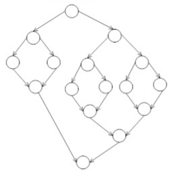
Software College Northeastern University

Software Quality Assurance and Testing

Chapter 5 White-Box Testing

wuchenni@qq.com

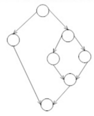
1. （1）控制流图



（2）测试用例

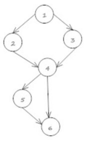
编号	输入	预期结果
1	居民用电，200 度以下（199 度）	A 类收费
2	居民用电，200 度以上（202 度）	B 类收费
3	动力电，非高峰 9000 度	B 类收费
4	动力电，非高峰 10100 度	C 类收费
5	动力电，高峰 9000 度	C 类收费
6	动力电，高峰 10100 度	D 类收费

2. （1）控制流图



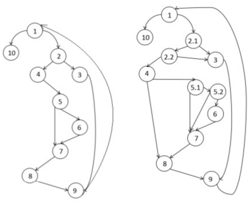
（2）判定覆盖、路径覆盖测试用例，满足条件 X>=80 and Y>=80 为真；X>=80 and Y>=80 为假，（X+Y）>=140 and（X>=90 or Y>=90）为真；X>=80 and Y>=80 为假，（X+Y）>=140 and （X>=90 or Y>=90）为假的测试用例即可，共需要 3 条测试用例。
条件覆盖测试用例，测试用例能够覆盖以下所有条件即可：X>=80,Y>=80, X<80, Y<80, X+Y>=140, X+Y<140, X>=90, X<90, Y>=90, Y<90。
条件组合覆盖测试用例，测试用例满足以下条件的组合即可：X>=80, Y>=80, X+Y>=140, X>=90, Y>=90; X>=80,Y<80, X+Y>=140, X>=90,Y<90; X>=80,Y<80, X+Y<140, X>=90,Y<90; X<80,Y>=80, X+Y>=140, X<90,Y>=90; X<80,Y>=80, X+Y<140, X<90,Y>=90; X<80,Y<80, X+Y>=140, X<90,Y>=90; X<80,Y<80, X+Y<140, X<90,Y<90。

3.（1）控制流图



（2）语句覆盖、判定覆盖的测试用例只要保证左侧分支和右侧分支都覆盖即可。
条件覆盖只要保证 x>0, y>0, magic<0 三个条件的真值和假值都取过值即可。
条件组合覆盖只要保证 x>0, y>0 这两个条件的真值假值的四种组合与 magic<0 条件的真值和假值组合过即可。
路径覆盖共有 4 条路径，即 1-2-4-5-6（不可能）；1-2-4-6; 1-3-4-5-6; 1-3-4-6。为可能的三条路径选择合适的 x, y, magic 值即可。

4.（1）

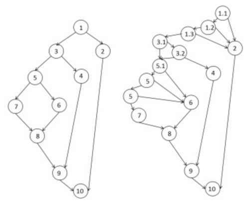


（2）4,6

（3）基本控制流图对应的独立路径：1-10；1-2-3-9-1-10；1-2-4-5-7-8-9-1-10；1-2-4-5-6-7-8-9-1-10；详细控制流图对应的独立路径：1-10；1-2.1-3-9-1-10；1-2.1-2.2-3-9-1-10；1-2.1-2.2-4-5.1-7-8-9-1-10；1-2.1-2.2-4-5.1-5.2-7-8-9-1-10；1-2.1-2.2-4-5.1-5.2-6-7-8-9-1-10。

根据每条路径的情况选择具体的变量值，并给出预期结果即可。

5.（1）



（2）4,9

（3）基本控制流图对应的独立路径：1-2-10；1-3-4-9-10；1-3-5-6-8-9-10；1-3-5-7-8-9-10；详细控制流图对应的独立路径：1.1-2-10；1.1-1.2-2-10；1.1-1.2-1.3-2-10；1.1-1.2-1.3-3.1-3.2-4-9-10；1.1-1.2-1.3-3.1-3.2-5.1-6-8-9-10；1.1-1.2-1.3-3.1-5.1-5.2-6-8-9-10；1.1-1.2-1.3-3.1-5.1-5.2-5.3-6-8-9-10；1.1-1.2-1.3-3.1-5.1-5.2-5.3-7-8-9-10；1.1-1.2-1.3-3.1-3.2-5.1-5.2-5.3-7-8-9-10。

根据每条路径的情况选择具体的变量值，并给出预期结果即可，如果路径的条件不可能达到，直接省略即可。

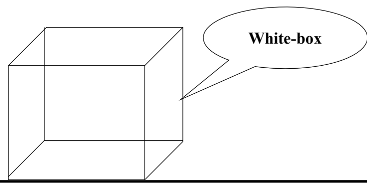
Contents

Chapter 5

White-Box Testing

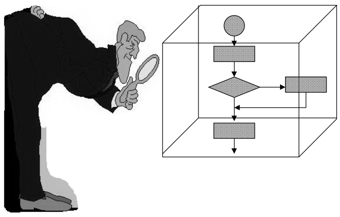
5.1 Basic Concepts

- White-box testing
  - It is a verification technique, software engineers can use it to examine if their codes work as expected.
  - It takes into account the internal mechanism of a system or component.
  - It is also known as structural testing, clear box testing, and glass box testing.



5.1 Basic Concepts

- White-box testing
  - It indicates that you have full visibility of the internal workings of the software product, specifically, the logic and the structure of the code.



5.1 Basic Concepts

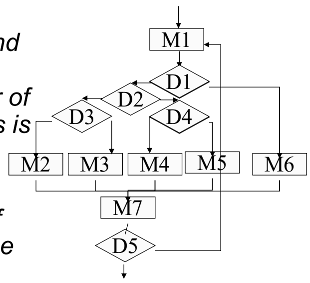
- White-box testing: static testing and dynamic test.
- Static white-box testing methods: Code inspection, Static structure analysis, Static quality metric method , etc.
- Dynamic white-box testing is based on coverage, as far as possible coverage of the test program structure characteristic and logical path.
- Dynamic white-box testing methods: logic coverage, loop coverage, basis path coverage, etc.
- Mainly used for unit test.

5.1 Basic Concepts

- White-box testing must follow several principles:
  - All independent path in a module must be implemented at least once.
  - All logic values require test two cases: true and false.
  - Inspection procedures of internal data structure, and ensuring the effectiveness of its structure.
  - Run all cycles within operational range.

5.1 Basic Concepts

- White-box Testing Difficulties
  - *For multiple choices and nesting cycles of the procedure, the number of different possible paths is astronomical.*
  - *cycle≤20times*
  - *Different paths is 5^{20},if the implementation time of each path is 1 ms , 3170years are needed.*



5.1 Basic Concepts

- Why we can’t use exhaustive testing?
  - Path exhaustive testing methods could not detect whether the procedure itself violated design specifications, whether it is a wrong procedure.
  - Path exhaustive testing procedure can’t detect the wrong because of path omission.
  - Path exhaustive testing can not discovery some errors associated with the data.

5.1 Basic Concepts

- Development of White-box Testing divided into four generations.
  - The first generation of white box testing :
    - Test development initial period.
    - Debugging,assert and print statements.
  - The second generation of white box testing :
    - Operation by formal language. (Test scripts)
    - Test scripts are combined into test cases, test cases are combined into test sets, using test engineering to manage test sets.
    - Use code coverage evaluation test results.
    - RTRT, Code Test, Visual Tester, C++ Tester , etc.

5.1 Basic Concepts

- The third generation of white box testing :
  - Solved the problem of repetition tests， the test mode changes from one-off transition to continue to test mode.
  - Xunit
- The fourth generation of white box testing :

第一关键域	第二关键域	第三关键域
在线测试	灰盒调测	持续测试
在线测试驱动	基于调用接口	测试设计先行
在线脚本桩	调试即测试	持续保障
在线测试用例设计、运行、以及评估改进	集编码、调试、测试于一体	重构测试用例

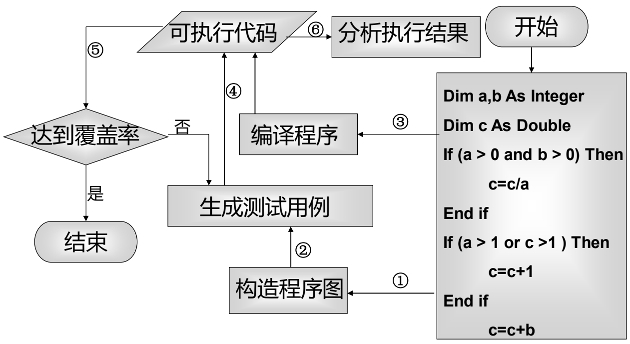
5.1 Basic Concepts

- A Comparison on four generations of White-box Testing

	Evaluation test results	Test automation	Continue to test	Debugging and test together
The first	No	No	No	No
The second	Yes	Yes	No	No
The third	Yes	Yes	Yes	No
The fourth	Yes	Yes	Yes	Yes

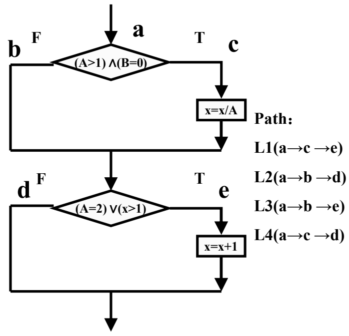
5.1 Basic Concepts

- 白盒测试方法测试过程



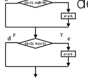
5.2 Logic Coverage

- Logic coverage
  - Statement coverage
  - Decision coverage
  - Condition coverage
  - Condition/decision coverage
  - Condition combination coverage
  - Path coverage



5.2 Logic Coverage—Statement Coverage

- Statement coverage（语句覆盖）
  - Statement coverage is to design a number of test cases, running the tested procedures, making each executable statement implement at least once.
  - In diagram, all the executable statements are in the path L1, so choose path design test cases, all the executable statements can be covered .

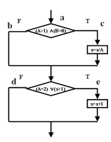


L1:(A=2) and (B=0) or (A>1) and (B=0) and (x/A>1)

【（2, 0, 3） 】covers ace【L1】

5.2 Logic Coverage—Statement Coverage

- In this example if swap ∧ and ∨， this test case can also cover all these four executable statements; if x>1 is written wrong as x>0, test case can't find this problem.



Logical operation error in judgment may not be found

5.2 Logic Coverage—Decision coverage

- Decision Coverage（判定覆盖） is to design a number of test cases, running the tested procedures, make the true and false branches of each judgment may go through at least once.

【2, 0, 3】covers ace【L1】 【1, 1, 1】covers abd【L2】
or
【3, 0, 3】covers acd【L4】 【2, 1, 1】covers abe【L3】

If x>1 is written wrong as x>0，test case can't find this problem. If x>1 is written wrong as x<1,use the above test case abe may get the same result.
Decision Coverage is not guaranteed they can detect the wrong conditions in judgment

5.2 Logic Coverage—Condition coverage

- Conditions coverage（条件覆盖） is to design a number of test cases, running the tested procedures, make possible values of each condition in the procedure may implement at least once.
  - For the first judgment
    - Condition A > 1 true value is T1, false value is !T1
    - Condition B = 0 true value is T2, false value is !T2
  - For the second judgment
    - Condition A = 2 true value is T3, false value is !T3
    - Condition X > 1 true value is T4, false value is !T4

5.2 Logic Coverage—Condition Coverage

Test case	Path	Condition value	Coverage branch
(2, 0, 3)	ace(L1)	T1 T2 T3 T4	c，e
(1, 1, 1)	abd(L2)	!T1 !T2 !T3 !T4	b，d

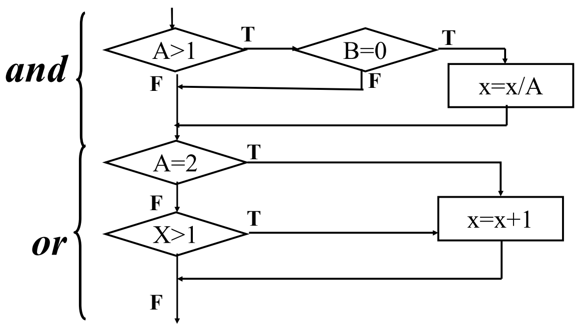
Test case	Path	Condition value	Coverage branch
【(1, 0, 3), (1, 0, 4)】	abe(L3)	!T1 T2 !T3 T4	b，e
【(2, 1, 1), (2, 1, 2)】	abe(L3)	T1 !T2 T3 !T4	b，e

5.2 Logic Coverage—Condition / Decision Coverage

- Condition/decision coverage（条件判断覆盖） is to design sufficient test cases, make all possible conditions of each judgment implement at least once, and make all possible results of each judgment implement at least once.
- For the first judgment to meet this requirement.

Test case	path	Condition value	Coverage branch
(2,0,3)	ace(L1)	T1 T2 T3 T4	c，e
(1,1,1)	abd(L2)	!T1 !T2 !T3 !T4	b，d

5.2 Logic Coverage—Decomposition



5.2 Logic Coverage—Condition Combination Coverage

- Condition combination coverage（条件组合覆盖） is to design sufficient test cases, running tested procedures, make all possible condition combinations of each judgment implement as least once.
- If test cases meet the condition combination coverage, then they certainly meet the decision coverage, condition coverage and condition/decision coverage.

5.2 Logic Coverage—Condition Combination Coverage

① A > 1, B = 0 as T1T2
② A > 1, B≠0 as T1!T2
③ A≯1, B = 0 as !T1T2
④ A≯1, B≠0 as !T1!T2
⑤ A = 2, X > 1 as T3T4
~~⑥ A = 2, X≯1 as T3!T4~~
⑦ A≠2, X > 1 as !T3T4
⑧ A≠2, X≯1 as !T3!T4

5.2 Logic Coverage—Condition Combination Coverage

Test case	Path	Coverage condition	Coverage branch	Combination coverage No.
(2, 0, 3)	ace(L1)	T1 T2 T3 T4	c，e	① ⑤
(2, 1, 1)	abe(L3)	T1 !T2 T3 !T4	b，e	② ⑥
(1, 0, 3)	abe(L3)	!T1 T2 !T3 T4	b，e	③ ⑦
(1, 1, 1)	abd(L2)	!T1 !T2 !T3 !T4	b，d	④ ⑧

There are four paths altogether in the procedure. Although the four test cases above cover all condition combinations and 4 branches, only 3 paths are covered and the path “acd” is missed.

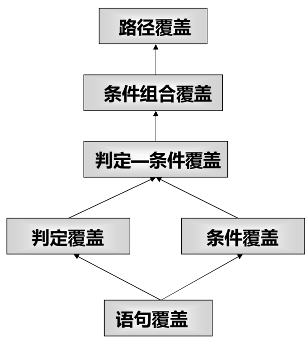
5.2 Logic Coverage—Path Coverage

- Path coverage（*路径覆盖*） is to design enough test cases to cover all possible paths in the procedure.

No.	Test case	path	Coverage condition
1	(2, 0, 3)	ace(L1)	T1T2T3T4
2	(1, 0, 1)	abd(L2)	!T1!T2!T3!T4
3	(2, 1, 1)	abe(L3)	!T1!T2!T3T4
4	(3, 0, 3)	acd(L4)	T1T2!T3!T4

5.2 Logic Coverage — Summary

- 逻辑覆盖中的递进关系



5.2 Logic Coverage

- Complete coverage

No.	Test case	path	Coverage condition
1	(2, 0, 3)	ace(L1)	① ⑤
2	(1, 0, 1)	abd(L2)	④ ⑧
3	(2, 1, 1)	abe(L3)	② ⑥
4	(3, 0, 3)	acd(L4)	① ⑧
5	(1, 0, 3)	abe(L3)	③ ⑦

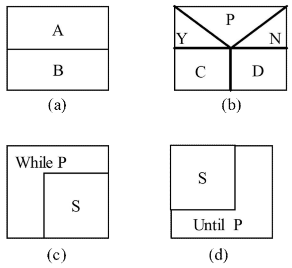
- Good — enough

5.2 Logic Coverage

- Calculate the minimal test cases number
  - Using N-S graph
- 结构化程序由3种控制结构组成
  - 顺序型 —— 构成串行操作
  - 选择型 —— 构成分支操作
  - 重复型 —— 构成循环操作

5.2 Logic Coverage

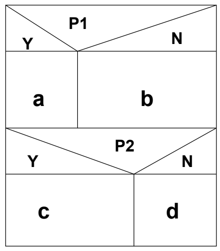
- N-S graph



- A,B,C,D,S
- P （Y or N ）

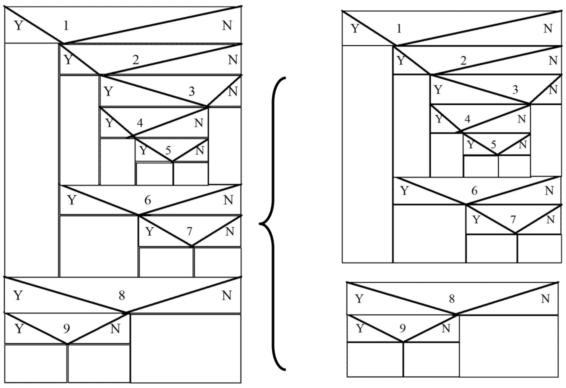
5.2 Logic Coverage

- Example：

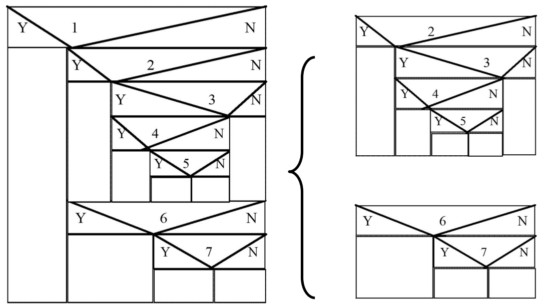


- 谓词 P1、P2 取不同值的时候将分别执行 a 或 b ， c 或 d 操作。
- 测试这个程序至少需要4个测试用例。
- 2 × 2 = 4
  - *2个分支谓词组合得到的*
  - *1 + 1 = 2 两个并列的操作*

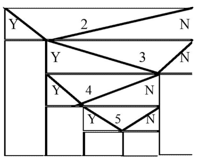
5.2 Logic Coverage



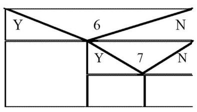
5.2 Logic Coverage



5.2 Logic Coverage



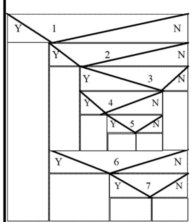
1+1+1+1+1=5



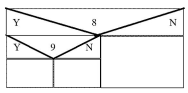
1+1+1=3

×= 15

5.2 Logic Coverage



1+15=16



1+1+1=3

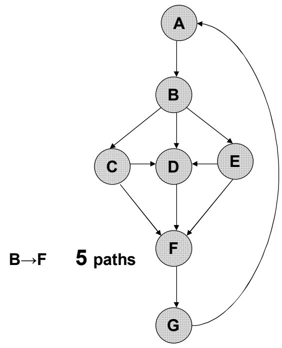
×= 48

5.2 Logic Coverage

- 请设计以下程序的逻辑覆盖测试用例。
  - Void DoWork (int x, int y, int z)
  - {
  - int k=0,j=0;
  - if ((x>3)&&(z<10))
  - { k=x*y-1;
  - j=sqrt(k);
  - }
  - if ((x==4)|||| (y>5))
  - { j=x*y+10; }
  - j=j%3;
  - }

5.3 Review

- Logic Coverage
  - Statement coverage
  - Decision coverage
  - Condition coverage
  - Condition/decision coverage
  - Condition combination coverage
  - **Path coverage**
- Basis Path Testing
- Control Flow Graph

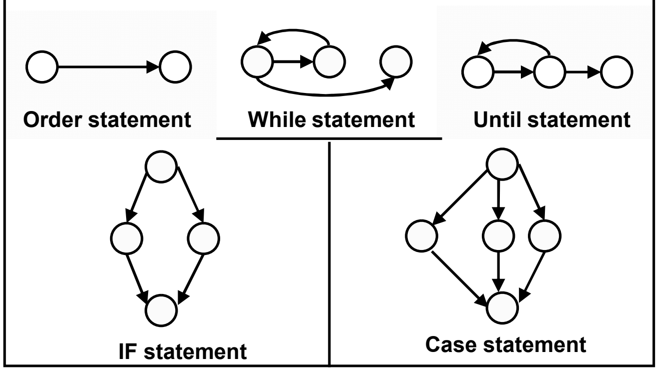


5.3 Control Flow Graph

- Concept
  - During procedure design , in order to more prominent the control flow structure, the procedure can be simplified, the simplified graph is called control flow graph.
  - On a flow graph:
    - Arrows called ***edges*** represent flow of control.
    - Circles called ***nodes*** represent one or more actions.
    - Areas bounded by edges and nodes called ***regions***.
    - A ***predicate node*** is a node containing a condition.

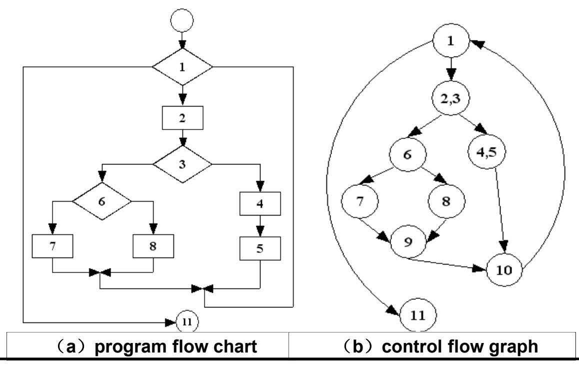
5.3 Control Flow Graph

- Common control flow graph



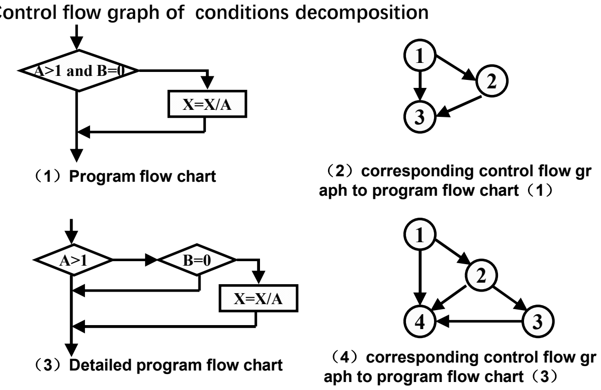
5.3 Control Flow Graph

Change a program flow chart into a control flow graph



5.3 Control Flow Graph

- **Control flow graph of conditions decomposition**



Exercise

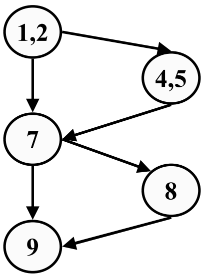
- Use logical coverage methods to test the program fragment below
  - void DoWork(int x, int y, int z)
  - {
  - 1 int k=0,j=0;
  - 2 if ((x>3)&&(z<10))
  - 3 {
  - 4 k=k*y-1;
  - 5 j=sqrt(k);
  - 6 }
  - 7 if ((x==4)||(y>5))
  - 8 j=x*y+10;
  - 9 j=j%3;
  - 10 }

Exercise

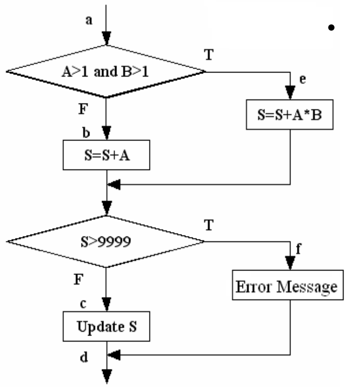
- Requirement
  - Draw the control flow graph of the program fragment.
  - Use statement coverage, decision coverage, condition coverage, condition/decision coverage, condition combination coverage, and path coverage to design test cases.

Exercise

- Answer
  - Control flow graph

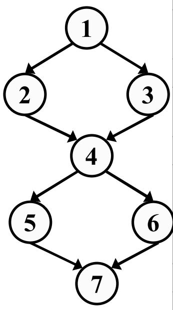


Exercise



- Requirement
  - Draw control flow graph
  - Use statement coverage, decision coverage, condition coverage, condition/decision coverage, condition combination coverage and path coverage to design test case.

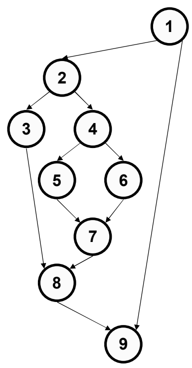
Exercise



No.	Coverage type	Test case	Expected output
1	Statement coverage	A=2，B=2，S=9999	Error Message
2		A=1，B=1，S=9997	Update S
3	Decision coverage	A=2，B=2，S=9999	Error Message
4		A=1，B=1，S=9997	Update S
5	Condition combination coverage	A=2，B=2，S=9997	Error Message
6		A=1，B=2，S=9999	Error Message
7		A=1，B=1，S=9997	Update S
8		A=2，B=1，S=9997	Update S
…	…	…	…

Exercise

```
main()
{ float a, b, c, x1, x2, mid;
  Scanf("%f, %f, %f", &a, &b, &c);
  if(a!=0)
  { mid=b*b-4*a*c;
    if(mid>0)
    { x1= (-b+sqrt(mid))/(2*a);
      x2=(-b-sqrt(mid))/(2*a);
      printf("two real roots\n");
    }
    else
    { if(mid==0)
        { x1=-b/2*a;
          printf("one real root\n");
        }
      else
      { x1= -b/(2*a);
        x2=sqrt(-mid)/(2*a);
        printf("two complex roots\n");
      }
    }
    printf("x1=%f, x2=%f\n",x1,x2);
  }
}
```



5.4 Basis Path Testing

- A testing mechanism proposed by McCabe.
- Aim is to derive a logical complexity measure of a procedural design and use this as a guide for defining a basic set of execution paths.
- Test cases which exercise basic set will execute every statement at least once.

5.4 Basis Path Testing

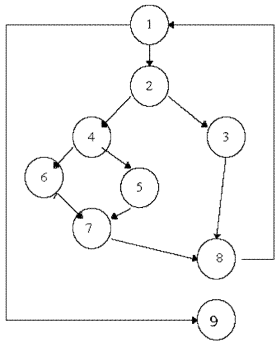
- Cyclomatic Complexity （*基本复杂度/圈复杂度*）
  - It gives a quantitative measure of the logical complexity.
  - This value gives the number of independent paths in the basis set, and an upper bound for the number of tests to ensure that each statement and both sides of every condition is executed at least once.
  - An independent path is any path through a program that introduces at least one new set of processing statements (i.e., a new **node**) or a new condition (i.e., a new **edge**)

5.4 Basis Path Testing

- Cyclomatic Complexity =
  - #Edges - #Nodes + #terminal vertices (usually 2)
  - #Predicate Nodes + 1
  - Number of regions of flow graph.

5.4 Basis Path Testing

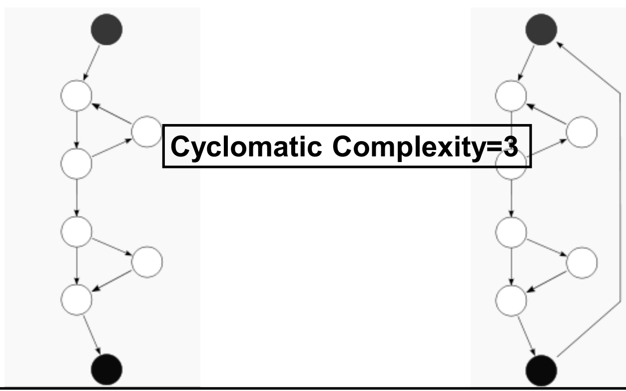
- Independent Paths:
  - 1, 9
  - 1, 2, 3, 8, 1, 9
  - 1, 2, 4, 5, 7, 8, 1, 9
  - 1, 2, 4, 6, 7, 8, 1, 9



5.4 Basis Path Testing

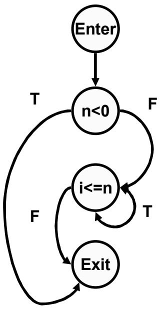
- Deriving Test Cases
  - Using the design or code, draw the corresponding flow graph.
  - Determine the cyclomatic complexity of the flow graph.
  - Determine a basis set of independent paths.
  - Prepare test cases that will force execution of each path in the basis set.

5.4 Basis Path Testing



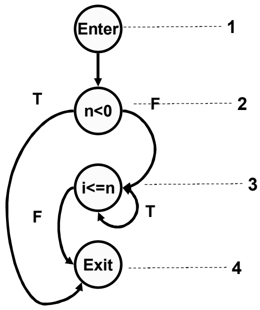
5.4 Basis Path Testing

```
main()
{
    int i, n, f;
    printf("n = ");
    Scanf("%d", &n);
    If (n<0) {
        pirntf("Invalid: %d\n" , n);
      n = -1;
    }  else {
        f = 1;
        for (i = 1; i <=n; i++) {
            f*=i;
    }
    Printf("%d! = %d.\n", n, f);
    }
    Return n;
}
```



5.4 Basis Path Testing

- Cyclomatic complexity
  - 2+1=3
  - 1 -> 2 -> 4
  - 1 -> 2 -> 3 -> 4
  - 1 -> 2 -> 3 -> 3 -> 4

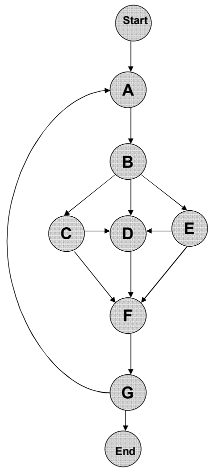


5.4 Basis Path Testing

Path	Input	Expected result
1 -> 2 -> 4	-1	Invalid :-1
1 -> 2 -> 3 -> 4	0	0!=1
1 -> 2 -> 3 -> 3 -> 4	1	1!=1

5.4 Basis Path Testing

- 练习1：
  - 计算右图的圈复杂度
  - 标识它的独立路径

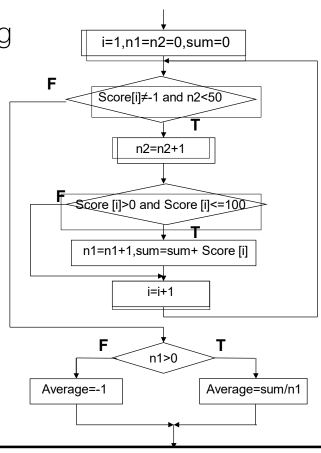


5.4 Basis Path Testing

- 答案：
  - 独立路径数=圈数= 5
  - 独立路径数= e – n + 1 = 11 – 7 + 1 = 5
- 独立路径：
  - ABCFG
  - ABDFG
  - ABEFG
  - ABCDFG
  - ABEDFGABDFG

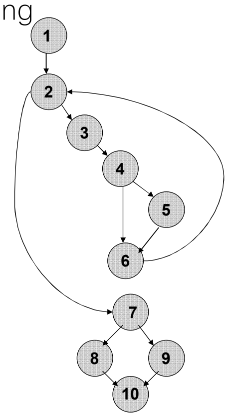
5.4 Basis Path Testing

- 练习2：
  - 从键盘上得到最多50个值（以-1作为输入结束标志），求出学生分数的个数、总分数和平均分。程序流程图如右所示。



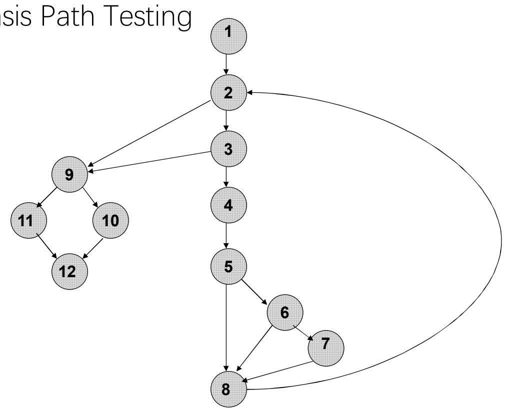
5.4 Basis Path Testing

- 答案1:



5.4 Basis Path Testing

- 答案2:



5.4 Basis Path Testing

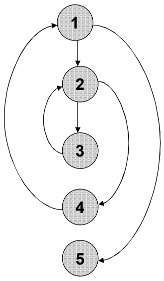
- 答案2:
  - 路径1: 1-2-9-10-12
  - 路径2: 1-2-9-11-12
  - 路径3: 1-2-3-9-10-12
  - 路径4: 1-2-3-4-5-8-2…
  - 路径5: 1-2-3-4-5-6-8-2…
  - 路径6: 1-2-3-4-5-6-7-8-2 …

Summary

- 1、给定程序流程图转换控制流图注意两个要点。
- 2、给定程序片段绘制控制流图要注意 if-then-else 语句和 if-then 语句的区别。
- 3、用基路径法测试注意控制流图是否是强连接。
- 4、通过将复合判定条件分解的方式可以提高测试强度，但生成的测试用例会增加。
- 5、基路径测试是路径测试的一种，尽管也考虑了循环问题，但是要着重测试循环体要用循环测试策略。

5.5 Loop Testing

- 引例:
- 代码（c语言）该循环的测试用例设计思路是怎样的?
  - For (i=0; i<num; i++)
  - {
  -   while (j>0)
  -   {
  -     j--;
  -   }
  - }

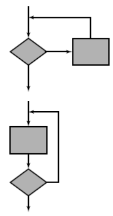


5.5 Loop Testing

- Focuses exclusively on the validity of loop constructs.
- Types of Loop
  - Simple Loops
  - Nested Loops
  - Concatenated Loops
  - Unstructured Loops

5.5 Loop Testing- Simple Loops

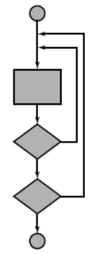
- The following set of tests can be applied to simple loops, where **n** is the maximum number of allowable passes through the loop
  - Skip the loop entirely.
  - Only one pass through the loop.
  - Two passes through the loop.
  - m passes through the loop where m < n.
  - n-1, n, n+1 passes through the loop



Simple loops

5.5 Loop Testing- Nested Loops

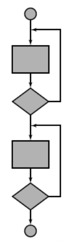
- Start at the innermost loop. Set all other loops to minimum values.
- Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.
- Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
- Continue until all loops have been tested.



Nested loops

5.5 Loop Testing- Concatenated Loops

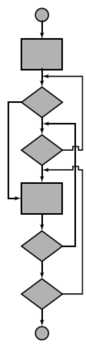
- If each of the loops is independent of the other:
  - Concatenated loops can be tested using the approach defined for simple loops,
- Else
  - the approach applied to nested loops is recommended.



Concatenated loops

5.5 Loop Testing- Unstructured Loops

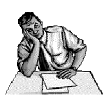
- Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs



Unstructured loops

5.5 Loop Testing

- Problem:
  - 1. To simple loops, where n is the maximum number, how to test n + 1 passes through the loop?
  - 2. For nested Loops test conduct simple loop tests for the innermost loop while holding the outer loops at their minimum values. The minimum value is one or zero?



5.5 Loop Testing

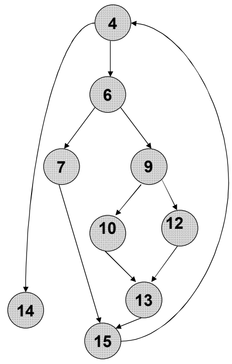
- The easy way to test loops is using the method Z path coverage.
- Z path coverage method is a program in loop structure will be simplified into If structure test method.
- The purpose is to limit cycle simplified cyclic number, regardless of the form and loop body circulating the number of actual implementation, simplified cycle test only once, or zero times. In this case, the effect of loop structure and If branches are same, namely the loop body or execution, or skip.

5.5 Loop Testing

- Example:
  - Void Sort(int iRecordNum, int iType)
  - 1 {
  - 2 int x=0;
  - 3 int y=0;
  - 4 while (iRecordNum-- >0)
  - 5 {
  - 6 If (iType==0)
  - 7 x=y+2;
  - 8 else
  - 9 If (iType==1)
  - 10 x=y+10;
  - 11 else
  - 12 x=y+20;
  - 13 }
  - 14 }

5.5 Loop Testing

- Answer:
- Path 1: 4 → 14
- Path 2: 4 → 6 → 7 → 15
- →4→14
- Path 3: 4 → 6 → 9 → 10
- → 13 →15→ 4 → 14
- Path 4: 4 → 6 → 9 → 12
- → 13 →15→ 4 → 14

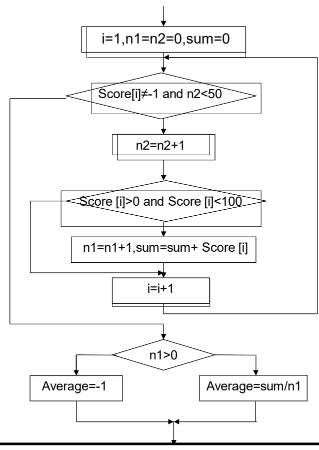


5.5 Loop Testing

	输入数据	预期输出
测试用例1	iRecordNum=0 iType=0	x=0 y=0
测试用例2	iRecordNum=1 iType=0	x=2 y=0
测试用例3	iRecordNum=1 iType=1	x=10 y=0
测试用例4	iRecordNum=1 iType=2	x=20 y=0

5.5 Loop Testing

- 练习：
  - *从键盘上得到最多50个值（以-1作为输入结束标志），求出学生分数的个数、总分数和平均分。程序流程图如右所示。*



5.5 Loop Testing

- 答案：
  - 采用较复杂的循环测试策略测试循环，可采用下面测试集：
    - 跳过整个循环；
    - 只循环一次；
    - 只循环两次；
    - 循环m次，m<n；
    - 分别循环 n-1、n次

5.6 Data Flow Testing

- The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program
- Data flow testing is a powerful tool to detect improper use of data values due to coding errors
  - Incorrect assignment or input statement
  - Definition is missing (use of null definition)
  - Predicate is faulty (incorrect path is taken which leads to incorrect definition)

5.6 Data Flow Testing

- Need to focus some testing effort on these as well (not a focus of coverage-based testing)
  - Explore the sequences of events related to the data state and the unreasonable things that can happen to data.
  - Explore the effect of using the value produced by each and every computation

5.6 Data Flow Testing

- Variables that contain data values have a defined life cycle: created, used, killed (destroyed).
- The "scope" of the variable

```
{            // begin outer block
  int x;    // x is defined as an integer within this outer block
  ...;       // x can be accessed here
    {        // begin inner block
    int y; // y is defined within this inner block
  ...;       // both x and y can be accessed here
    }        // y is automatically destroyed at the end of
             // this block
    ...;     // x can still be accessed, but y is gone
  }   // x is automatically destroyed
```

5.6 Data Flow Testing

- Three possibilities exist for the first occurrence of a variable through a program path:
  - ~d - the variable does not exist (indicated by the ~), then it is defined (d)
  - ~u - the variable does not exist, then it is used (u)
  - ~k - the variable does not exist, then it is killed or destroyed (k)

5.6 Data Flow Testing

- d – defined, created, initialized
  - Data declaration; on left hand side of computation
- k – killed, undefined, released
- u – used for something (=c and p)
- c – right hand side of computation, pointer (calculation)
- p – used in a predicate (or as control variable of loop)

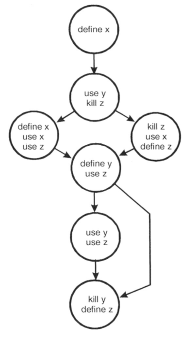
5.6 Data Flow Testing

- Time-sequenced pairs of defined (d), used (u), and killed (k):
  - dd - Defined and defined again—not invalid but suspicious. Probably a programming error.
  - du - Defined and used—perfectly correct. The normal case.
  - dk - Defined and then killed—not invalid but probably a programming error.
  - ud - Used and defined—acceptable.
  - uu - Used and used again—acceptable.
  - uk - Used and killed—acceptable.
  - kd - Killed and defined—acceptable. A variable is killed and then redefined.
  - ku - Killed and used—a serious defect. Using a variable that does not exist or is undefined is always an error.
  - kk - Killed and killed—probably a programming error

5.6 Data Flow Testing

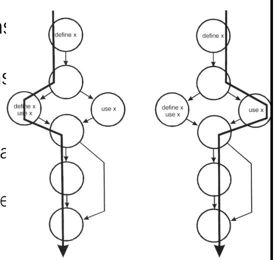
- A data flow graph is similar to a control flow graph in that it shows the processing flow through a module.

  In addition, it details the definition, use, and destruction of each of the module's variables.
- Technique
  - Construct diagrams
  - Perform a static test of the diagram
  - Perform dynamic tests on the module



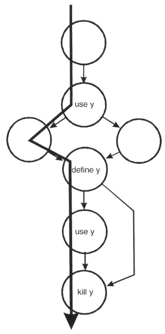
5.6 Data Flow Testing

- Perform a static test of the diagram
  - For each variable within the module we will examine define-use-kill patterns along the control flow paths
  - The define-use-kill patterns for x (taken in pairs as we follow the paths) are:
    - ~define - correct, the normal case
    - define-define - suspicious, perhaps a programming error
    - define-use - correct, the normal case



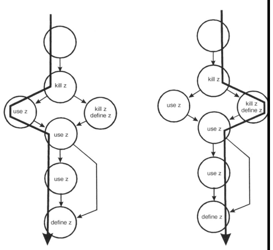
5.6 Data Flow Testing

- Perform a static test of the diagram
  - The define-use-kill patterns for y (taken in pairs as we follow the paths) are:
    - ~use - major blunder
    - use-define - acceptable
    - define-use - correct, the normal case
    - use-kill - acceptable
    - define-kill - probable programming error



5.6 Data Flow Testing

- Perform a static test of the diagram
  - The define-use-kill patterns for z (taken in pairs as we follow the paths) are:
    - ~kill - programming error
    - kill-use - major blunder
    - use-use - correct, the normal case
    - use-define - acceptable
    - kill-kill - probably a programming error
    - kill-define - acceptable
    - define-use - correct, the normal case



5.6 Data Flow Testing

- In performing a static analysis on this data flow model the following problems have been discovered:
  - x: define-define
  - y: ~use
  - y: define-kill
  - z: ~kill
  - z: kill-use
  - z: kill-kill

5.6 Data Flow Testing

- While static testing can detect many data flow errors, it cannot find them all => Dynamic Data Flow Testing
- Dynamic Data Flow Testing
  - Every "define" is traced to each of its "uses"
  - Every "use" is traced from its corresponding "define"
  - Steps
    - Enumerate the paths through the module
    - For every variable, create at least one test case to cover every define-use pair.

5.7 Mutation Testing

- Mutation Testing
  - is a fault-based testing technique.
  - provides a testing criterion called the "mutation adequacy score".
  - mutation adequacy score can be used to measure the effectiveness of a test set in terms of its ability to detect faults.

5.7 Mutation Testing

- We use mutation analysis for testing to :
  - Help testers design high quality tests
  - Evaluate the quality of existing tests

5.7 Mutation Testing

- Mutation Testing is capable of testing software at
  - unit level
    - Fortran, Ada, C, C#, Java, SQL, AspectJ
  - integration level
  - specification level
    - Finite State Machines, State charts, Petri Nets, Network protocols, Security Policies, Web Services

5.7 Mutation Testing

- Fundamental Premise（基本前提） of Mutation Testing
  - If the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects that fault

5.7 Mutation Testing

- Mutation and Mutants（变异体）
  - Mutation is the act of changing a program, albeit only slightly.
  - P: the original program under test.
  - M: a program obtained by slightly changing P.
  - M is known as a mutant of P,P the parent of M.
  - Mutate refers to the act of mutation. To mutate a program means to change it.

5.7 Mutatio

- Mutation operat
  - are designed tc insertion or del

Mutation Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	DO statement alterations
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

5.7 Mutation Testing

- Partial list of mutation tools

Language	Tool
Fortran	Mothra
C	Proteum
CSP	MsGAT
C#	Nester
Java	Jester
	μJava
	Lava
Python	Pester

5.7 Mutation Testing

- Example1:

```
begin
    int x , y;
    input (x , y);
    if (x < y)
        output (x + y);
    else
        output (x * y);
end
```

P

```
begin
    int x , y;
    input (x , y);
    if (x ≤ y)
        output (x + y);
    else
        output (x * y);
end
```

M1

5.7 Mutation Testing

- Example 1:

```
begin
    int x, y;
    input (x , y);
    if (x < y)
        output (x + y);
    else
        output (x * y);
end
```

P

```
begin
    int x, y;
    input (x , y);
    if (x < y)
        output (x + y);
    else
        output (x / y);
end
```

M2

5.7 Mutation Testing

- First-order mutants（一阶变异体）
  - Mutants generated by introducing only a single change to a program under test
  - Generally used in practice
  - Are preferred to higher-order mutants
- Higher-order mutants（高阶变异体）
  - Mutants other than first order

5.7 Mutation Testing

- Example 2

```
begin
    int x,y;
    input (x , y);
    if (x < y)
        output (x + y);
    else
        output (x * y);
end
```

P

```
begin
    int x,y;
    input (x , y);
    if (x < y + 1)
        output (x + y);
    else
        output (x / y);
end
```

M: Second-order mutant

5.7 Mutation Testing

- Syntax and semantics of mutants
  - Given a program P written in a well-defined programming language, a semantic change in P is made by making one or more syntactic changes.

$$f_p(x, y) = \begin{cases} x + y & \text{if } x<y \\ x * y & \text{otherwise} \end{cases}$$

$$f_p(x, y) = \begin{cases} x + y & \text{if } x\leq y \\ x * y & \text{otherwise} \end{cases}$$

$$f_p(x, y) = \begin{cases} x + y & \text{if } x<y \\ x / y & \text{otherwise} \end{cases}$$

5.7 Mutation Testing

- Strong mutation testing & Weak mutation testing
  - Strong mutation testing uses external observation.
  - A strong mutant and its parent are allowed to run to completion at which point their respective outputs are compared.
  - Weak mutation testing uses internal observation.
  - It is possible that a mutation behaves similar to its parent under weak mutation but not under strong mutation.

5.7 Mutation Testing-Example 3

```
enum dangerLevel (none, moderate, high, veryHigh);
procedure checkTemp (currentTemp, maxTemp) {
    float currentTemp[3], maxTemp; int highCount=0;
    enum dangerLevel danger;
    danger=none;
    if (currentTemp[0]>maxTemp)
        highCount=1;
    if (currentTemp[1]>maxTemp)
        highCount=highCount+1;
    if (currentTemp[2]>maxTemp)
        highCount=highCount+1;
if (highCount==1)    danger=moderate;
if (highCount==2)    danger= high;
if (highCount==3)    danger= veryHigh;
return(danger);
}
```

P

```
enum dangerLevel (none, moderate, high, veryHigh);
procedure checkTemp (currentTemp, maxTemp) {
    float currentTemp[3], maxTemp; int highCount=0;
    enum dangerLevel danger;
    danger=none;
    if (currentTemp[0]>maxTemp)
        highCount=1;
    if (currentTemp[1]>maxTemp)
        highCount=highCount+1;
    if (currentTemp[2]>maxTemp)
        highCount=highCount+1;
if (highCount≥1)    danger=moderate;
if (highCount==2)    danger= high;
if (highCount==3)    danger= veryHigh;
return(danger);
}
```

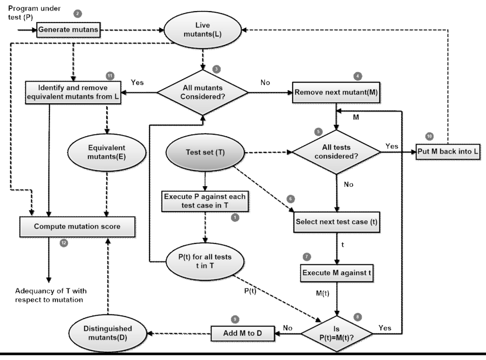
M

5.7 Mutation Testing

- The problem of test assessment using mutation can be stated as follow:
  - Let P be a program under test. T a test set for P, and R the set of requirements that P must meet.
  - Suppose that P has been tested against all tests in T and found to be correct with respect to R on each test case.
  - We want to know How good is T ?"

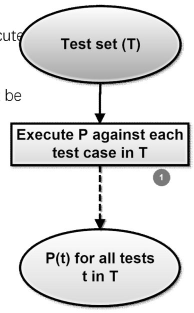
5.7 Mutation Testing

- Procedure for test-adequacy assessment



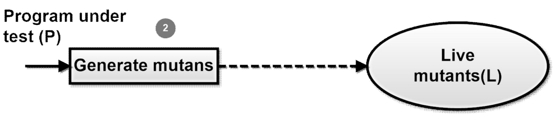
5.7 Mutation Testing

- Step1: Program execution
  - P(t): the observed behavior of P when executed against t.
  - P(t) is correct with respect to R for all t∈T.
  - If P(t) is found to be incorrect, then P must be corrected and Step 1 executed again.



5.7 Mutation Testing

- Step 2: Mutant generation

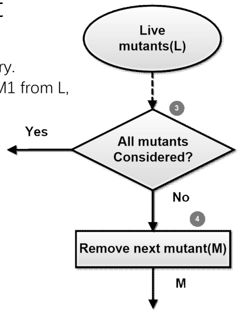


Line	Original	Mutant ID	Mutant(s)
1	begin		None
2	int x, y		None
3	input (x, y)		None
4	if (x < y)	M1	if(x+1<y)
5			
6			
		M4	output (x + y+1)
		M5	output (x - y)
7	else		None
8	output(x * y)	M6	output ((x + 1) * y)
		M7	output (x *(y+1))
		M8	output (x / y)
9	end		None

L={M1, M2,M3,M4,M5,M6,M7,M8}

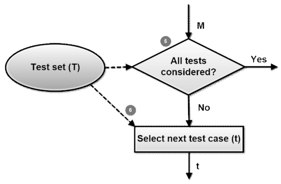
5.7 Mutation Testing

- Step 3 and 4: Select next mutant
  - The choice of mutant to select is arbitrary.
  - Let us select mutant M1. After moving M1 from L, we have L={M2,M3,M4,M5,M6,M7,M8}



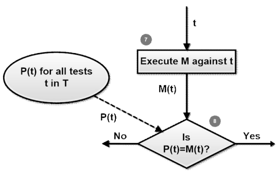
5.7 Mutation Testing

- Step 5 and 6: Select next test case
  - Find whether at least one of the tests in T can distinguish it from its parent P.
  - Terminate condition
    - All tests are exhausted
    - Or M is distinguished by some test



5.7 Mutation Testing

- Steps 7,8, and 9: Mutant execution and classification



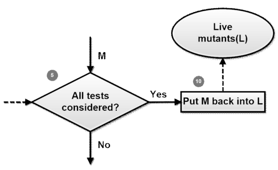
5.7 Mutation Testing-Example 5

Program	t1	t2	t3	t4	D
P(t)	0	1	0	2	{ }
Mutant					
M1(t)	0	0*	NE	NE	{M1}
M2(t)	0	1	0	2	{M1}
M3(t)	0	2*	NE	NE	{M1,M3}
M4(t)	0	2*	NE	NE	{M1,M3,M4}
M5(t)	0	-1*	NE	NE	{M1,M3,M4,M5}
M6(t)	0	1	0	0*	{M1,M3,M4,M5,M6}
M7(t)	0	1	1*	NE	{M1,M3,M4,M5,M6,M7}
M8(t)	U*	NE	NE	NE	{M1,M3,M4,M5,M6,M7,M8}

t1:<x=0,y=0>
t2:<x=0,y=1>
t3:<x=1,y=0>
t4:<x=-1,y=-2>

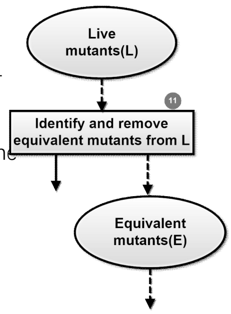
5.7 Mutation Testing

- Step 10: Live mutants
  - In our example, M2 could not be distinguished by Tp.
  - M2 is returned to the set of live mutants



5.7 Mutation Testing

- Step 11: Equivalent mutants
  - Check if L is nonempty.
  - Remaining live mutants are tested for equivalence to their parent program.
  - Equivalent: for each test input from the input domain of P, the observed behavior of M is identical to that of P.



5.7 Mutation Testing

- Example 6

$$f_p(x, y) = \begin{cases} x + y & \text{if } x<y \\ x * y & \text{otherwise} \end{cases}$$

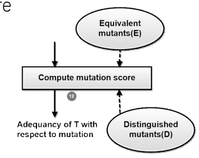
$$g_{M2}P(x, y) = \begin{cases} x + y & \text{if } x<y+1 \\ x * y & \text{otherwise} \end{cases}$$

- $f_p(x1,y1) \neq g_{M2}(x1,y1)$
  - Two condictions(C1 & C2) must hold for this.
  - C1: (x1<y1)≠(x1<y1+1)
  - C2:x1∗y1 ≠x1+y1
- t:<x=1
- P(t)=2

M2 is not equivalent to its parent P

5.7 Mutation Testing

- Step 12: Computation of mutation score
  - MS(T): mutation score
  - $MS(T)=\frac{|D|}{|L|+|D|}$
  - 0≤MS(T)≤1
  - $MS(T)=\frac{|D|}{|M|-|E|}$



|M|: total number of mutants generated in Step2

5.7 Mutation Testing

- Example 7
  - |D|=7
  - |L|=1
  - |E|=0
  - MS(T)=7/(7+1)=0.875

5.7 Mutation Testing

- Conditions for distinguishing a mutant
  - C1∧C2∧C3
    - C1: Reachability
    - C2: State infection
    - C3: State propagation
  - If there is no test case in the input domain of P that satisfies each of the three conditions above that the mutant is considered *equivalent* to the program under test

5.7 Mutation Testing

- Example 8
  - Reachability condition:
    - require that control arrive at line 14
  - State infection condition:
    - After execution of the statement at line 14, the state of the mutant must differ from that of its parent.
  - State propagation condition
    - No more changes can occur to the value of danger.

```
1   enum dangerLevel (none, moderate, high, veryHigh);
2   procedure checkTemp (currentTemp, maxTemp) {
3       float currentTemp[3], maxTemp; int highCount=0;
4       enum dangerLevel danger;
5       danger=none;
6       if (currentTemp[0]>maxTemp)
7           highCount=1;
8       if (currentTemp[1]>maxTemp)
9           highCount=highCount+1;
10      if (currentTemp[2]>maxTemp)
11          highCount=highCount+1;
12  if (highCount==1)    danger=moderate;
13  if (highCount==2)    danger= high;
14  if (highCount==3)    danger= none
15  return(danger);
16  }
```

5.7 Mutation Testing

- t:<currentTemp=[20,34,29], maxTemp=18>
- P(t)=veryHigh , M(t)=none

```
1   enum dangerLevel (none, moderate, high, veryHigh);
2   procedure checkTemp (currentTemp, maxTemp) {
3       float currentTemp[3], maxTemp; int highCount=0;
4       enum dangerLevel danger;
5       danger=none;
6       if (currentTemp[0]>maxTemp)
7           highCount=1;
8       if (currentTemp[1]>maxTemp)
9           highCount=highCount+1;
10      if (currentTemp[2]>maxTemp)
11          highCount=highCount+1;
12  if (highCount==1)    danger=moderate;
13  if (highCount==2)    danger= high;
14  if (highCount==3)    danger= veryHigh;
15  return(danger);
16  }
```

P

```
1   enum dangerLevel (none, moderate, high, veryHigh);
2   procedure checkTemp (currentTemp, maxTemp) {
3       float currentTemp[3], maxTemp; int highCount=0;
4       enum dangerLevel danger;
5       danger=none;
6       if (currentTemp[0]>maxTemp)
7           highCount=1;
8       if (currentTemp[1]>maxTemp)
9           highCount=highCount+1;
10      if (currentTemp[2]>maxTemp)
11          highCount=highCount+1;
12  if (highCount==1)    danger=moderate;
13  if (highCount==2)    danger= high;
14  if (highCount==3)    danger= none ;
15  return(danger);
16  }
```

M

5.7 Mutation Testing

- Example 9

```
String findElement (name, atWeight, int size, float w){
  String name[maxSize]; float atWeight[maxSize], w;
  int index=0;
  while ( index≤size ) {
    if (atWeight [index]>w)
      return(name[index]);
    index=index+1;
  }
  return ("None");
}
```

C1: any test case satisfies
C2:CP≠CM
C3: value returned by the mutant differs from the one returned by its parent

t1: <name=["Hydrogen", "Nitrogen", "Oxygen"] , atWeight=[1.0079,14.0067,15.9994], maxSize=3, size=2, w=15.0>

5.8 A Comparison of White-box Testing and Black-box Testing

	White-box Testing	Black-box Testing
Tester visibility	have visibility to the code and write test cases based upon the code	have no visibility to the code and write test cases based on possible inputs and outputs for functionality documented in specifications and /or requirements
A failed test case reveals controlled?	a problem (fault) Yes – the test case helps to identify the specific lines of code involved	a symptom of a problem (a failure) No – it can be hard to find the cause of the failure

3.8 A Comparison of White-box Testing and Black-box Testing

	White-box Testing	Black-box Testing
程序结构	已知程序结构	未知程序结构
规模	小规模测试	大规模测试
依据	详细设计说明	需求说明、概要设计说明
面向	程序结构	输入输出接口/功能要求
适用	单元测试	组装、系统测试
测试人员	开发人员	专门测试人员/外部人员
优点	能够对程序内部的特定部位进行覆盖	能站在用户的立场上进行测试
缺点	无法检验程序的外部特性 不能检测对要求的遗漏	不能测试程序内部特定部位 如果规格说明有误，则无法发现

Thank you !