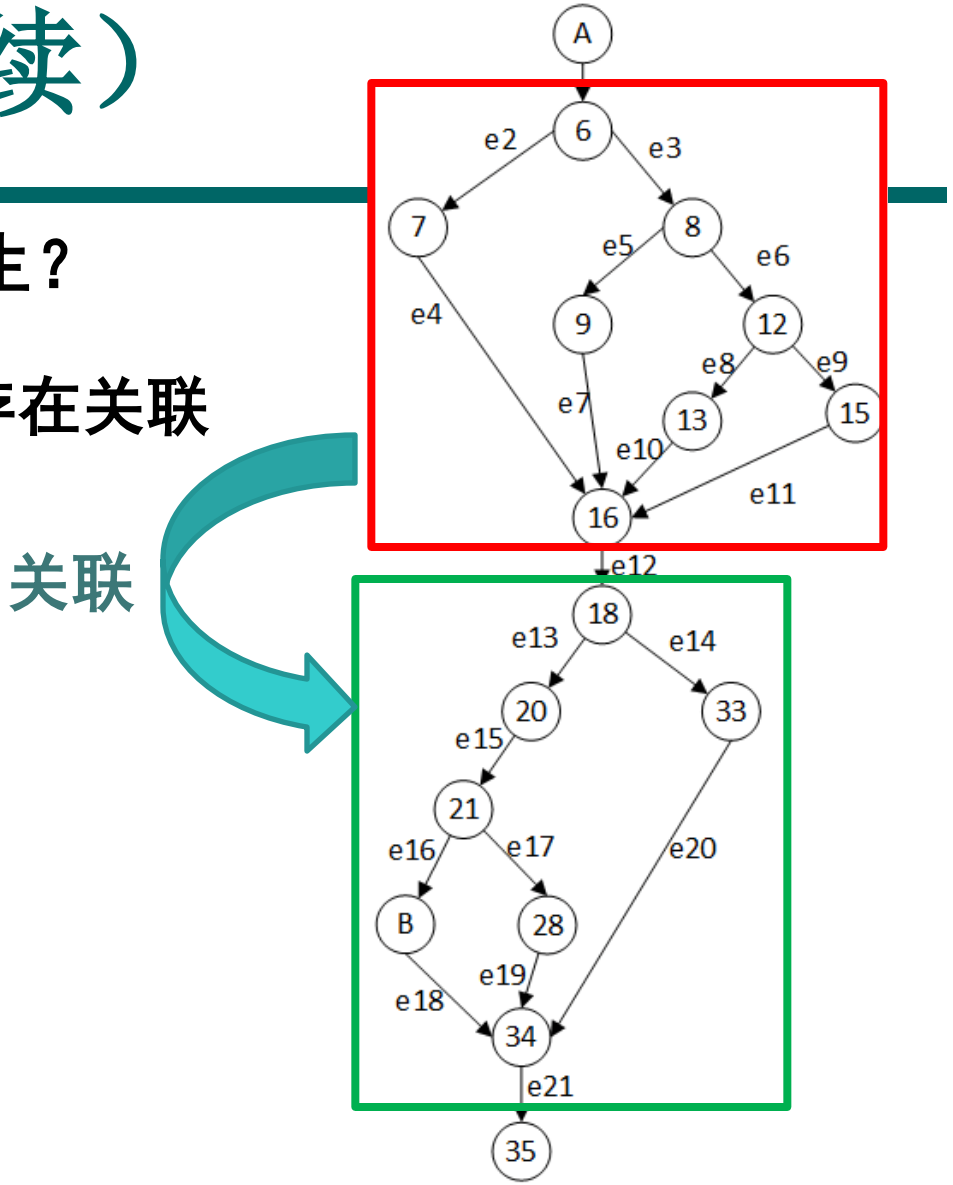
不可行路径如何产生？



路径覆盖（续）

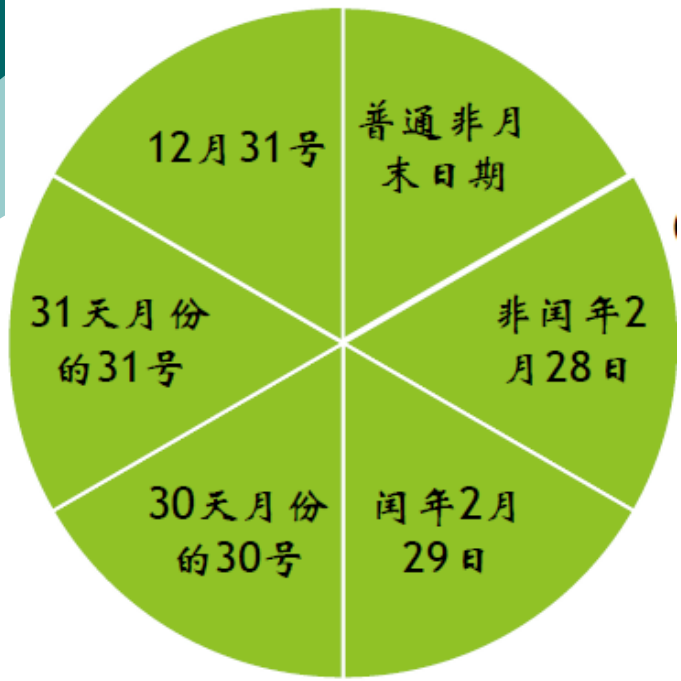
不可行路径如何产生？

- 判定结构体之间存在关联



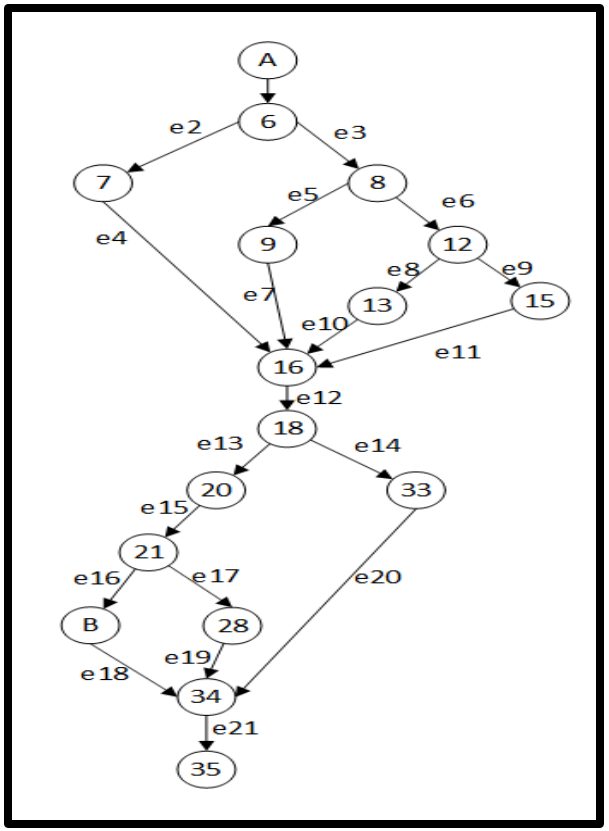
路径覆盖（续）

开发过程中如何避免不可行路径？



需求

6条路径



程序实现

输出

路径覆盖（续）

开发过程中如何避免不可行路径？

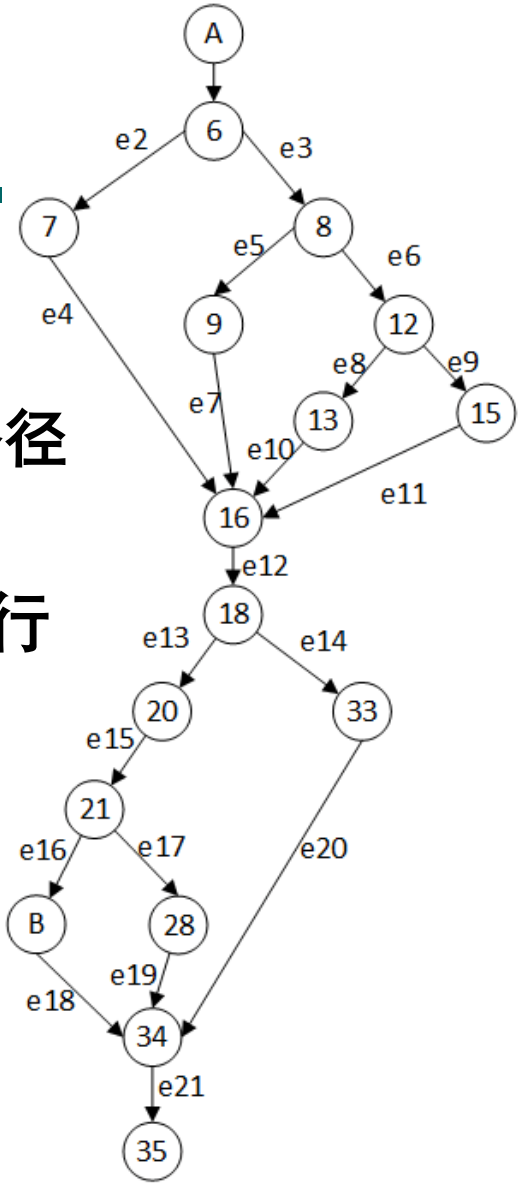
6条路径

- **普通非月末日期**
- **非闰年2月28日**
- **闰年2月29日**
- **30天月份的30号**
- **31天月份的31号**
- **12月31号**

需求与实现不一致

12条路径

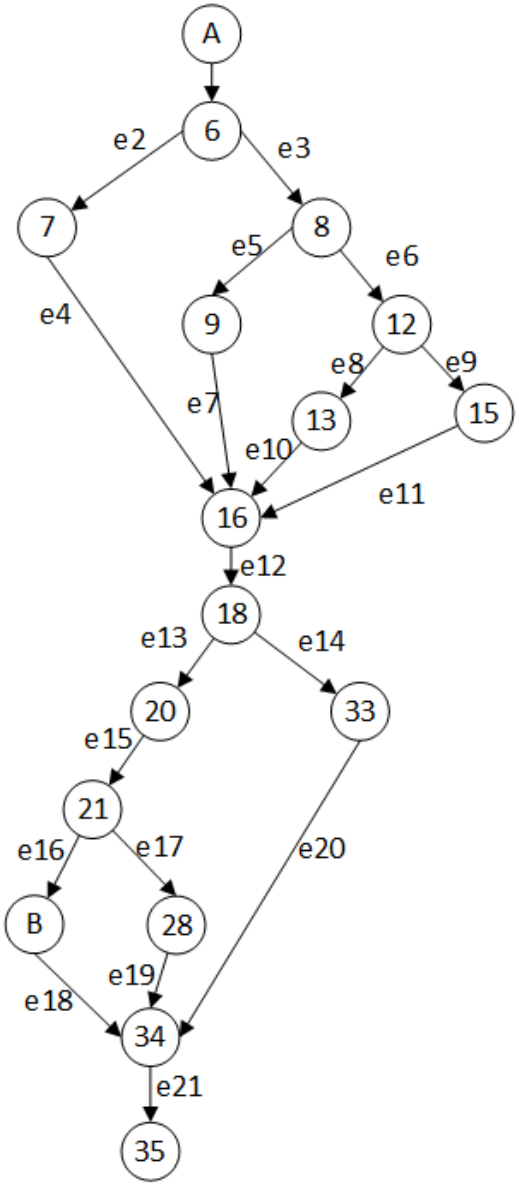
- **6条可行**
- **3条不可行**
- **3条冗余**



路径覆盖（续）

开发过程中如何避免不可行路径？

- 从测试的角度看待和分析需求
- 设计合理的程序结构



内容提要

- 4.1 白盒测试概述
- 4.2 控制流测试
- 4.3 数据流测试
- 4.4 程序插装
- 4.5 程序变异测试
- 4.6 小结

4.3 数据流测试

- **基本思想：设计测试用例使得程序中变量的定义和使用被覆盖。**
- **数据流测试相关的概念**
  - DEF(S)={x|语句S包含变量x的定义}
  - USE(S)={x|语句S包含变量x的使用}
  - 定义-使用对(x, S, S')：x ∈DEF(S), x ∈USE(S')，且程序中存在从S到S'的路径并且该路径中不包含x的其他定义

数据流测试(续)

- **数据流测试策略**
  - 全定义：要求覆盖每个变量的每个定义到该定义的至少一个使用
  - 全使用：要求覆盖每个变量的每个定义到该定义的每个使用
  - 全定义-使用：要求覆盖每个变量定义-使用对的每条路径

内容提要

- 4.1 白盒测试概述
- 4.2 控制流测试
- 4.3 数据流测试
- 4.4 程序插装
- 4.5 程序变异测试
- 4.6 小结

4.4 程序插装

- **程序插装（Program Instrumentation）**
  - 在程序中设置断点
  - 在程序中设置打印输出语句
  - 在特定部位插入记录动态特性的语句
- **目的**
  - 一方面检验测试结果数据
  - 另一方面借助插入语句给出的信息了解程序的动态执行情况

程序插装(续)

- **程序插装技术的研究涉及下列问题：**
  - 探测哪些信息？
  - 程序的什么位置设置探测点
  - 需要多少探测点

内容提要

- 4.1 白盒测试概述
- 4.2 控制流测试
- 4.3 数据流测试
- 4.4 程序插装
- 4.5 程序变异测试
- 4.6 小结

4.5 程序变异测试

- **变异测试：一种在细节方面改进程序源代码，针对某类特定错误而进行的软件测试方法**
- **变异操作**
  - 模拟典型应用错误，如使用错误的操作符或变量名字
  - 强制产生有效的测试，如使每个表达式都等于0

程序变异测试(续)

- **假设P在测试集T上是正确的，M＝{M（P）|M（P）是p的变异体}**
  - 若变异体M在T中每一元素上都存在错误
    - 可以认为源程序P的正确程度比较高
  - 若M在T中某些元素上不存在错误，则可能存在以下三种情况：
    - 这些变异体与源程序P在功能上是等价的。
    - 现有的测试数据不足以找出源程序P与气变异体的差别。
    - 源程序P可能含有错误，而某些变异体却可能是正确的。

程序变异测试(续)

- **变异测试的基本特征：通过变异算子对程序做一个较小的语法变动来产生一个变异体**
  - 程序员的能力假设
  - 简单的和复杂的程序设计错误组合效应假设

4.6小结

- **白盒测试的基本概念和技术**
  - 语句覆盖测试
  - 分支覆盖测试
  - 条件覆盖测试
  - 分支-条件覆盖测试
  - 条件组合覆盖测试
  - 路径覆盖测试