软件度量与质量分析

内容提要

- 没有测量，何来控制
  - 软件度量概述
- 软件度量，各有所长
  - 常用软件度量方法
  - 从分析结构入手--McCabe复杂度概述
- 代码结构的复杂程度
  - 圈复杂度
- 代码结构的良好程度
  - 基本复杂度
- 软件质量分析
  - McCabe复杂度与质量的关系
  - 结构化测试

没有测量，何来控制

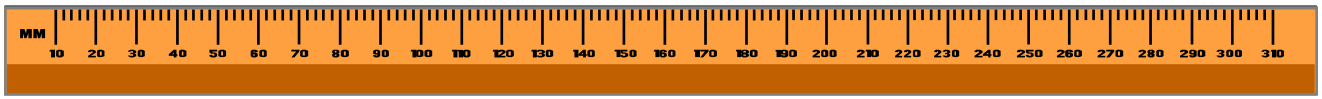
只有当你对你所谈到的东西进行测量，并用数量表示出来时，你就对它有了一定了解，反之，你对它并没有真正的了解 .

Lord Kelvin, 1889

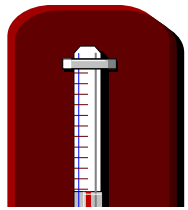
´You can‘t control what you can‘t measure.´

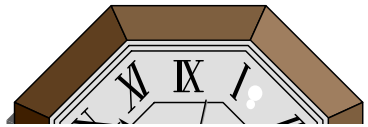
没有测量，何来控制

什么是度量（ Metric ）?



- 度量（Metric）
  - 测量、计算，用定量的方法描述一个实体的某一属性
  - 软件的度量
    - 和项目相关
    - 估算性测量
    - 复杂性度量





没有测量，何来控制

- **软件代码度量**
  - 软件是特殊产品，难以度量
  - 通过度量了解质量特性
- **软件质量属性**
  - 正确性
  - 健壮性
  - 可靠性
  - 性能
  - 易用性
  - 安全性
  - 可扩展性
  - 兼容性
  - 可移植性
  - …
- **度量的好处**
  - 从定性到定量
  - 质量分析、控制
  - 为进一步的分析、评估、

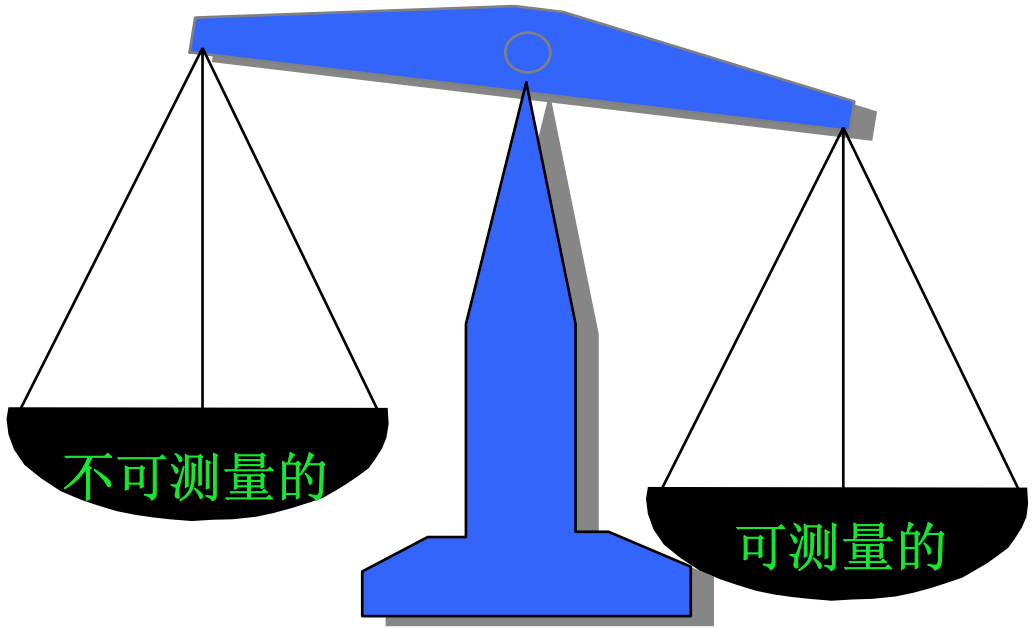
没有测量，何来控制

- 度量要面对的问题
  - What do we want to measure?
    我们要测量什么？
  - How do we measure?
    我们应如何测量？
  - What do the measurements mea
    测量的意义何在？

软件度量与质量分析

没有测量，何来控制

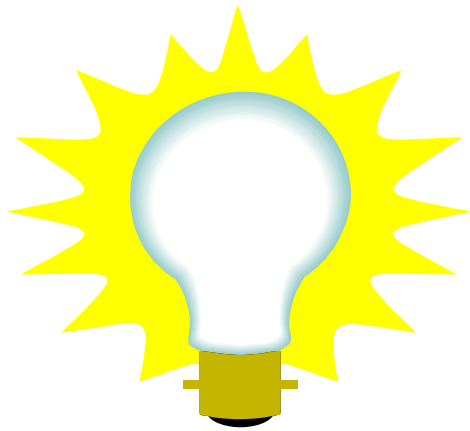
- 测什么？
  - 功能、性能
  - 质量属性
  - 复杂程度
  - 结构
  - 物理尺寸
  - 模块、路径
  - …



- 如何测？
  - 并非所有的属性都是可直接测量的
  - 简单、易实现、直观、数字化
- 意义？
  - 并非可测量的都有意义
  - 好的度量如何评价

没有测量，何来控制

- 何谓好的度量
  - 好的度量必须：
    - 直观
    - 客观
    - 语言独立
  - 好的度量应该：
    - 和错误的出现有直接关联
    - 能反映测试的工作量
    - 自动化
    - 简单
- 不少软件度量方法都是针对复杂性的



软件度量，各有所长

- 软件产品的复杂性
  - 文本复杂性
  - 逻辑结构复杂性
  - 功能体系复杂性
  - …
- 常用软件度量
  - Line Count Metrics
    行数统计度量
  - Halstead Metrics
    Halstead 度量
  - Function Point
    功能点
  - McCabe Metrics

行数度量（LOC）

- 起源
  - 测量物体的尺寸是研究它的开始
  - 最简单、直接的一种方法
- 技术
  - 对模块代码行进行分类统计，提供代码行、注释行、空白行、混合行
  - 提供代码的总额，但不关注内容和结构
  - LOC复杂度：
    - blanks（空白行）：空行（只有空格、Tab），没有其他字符
    - code（代码行）：纯代码行，不含注释
    - comments（注释行）：纯
    - mixed（混合行）：既有代

行数度量（LOC）

- 优势
  - 软件物理规模的直接度量
  - 易于理解
  - 计算简单
  - 语言独立
  - 定义了不同类别的代码行（如注释行和空行），有利于进一步分析、引用
  - 协助指出难以理解的模块
    - 注释行通常增加了可读性，但不寻常的大量注释意味着模块难以理解

Halstead 度量

- 起源
  - 1977年，由Maurice Halstead 提出
  - 计算模块的操作数和操作符（运算符），直接得出模块的复杂程度的一种定量测试方法
  - 通过分析源代码对模块进行度量，重点是复杂性的计算
  - 应用于代码，经常作为一种维护性度量。也可作为对难度、工作量的一种评估
  - 自诞生之日起，就充满争议

Halstead 度量

- 技术
  - Halstead度量主要基于4种源自程序代码的统计数值：
    - n1 = 独立的操作符的数量（操作符的种类数）
    - n2 = 独立的操作数的数量（操作数的种类数）
    - N1 = 操作符总数
    - N2 = 操作数总数
  - 通过以上数据，可以计算5种复杂度：

复杂度	*符号*	*公式*
程序长度（Program length）	N	N= N1 + N2
词汇量（Program vocabulary）	n	n= n1 + n2
容量（Volume）	V	V= N * (Log_2 n)
难度（Difficulty）	D	D= (n1/2) * (N2/n2)
工作量（Effort）		

  - 错误数预测：
    - Error = V/3000 = N * (Log_2 n

Halstead 度量

- 优势
  - 不必深入分析程序的结构
  - 计算简单
  - 适用于任何编程语言
  - 预测错误/缺陷的数目
  - 预测测试的工作量
  - 预测维护的工作量
  - 有利于项目规划
  - 对整个开发过程有帮助
  - 经过很多机构的使用和研究，表明其[illegible]复杂度对预测开发工作量和平均缺陷

软件度量，各有所长

功能点

- 起源
  - 1979 由 Allan.Albrecht 开发
  - 从系统的需求和设计出发，对软件的功能进行分类、统计、分析
  - 度量软件的规模和生产力
  - 和软件完成的功能紧密相关
  - 较广泛的认可
  - 国际功能点组织、标准化活动
  - ISO/IEC标准

软件度量，各有所长

功能点

- 技术
  - 功能点
    - 就是最终用户的业务功能, 比如对输入的查询
  - 用户业务功能需求：
    - 数据功能
      - 内部逻辑数据
      - 外部接口数据
    - 事务功能
      - 外部输入
      - 外部输出
      - 外部查询
  - 再配以不同的复杂性，确
  - 计算调整系数，计算功能

软件度量，各有所长

功能点

- 优势
  - 对以下情况，功能点度量通常被公认为是一种有效方法：
    - 评估软件工程的大小（以及周期）
    - 建立每小时功能点的生产力
    - 评估对需求的支持
    - 评估系统变更的开销
    - 软件模块比较的标准化
  - 唯一与软件功能相关的度量方法
  - 语言独立
  - 较大的用户群：
    - International Function Point Users' Group (IFPUG，国际功能点用户组织)
    - 多于 1,200 会员公司
  - IFPUG提供功能点实践手

McCabe度量

- 起源
  - 1976年，由 Thomas McCabe 提出
    - 1976年发表《软件复杂度》的论文
    - 1982年发表论文《结构测试：使用圈复杂度的软件测试方法》
    - McCabe测试技术被美国国家标准技术学会（NIST）采用
  - 从分析结构入手，计算模块/程序的复杂程度
  - 提供了比较两个程序复杂度的一种简单、客观的指标
  - 广泛的认可
  - McCabe复杂度又称：
    - 程序复杂度
    - 循环复杂度（Cyclomatic C
    - 圈复杂度

软件度量，各有所长

McCabe度量

- 技术
  - McCabe复杂度是基于对软件结构进行严格的算术分析得来，其本质上是对程序拓扑结构复杂性的度量
  - 对模块结构的复杂程度进行量化，并以该复杂度为基础分析基本复杂度等其他复杂度
  - McCabe复杂度分为模块、类、程序3个层面
  - McCabe复杂度种类：
    - 圈复杂度
    - 基本复杂度
    - 模块设计复杂度
    - 设计复杂度
    - 集成复杂度
    - 数据复杂度
    - ...

软件度量，各有所长

McCabe度量

- 优势
  - 基于软件结构的严格的数学分析
  - 反映代码的质量
  - 严谨、客观
  - 易于理解
  - 和程序语言无关
  - 广泛的认可
  - 与其他度量互补
  - 可扩展或派生出更多的复杂度
  - 应用优势

McCabe度量

- 优势
  - McCabe复杂度的应用
    - 代码质量分析
    - 作为动态测试的指导
    - 代码开发/维护的风险分析
    - 作为项目开发和管理指南
    - 再工程

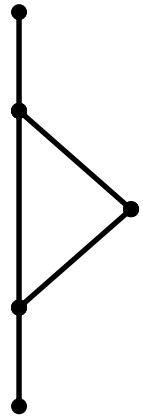
McCabe度量

- 预备知识
  - 结构流图
    - 一种描述软件模块逻辑的结构图，类似于流程图
    - 通过源代码建立
    - 是模块判逻辑结构的可视化
  - 流图符号
    -  （箭头）称为“边”（edge），代表控制流
    -  （圆圈或圆点）称为“节点”（node），代表一个或多个语句动作
    - 由节点和边围成的范围称为“域”（region），在计算域时，图形外的区域也应算作一个“域”
    - 判定节点，指包含条件的支数

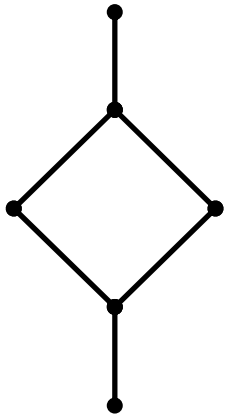
软件度量，各有所长

McCabe度量

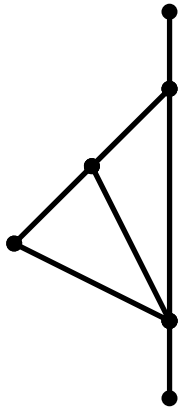
- C常用控制结构流图



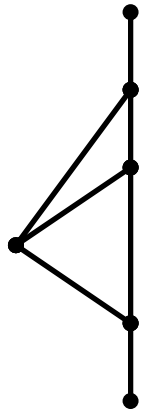
If .. then



If .. then .. else

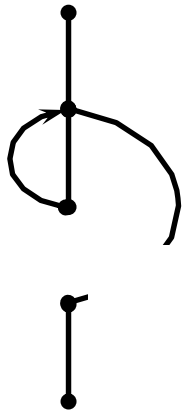


If .. and .. then

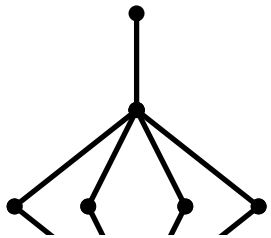


If .. or .. then

Do .. While



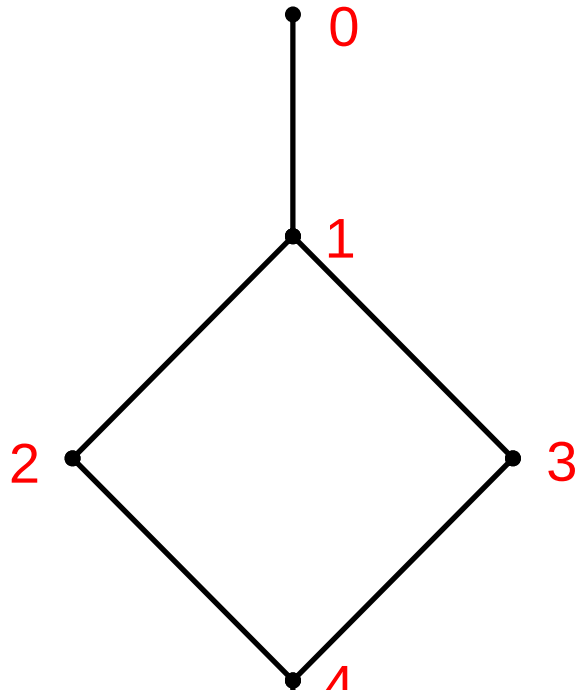
While ..



McCabe度量

- 结构流图举例1

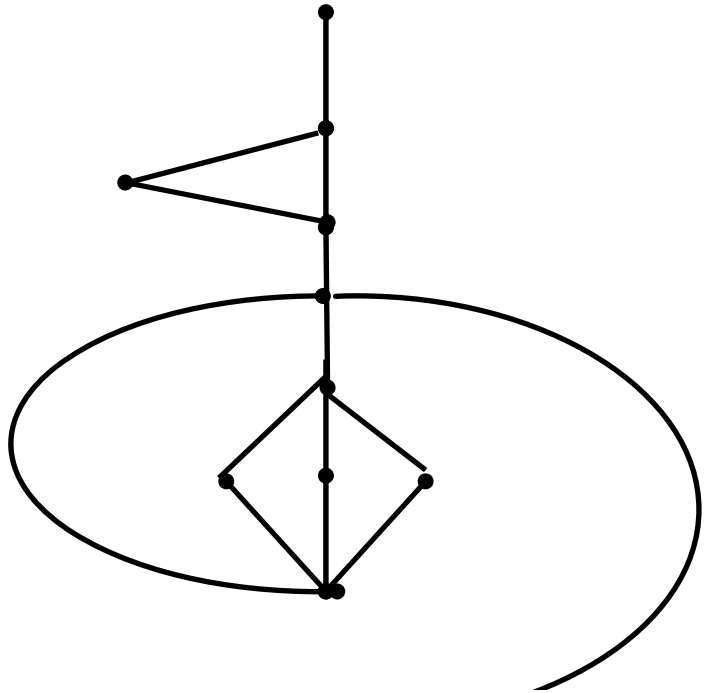
```
          function_test(y)
0         {
          x=3;
1         if ( y < 4)
2             x=sin(y);
          else
3             x=cos(y);
4         x=x*x;
5         }
```



McCabe度量

- 结构流图举例2

```
1     IF    condition
2           statement
3     ENDIF
4     statement
5     WHILE condition
6           DO SWITCH
7                 Label1
8                       statement
9                 Label 2
10                      statement
11                Default
12                      statement
13          END SWITCH
14    END WHILE
```



代码结构的复杂程度

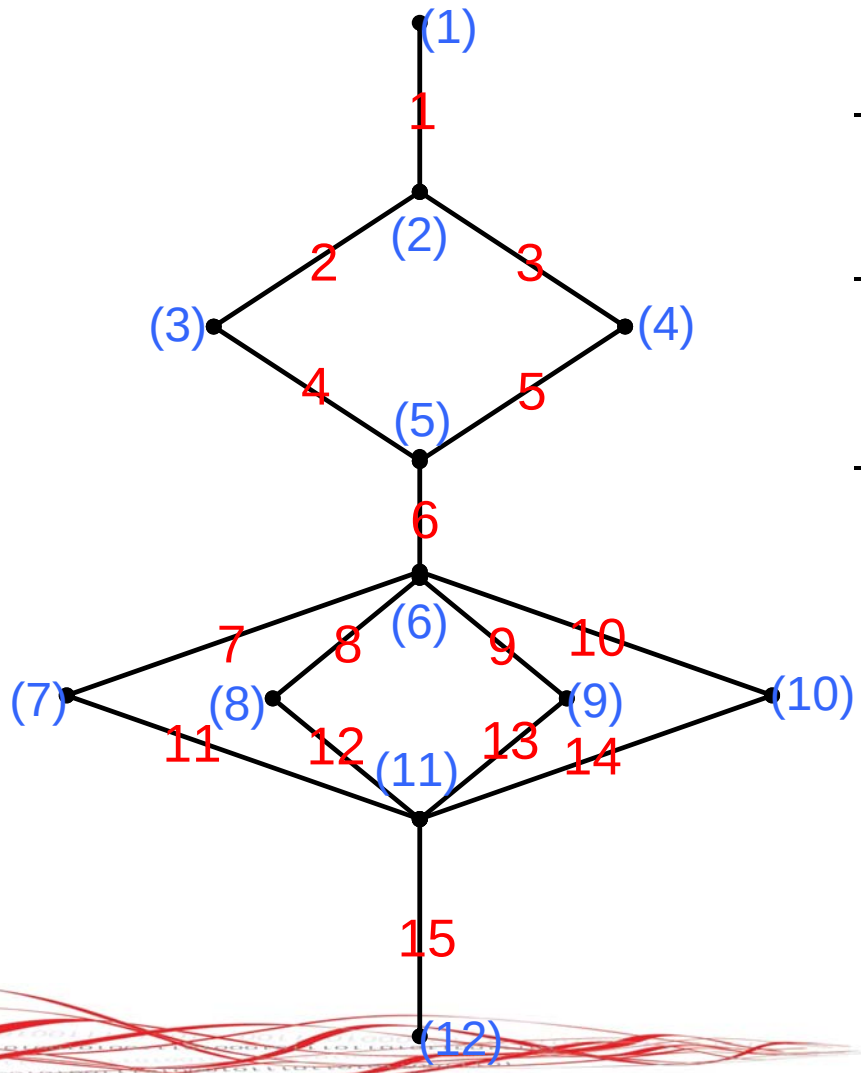
圈复杂度

- 圈复杂度（Cyclomatic complexity）
  - 是模块结构复杂程度的一种度量
  - 定义为贯穿模块的独立线行路径数
  - 也是保证测试充分所需的最小测试路径数
  - 缩写为 v(G)
  - 圈复杂度高，表明模块复杂程度高，不易理解
  - 经验研究表明，圈复杂度与模块错误之间存在较强的关联性
- 计算方法
  - 公式
  - 断言（判定）
  - 区域（数圈法）

代码结构的复杂程度

圈复杂度

- 公式法



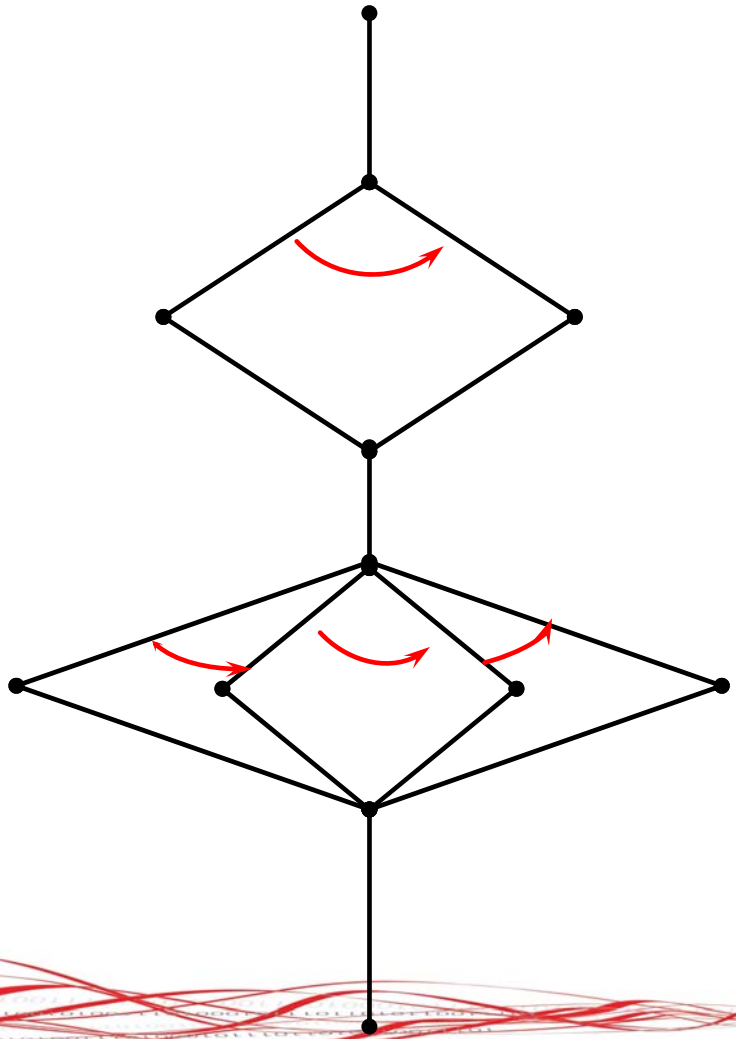
- 计算流图中所有的边数（e）和节点数（n）
- 则：v(G) = e – n + 2

- 举例：
  - e = 15
  - n = 12
  - v(G) = e – n + 2

代码结构的复杂程度

圈复杂度

- 断言法



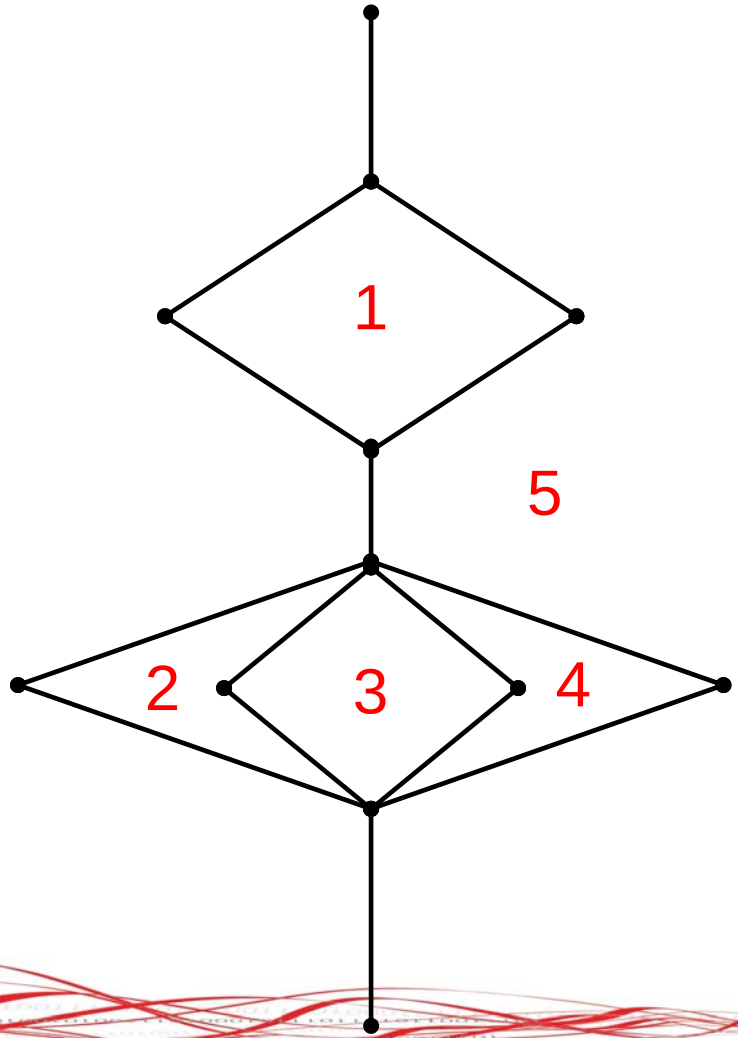
- 计算流图中所有的判定数（断言数）p
- 则：v(G) = p + 1

- 举例：
  - p = 4
  - v(G) = p + 1
    = 4 +1

代码结构的复杂程度

圈复杂度

- 区域法

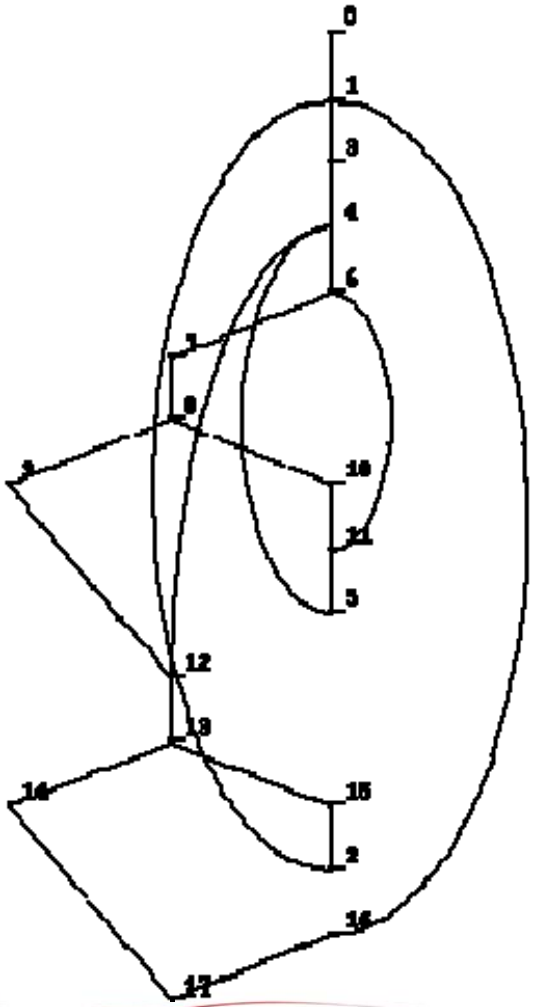


- 计算所有的由边和节点所围成的区域数（圈的个数）R
  - 注意圈与圈之间要线性独立
  - 最外围也算一个圈
- 则：v(G) = R
- 举例：
  - R = 5

代码结构的复杂程度

圈复杂度

- 练习



– 图中

e = 22

n = 18

– 所以

v(G) = e - n +2

= 22 - 18 +2

= 6

代码结构的复杂程度

圈复杂度

- 优势
  - 量化了模块逻辑的复杂程度
  - 反映了模块出错的可能性
  - 指出极复杂的、需要进一步分解的模块
  - 指导测试：
    - 圈复杂度 = 基本路径数 = 最小测试数
    - 帮助制定测试计划，确定测试重点
  - 帮助管理测试和维护的资源
  - 独立于编程语言
  - 客观、易理解

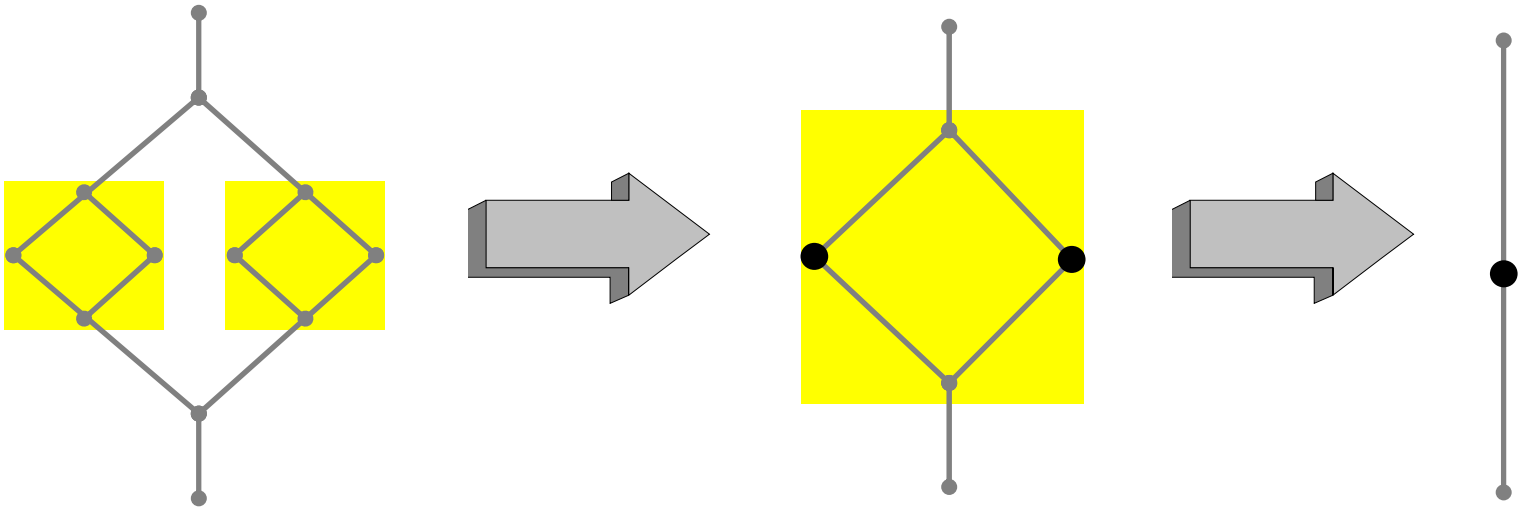
基本复杂度

- 基本复杂度（Essential Complexity）
  - 模块“非结构化”程度的定量描述，即模块逻辑结构的“良好”程度
  - 基本复杂度是模块按结构化方法简化后的圈复杂度
  - 缩写为ev(G)
  - 基本复杂度高，表明模块的结构“不够良好”
    - 代码质量下降
    - 维护工作加重
    - 模块分割困难
    - 在维护时修改一个错误经常引入其他错误
- 计算方法
  - 将模块结构流图按结构化的部分）
  - 再计算简化后的圈复杂度

代码结构的良好程度

基本复杂度

- 结构化简化



圈复杂度 = 4
v(G) = 4

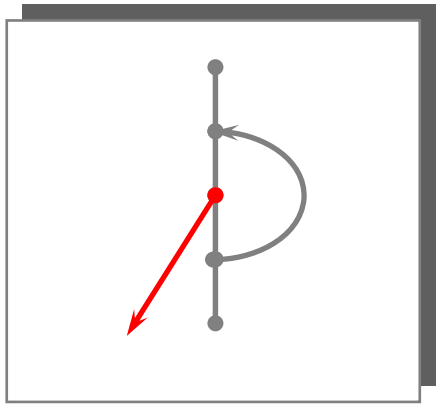
基本复杂度 = 1
ev(G) = 1

McCabe 基本
去掉结构化部分

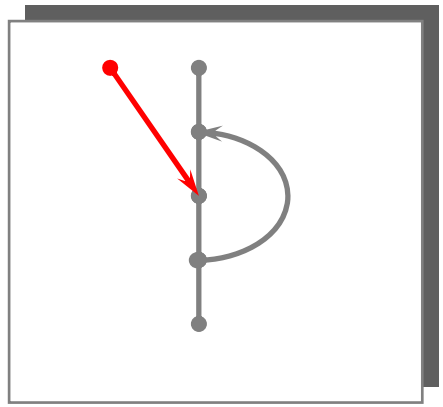
代码结构的良好程度

基本复杂度

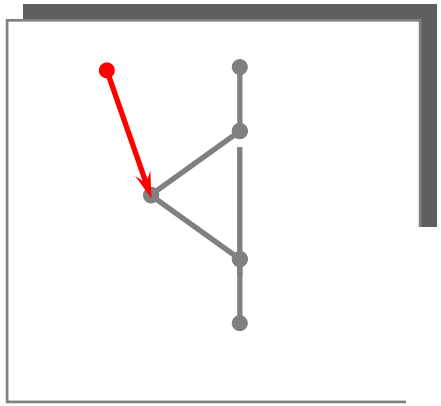
- 非结构化逻辑举例



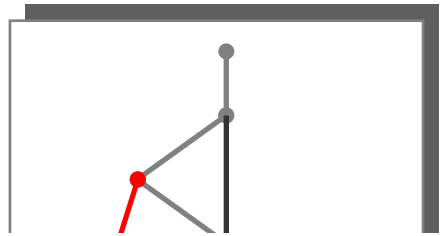
Branching out of a loop



Branching into a loop



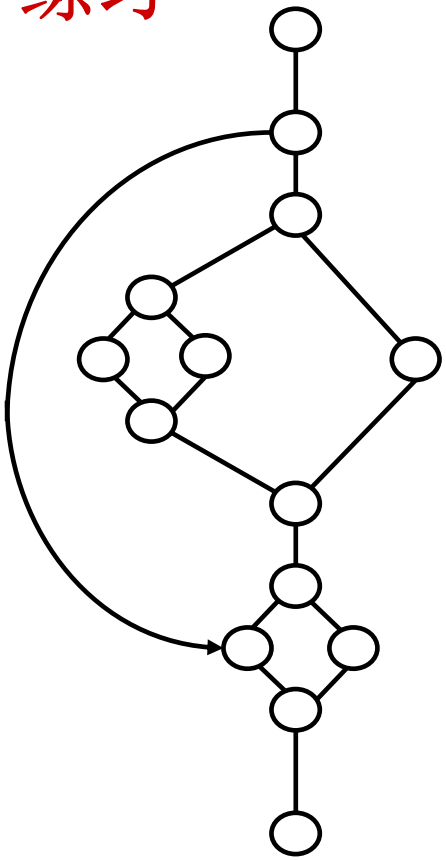
Branching into a decis



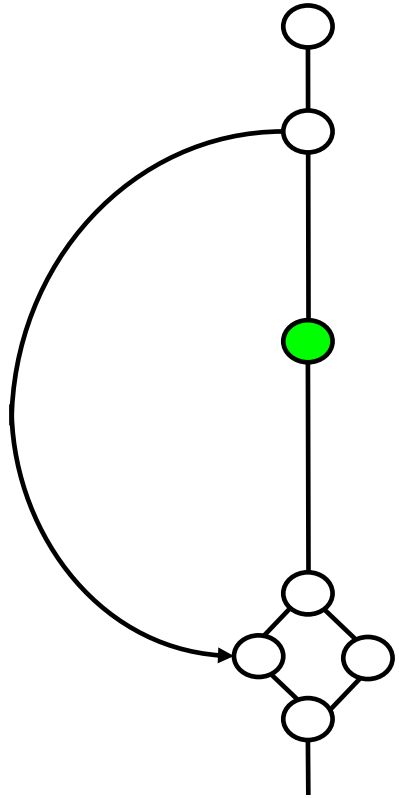
代码结构的良好程度

基本复杂度

- 练习



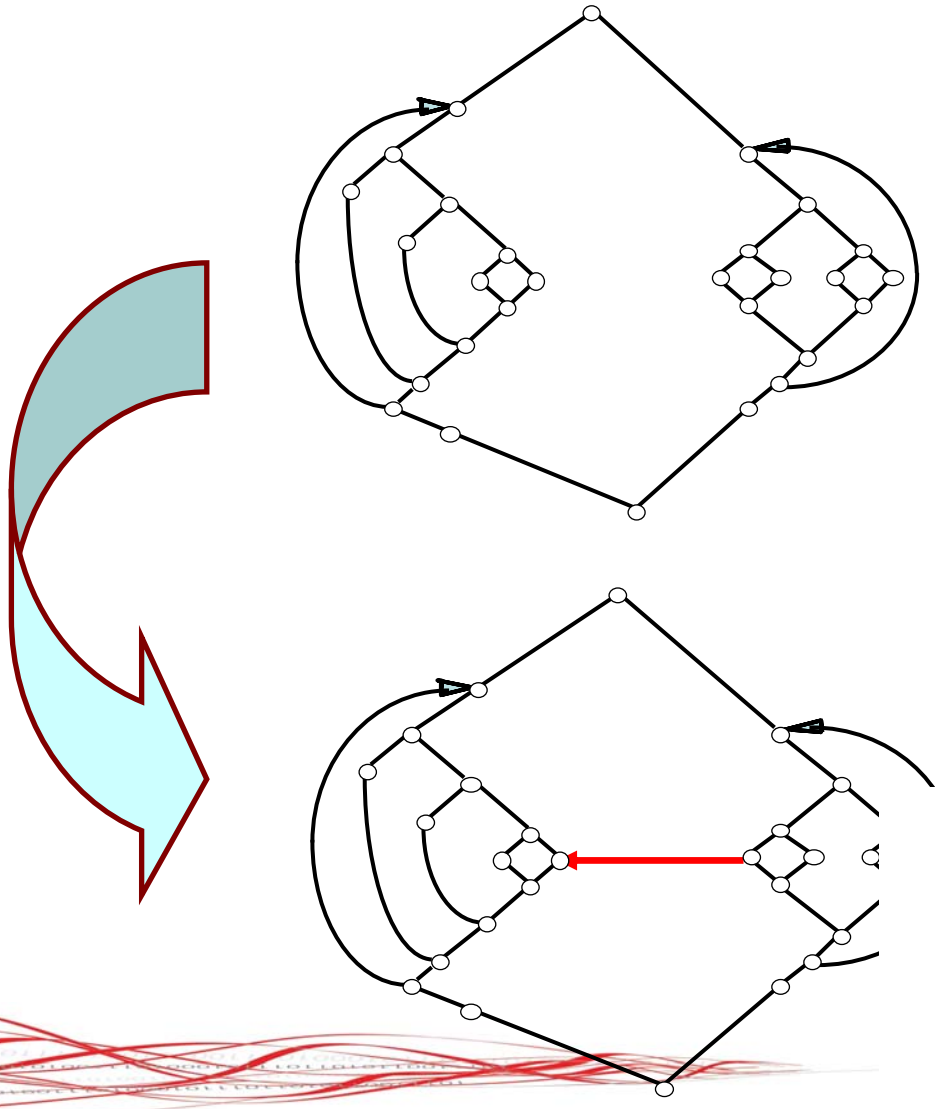
v(G) = 5



代码结构的良好程度

基本复杂度

- 基本复杂度可以帮助我们发现非结构化代码



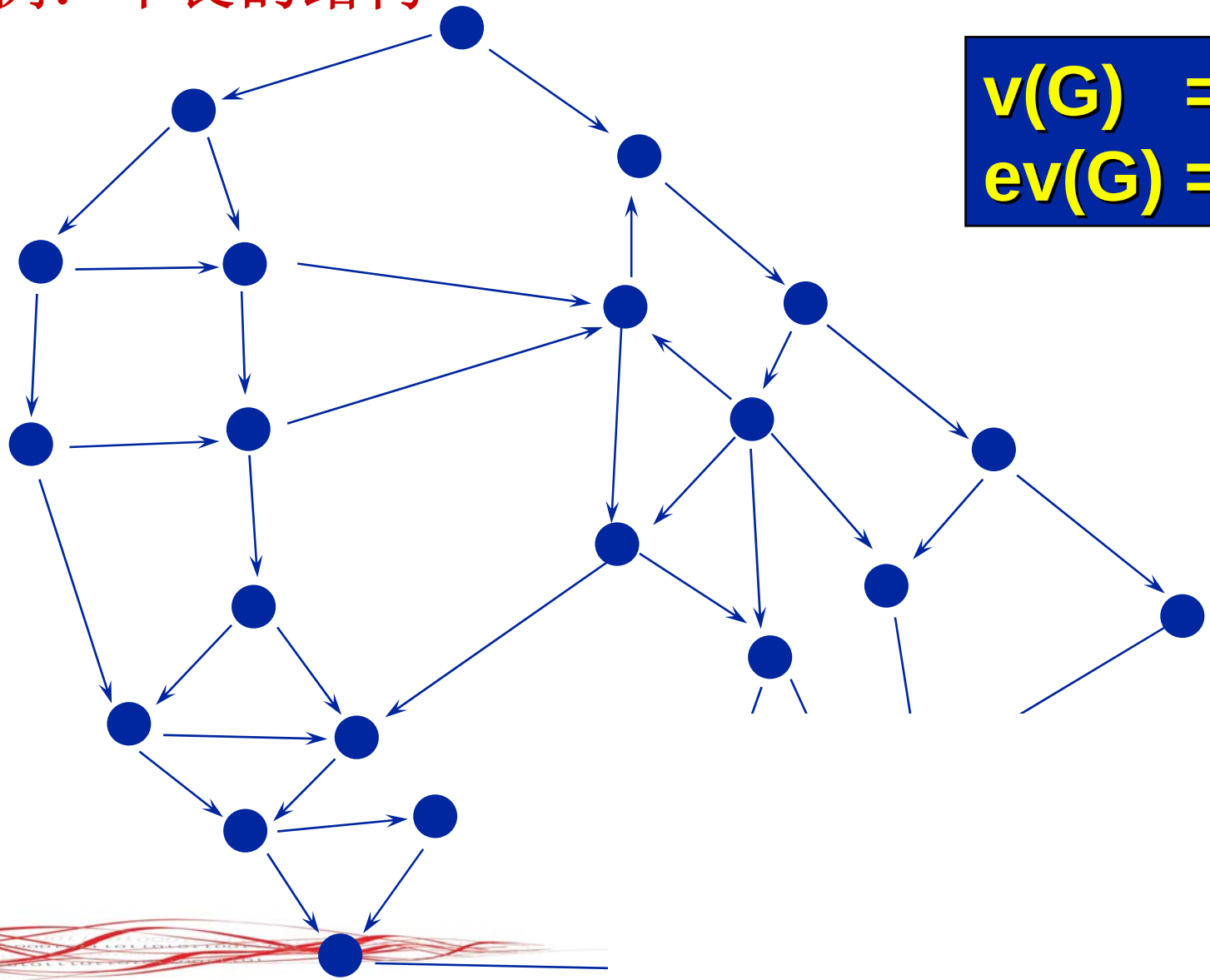
v(G) = 10
ev(G) = 1

好的设计

代码结构的良好程度

基本复杂度

- 举例：不良的结构

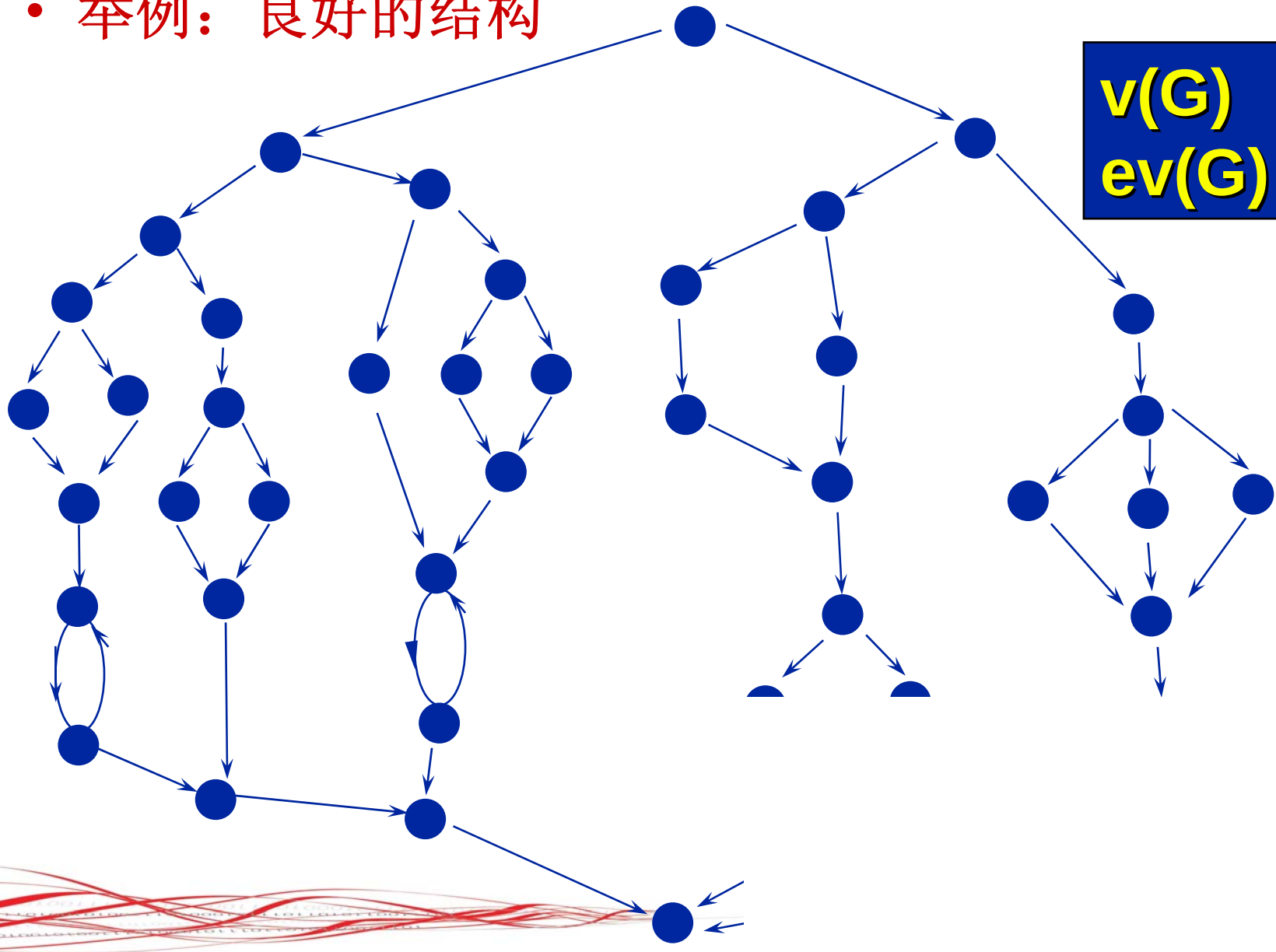


v(G) = 18
ev(G) = 17

代码结构的良好程度

基本复杂度

- 举例：良好的结构



v(G) = 18
ev(G) = 1

基本复杂度

- 优势
  - 量化了一个模块逻辑结构的“良好”程度，即“结构化”程度：
    - 当基本复杂度为1，这个模块是充分结构化的；
    - 当基本复杂度大于1 而小于圈复杂度，这个模块是部分结构化的；
    - 当基本复杂度等于圈复杂度，这个模块是完全非结构化的。
  - 揭示代码的质量
  - 预测维护代码和分解代码所需的工作量
  - 适用于任何编程语言

模块设计复杂度

- 模块设计复杂度（Module Design Complexity）
  - iv(G)
    - 模块设计复杂度是一个模块调用其他模块的结构复杂程度
    - 量化了一个模块和相关模块集成时的测试工作量
    - 模块设计复杂度高意味着：
      - 控制耦合程度高。对隔离、维护、代码重用不利
    - 计算方法：
      - 去除那些不影响子模块调用控制的判定、循环、和节点，再计算圈复杂度

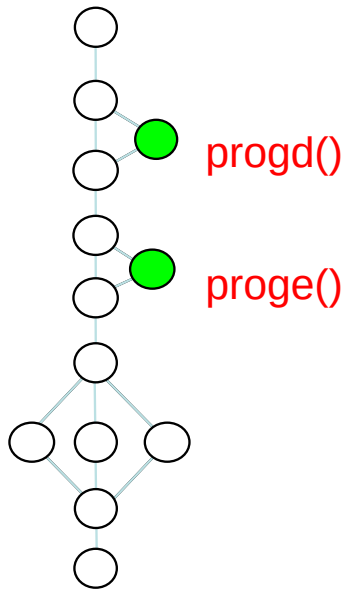
其他McCabe复杂度

模块设计复杂度

例子… iv(G)

```
main()
{
    if (a == b) progd();
    if (m == n) proge();
    switch(expression)
    {
    case value_1:
        statement1;
        break;
    case value_2:
        statement2;
        break;
    case value_3:
        statement3;
    }
}
```

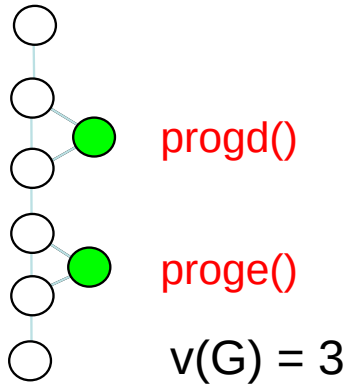
Main结构流图



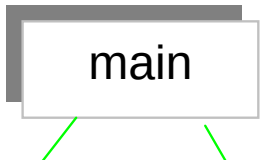
v(G) = 5

简化

简化后流图



v(G) = 3



所以，

iv(G) = 3

设计复杂度

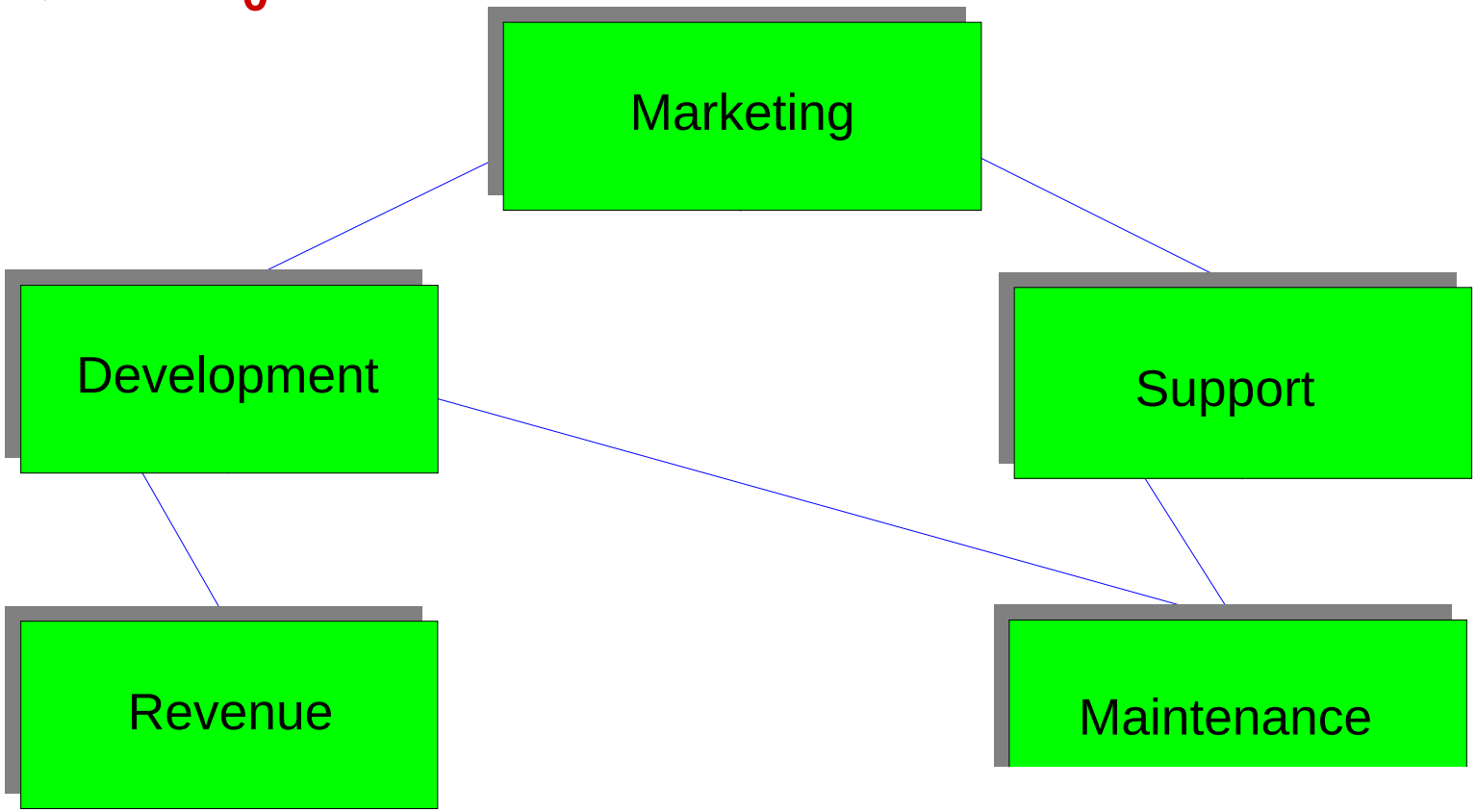
- 系统设计复杂度（Design Complexity）
  - S_0
    - 系统各模块间的相互影响的定量描述
    - 代表了“自底向上”集成测试所需的工作量
    - 提供了全面评价系统设计规格和复杂程度的数据，但不反映独立模块的内部情况。
    - 设计复杂度高的系统意味着系统各部分之间有着复杂的相互关系，这样系统将难以维护，不利于集成
    - 计算方法：
      - 系统各组成部分模块设计复杂度之和：

$$S_0 = \sum i$$

其他McCabe复杂度

设计复杂度

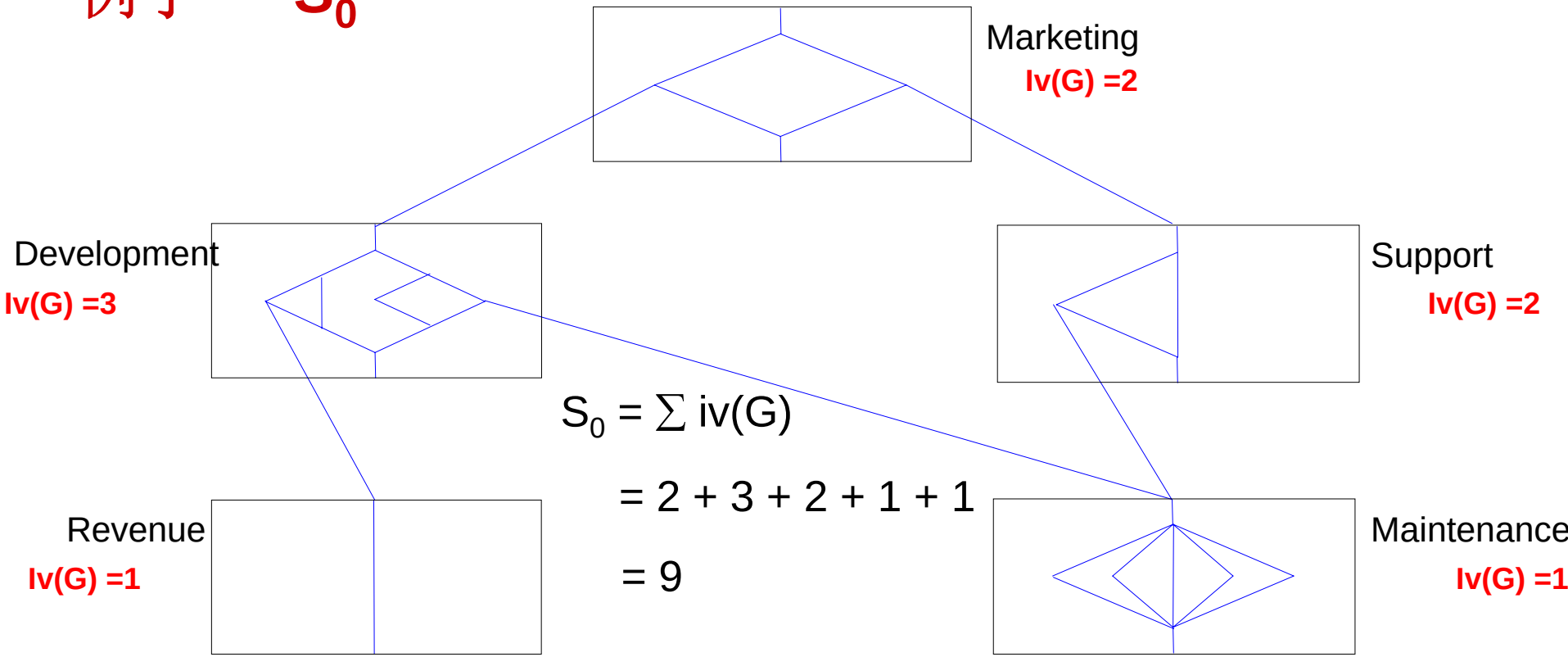
例子… S_0



其他McCabe复杂度

设计复杂度

例子… S_0



$S_0 = \sum iv(G)$

$= 2 + 3 + 2 + 1 + 1$

$= 9$

系统设计复杂度 S_0 = 各模

集成复杂度

- 集成复杂度（Integration Complexity）
  - S_1
    - 系统集成所必须的集成测试的数量
    - 代表了集成测试的工作量，也反映了系统设计的复杂成度
    - 有助于集成测试的计划和实施
    - 计算方法

$$S_1 = S_0 - n + 1$$

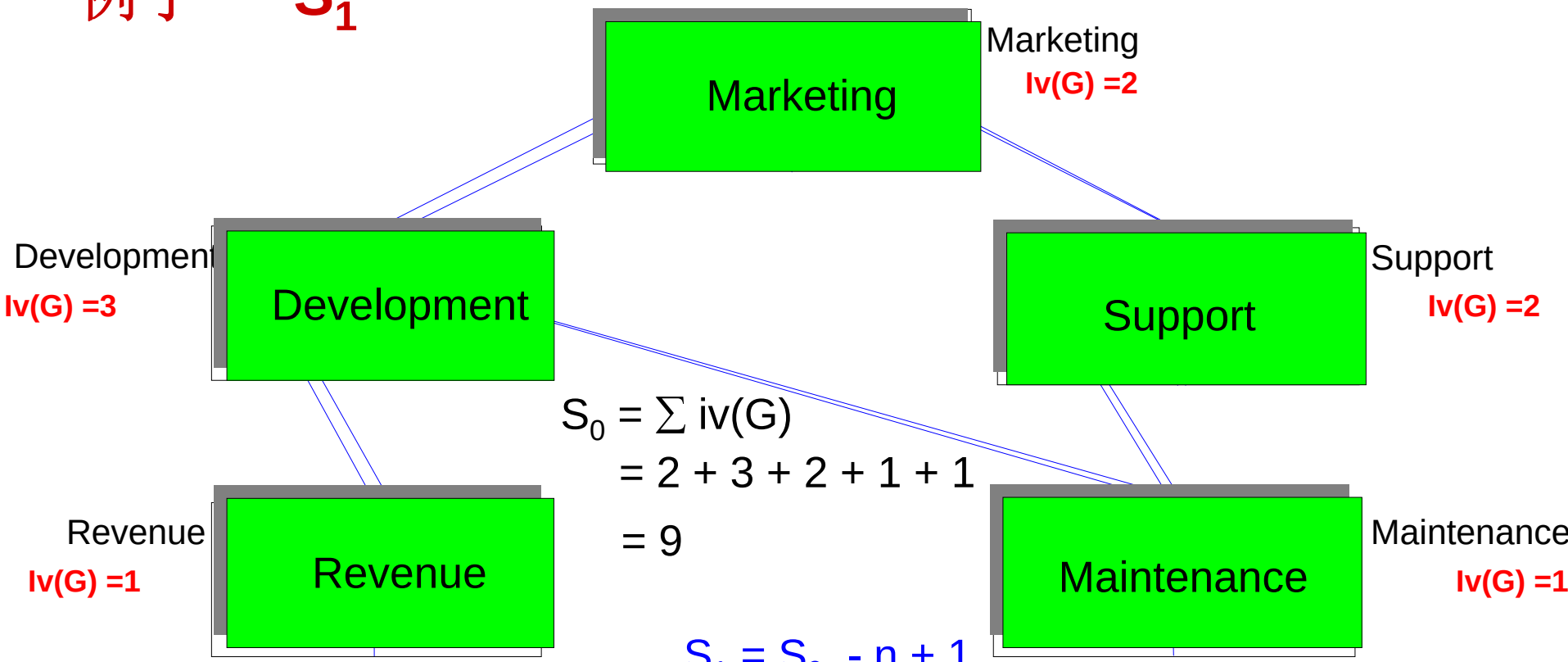
其中 S_0：系统设计复杂度

n： 系统中模块

其他McCabe复杂度

集成复杂度

例子… S_1



$S_0 = \sum iv(G)$

$= 2 + 3 + 2 + 1 + 1$

$= 9$

$S_1 = S_0 - n + 1$

$S_1 = ?$

$S_1 = ?$

全局数据复杂度

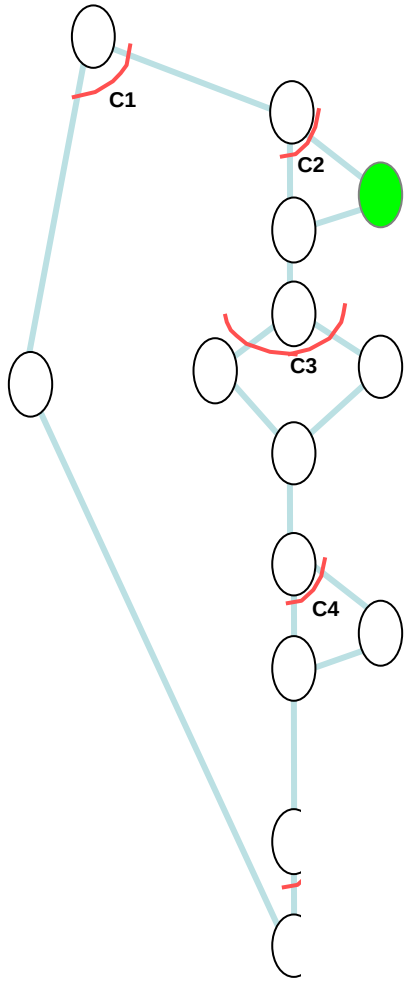
- 全局数据复杂度（Global Data Complexity）
  - gdv(G)
    - 模块中与全局数据、参量相关联的结构复杂性的一种定量描述。
    - 表明模块对外部数据的依赖程度
    - 评估与全局数据有关的测试工作量
    - 也可衡量每个模块对系统数据耦合的影响
    - 计算方法：
      - 将所有不包含全局变量和参数的判定和循环都去掉
      - 再计算此时的圈复杂度就得
      - 全局数据复杂度不大于原流

其他McCabe复杂度

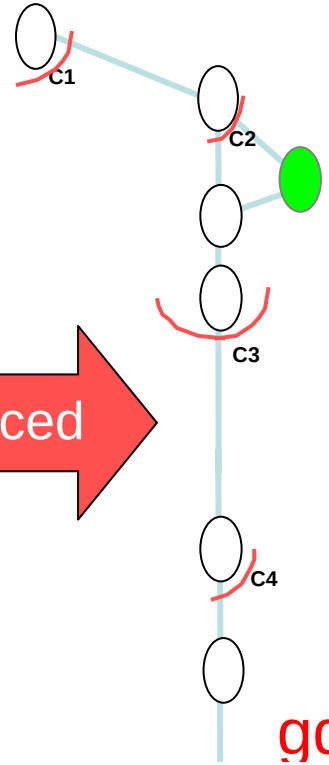
全局数据复杂度

例子… gdv(G)

```
subroutine(x)
{
    if (c1)
    {
        if (c2)
                x = y + n;
        if (c3)
                 z = y + m;
        else
                printf
(message1);
        if (z > 10)
                printf
(message2);
                flaga = 1;
        if (c5)
                moda(x);
    }
    else
    {
        printf (message3);
    }
}
```



Reduced



$gdv(G)_x = 3$

指定数据复杂度

- 指定数据复杂度（Specified Data Complexity）
  - sdv(G)
    - 模块中与用户数据相关联的结构复杂度的一种定量描述
    - 表明一个模块与指定数据相关联的数据复杂程度
    - 等于（用来测试指定数据所需的）基本测试路径数
    - 可分析数据结构变化对软件的影响
    - 计算方法（同全局数据复杂度）：
      - 将所有不包含指定数据元素的判定和循环都去除
      - 再计算此时的圈复杂度就得到指定数据复杂度

软件质量分析

- 什么是软件质量
  - ISO 9126标准
    - 软件质量：

      与软件产品满足规定的和隐含的需求的能力有关的特征或特性的集合
    - 软件质量特性：

      软件质量特性是软件产品的一组属性，用来对软件的能力进行描述和评估

      软件质量特性可细分为多层次的子特性
    - 软件质量度量

      用于确定软件产品质量特性的定量的计量和分析方法

      因此软件复杂度是软件质

什么是软件质量

- 从技术上讲，软件质量是：
  - 较少的错误和缺陷
  - 易于理解
  - 易于维护/变更
  - 易于测试
  - 易于实现
  - …
- 影响软件质量的因素
  - 时间
  - 预算
  - 人员的变化
  - 需求的改变
  - 缺陷排除
  - 测试

McCabe复杂度与软件质量的关系

- “简单就是可靠”
  - 硬件的可靠性设计的原则之一。这个原则显然也适合于软件。
  - “结构越复杂，就越容易出错”
  - McCabe复杂度，特别是圈复杂度和基本复杂度能购客观地反映软件的质量：
    - 圈复杂度：可靠性
    - 基本复杂度：可维护性
  - 研究和经验显示，圈复杂度与错误发生的概率密切相关
  - McCabe复杂度不仅可以度量软件的质量，更重要的是它指出了改进软件质量的有效途径
    - 基本路径测试
    - 质量趋势
    - 重点难点
    - 资源分配
    - ...

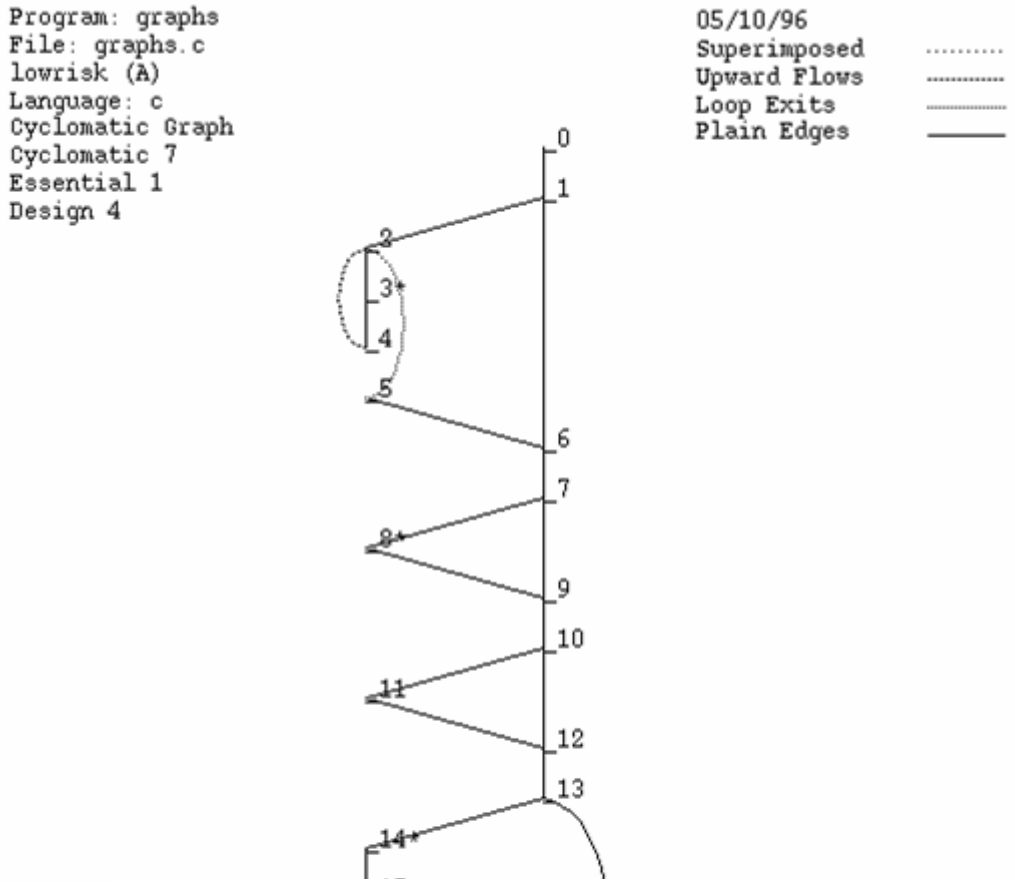
McCabe复杂度与软件质量的关系

- McCabe复杂度与软件质量的关系

McCabe复杂度	质量因素/工程因素
圈复杂度	可靠性、可读性、测试工作量（基本路径）
基本复杂度	可维护性、结构化、再工程工作量
模块设计复杂度	模块耦合性、隔离测试的工作量
系统设计复杂度	系统复杂性、模块调用复杂性
集成复杂度	集成工作量
全局数据复杂度	全局数据耦合性、与全局数据有关的测试量
指定数据复杂度	指定数据耦合

McCabe复杂度与软件质量的关系

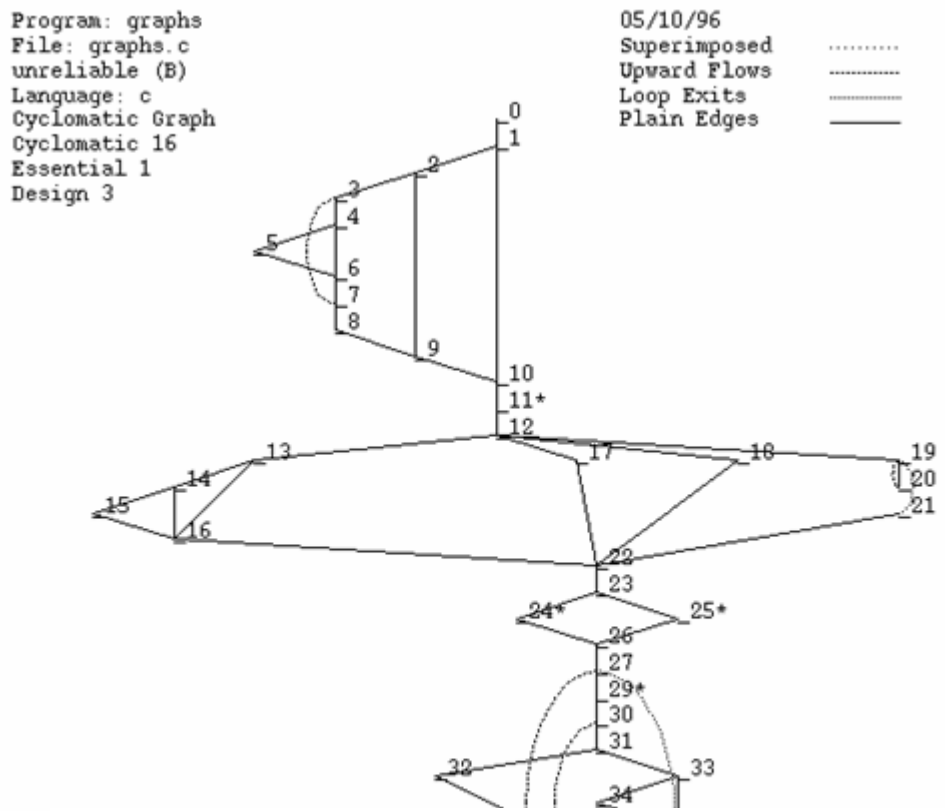
- 低复杂度模块
  - 圈复杂度 = 7
  - 基本复杂度 = 1
  - 模块设计复杂度 = 4



  - 可靠
    - 逻辑简单
    - 不易出错
    - 易于测试
  - 可维护
    - 结构良好
    - 易于理解
    - 易于修改

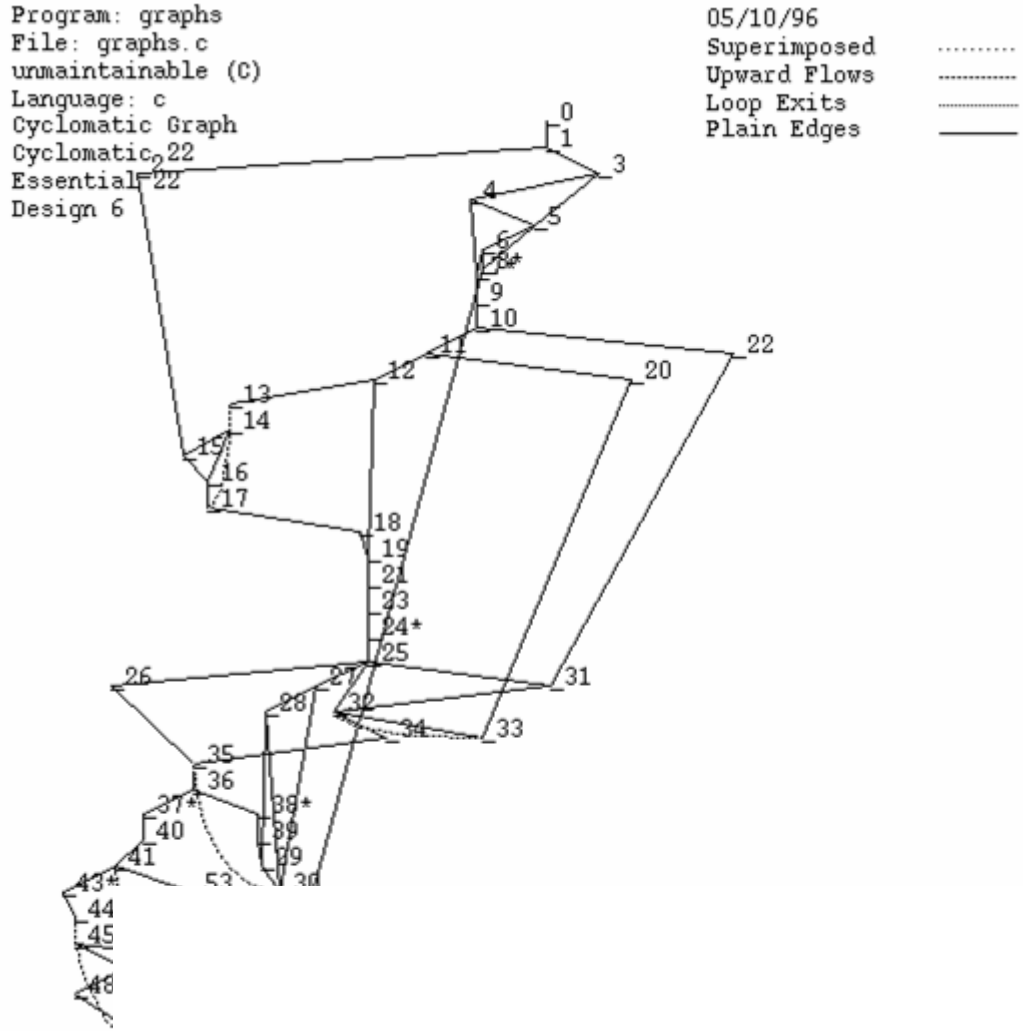
McCabe复杂度与软件质量的关系

- 适度复杂度模块
  - 圈复杂度 = 16
  - 基本复杂度 = 1
  - 模块设计复杂度 = 3
  - 不太可靠
    - 逻辑复杂
    - 易于出错
    - 测试困难
  - 可维护
    - 能够理解
    - 能够修改
    - 复杂度可以降低



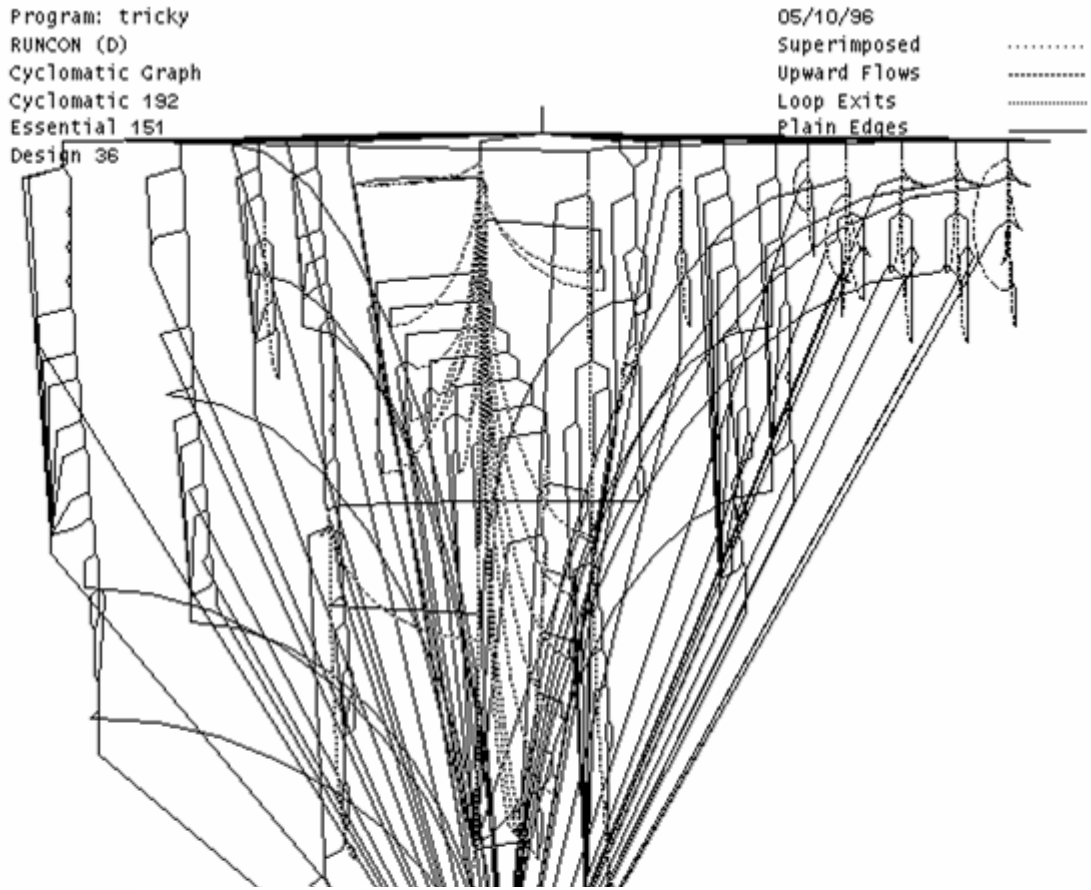
McCabe复杂度与软件质量的关系

- 高复杂度模块
  - 圈复杂度 = 22
  - 基本复杂度 = 22
  - 模块设计复杂度 = 6
  - 不可靠
    - 逻辑太复杂
    - 易于出错
    - 测试非常困难
  - 难以维护
    - 难以理解
    - 难以修改
    - 复杂度难以降低



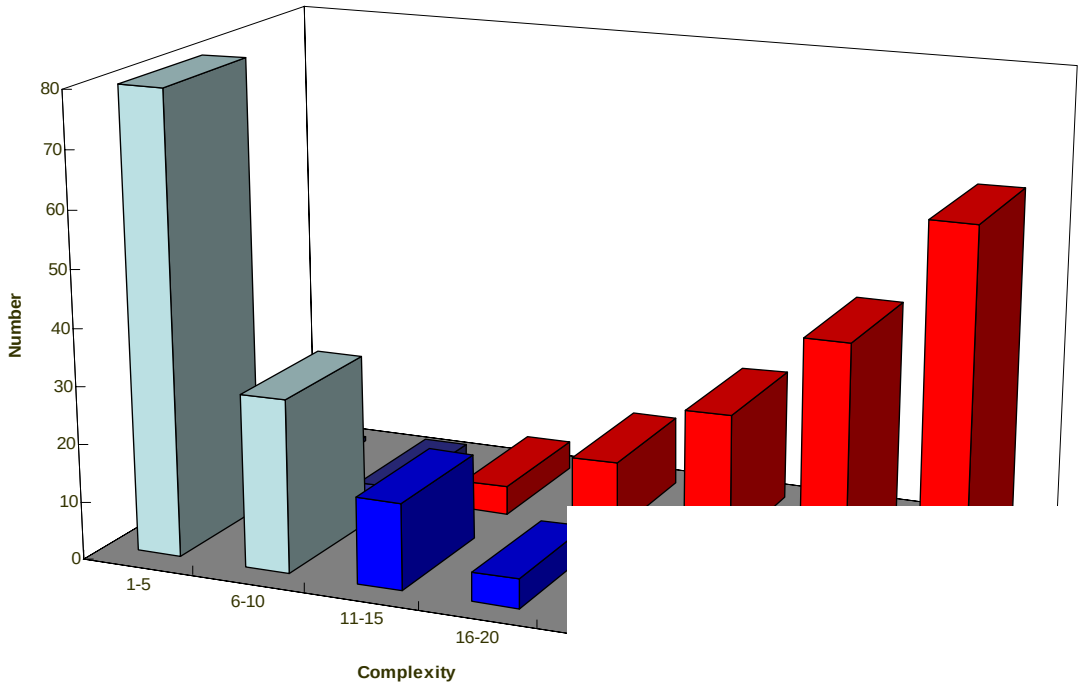
McCabe复杂度与软件质量的关系

- 超级复杂度模块
  - 圈复杂度 = 192
  - 基本复杂度 = 151
  - 模块设计复杂度 = 36
  - 问题：这样的代码没有希望
    - 太容易出错，无法使用
    - 太过复杂，难以整治
    - 规模太大，不宜再开发
  - 解决办法：在开发和维护过程中控制复杂度
    - 设定复杂度限定值，并贯彻执行



McCabe复杂度与软件质量的关系

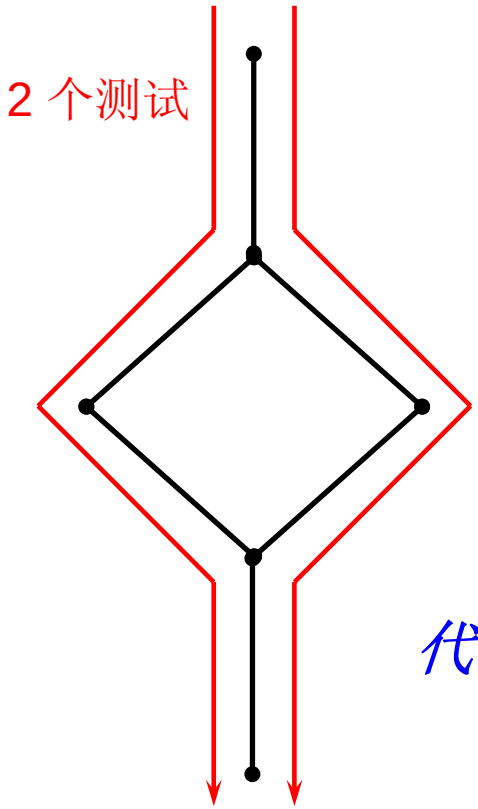
- McCabe推荐使用标准
  - 圈复杂度 v(G) ≤10
  - 基本复杂度 ev(G) ≤ 4



结构化测试

测试技术比较…

- 代码覆盖率

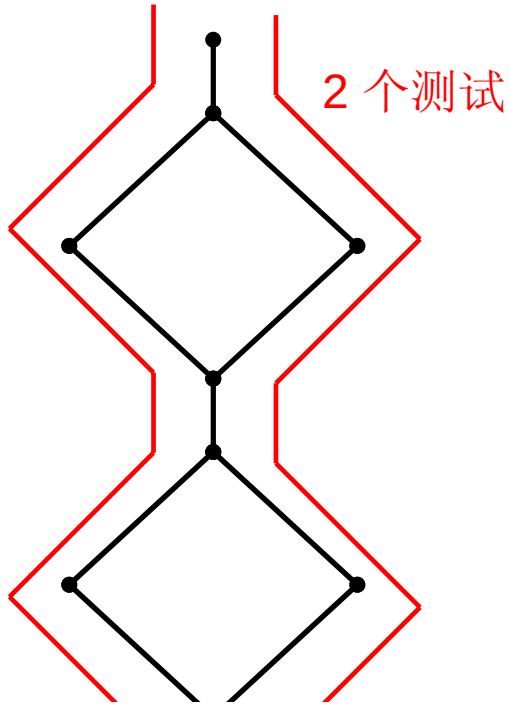


流图 ‘A’

哪个函数更复杂？

要求覆盖率达到**100%**，
各需多少个测试？

代码覆盖与复杂度不成比例

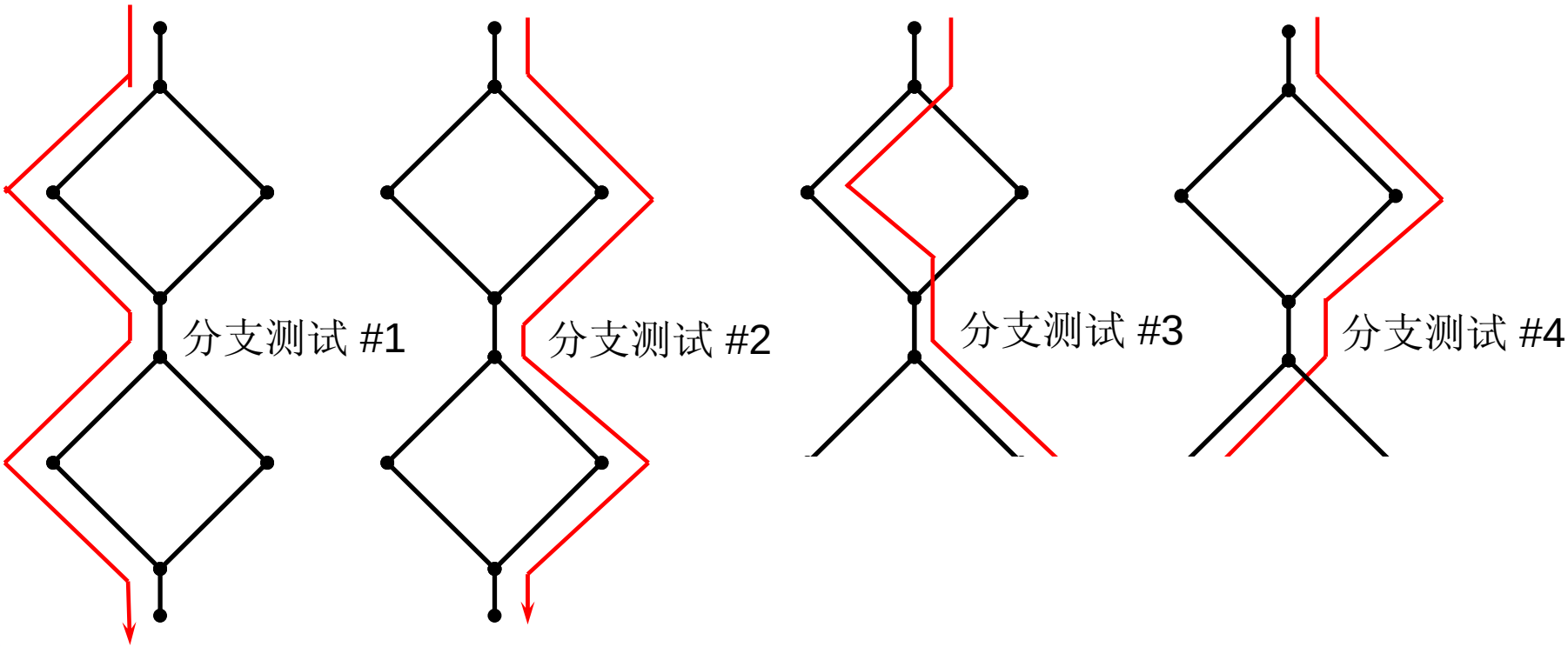


结构化测试

测试技术比较…

- 分支覆盖

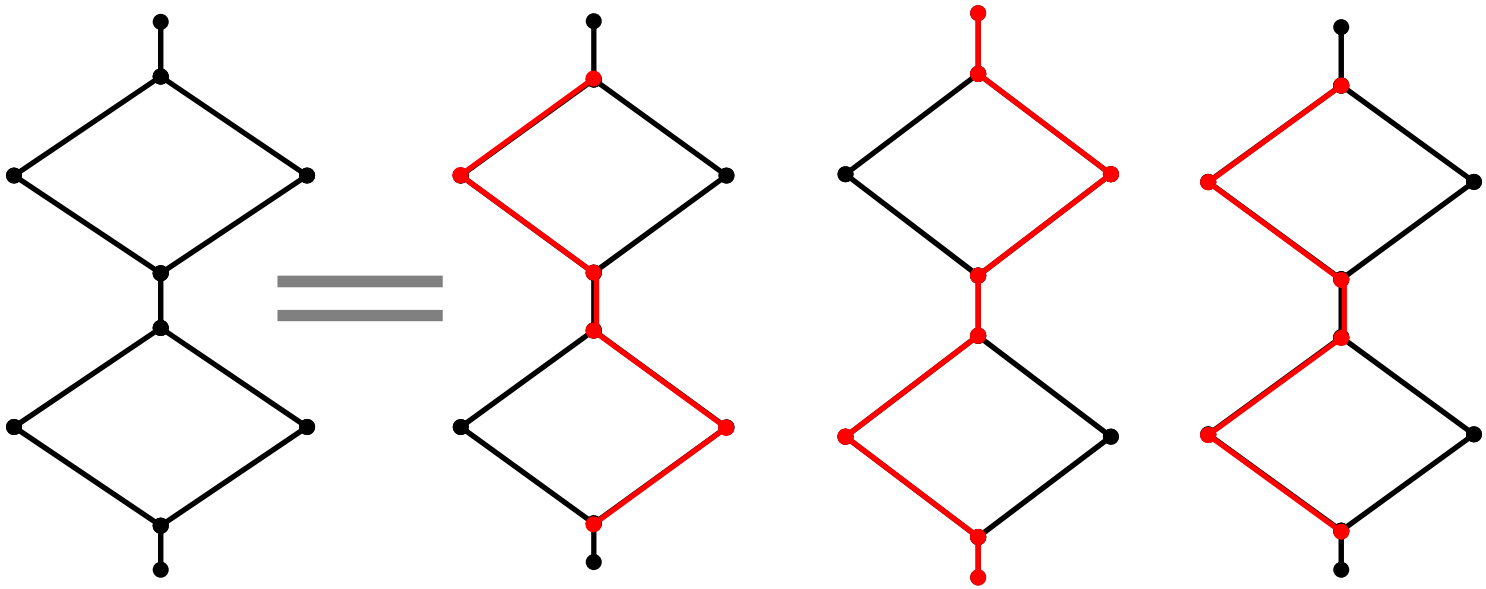
需要多少个测试可以做到100%覆盖?



结构化测试

测试技术比较…

- 基本路径覆盖



(TRUE & FALSE) + (FALSE & TRUE) - (TRUE & TRUE) = (FALSE & FALSE)

任何其他路径 (如 **FALSE and FAI**

基本路径覆盖可以达到分支
基本路径数 = 模块的圈复杂

结构化测试

➢基本路径覆盖

简单地说，*结构化测试* 要求贯穿模块的一组基本路径必须测试，以保证测试的充分性

➢结构化测试的原理

- ➢贯穿模块流图的基本路径是线性独立的
- ➢因此，任何其他路径都可以由基本路径组合而来
- ➢基本路径数等于McCabe圈复杂度

本篇小结

- 常见的软件度量方法
  - 行数度量
  - Halstead度量
  - 功能点
- McCabe度量
  - 圈复杂度
  - 基本复杂度
  - 模块设计复杂度
  - ……
- 用McCabe复杂度分析软件
- 基本路经测试的原理

谢谢大家