

Program Testing

(Lecture 14)

Anil Kumar Dudyala
Dept. of CSE, NIT, Patna

Source
Rajib Mall

Organization of this Lecture

- Introduction to Testing.
- White-box testing:
 - statement coverage
 - path coverage
 - branch testing
 - condition coverage
 - Cyclomatic complexity
- Summary

Black-box Testing

- Test cases are designed using only functional specification of the software:
 - Without any knowledge of the internal structure of the software.
- For this reason, black-box testing is also known as functional testing.

White-Box Testing

- Designing white-box test cases:
 - Requires knowledge about the internal structure of software.
 - White-box testing is also called structural testing.

Black-Box Testing

- Two main approaches to design black box test cases:
 - Equivalence class partitioning
 - Boundary value analysis

White-Box Testing

- There exist several popular white-box testing methodologies:
 - Statement coverage
 - Branch coverage
 - Path coverage
 - Condition coverage
 - Mutation testing
 - Data flow-based testing

Coverage-Based Testing Versus Fault-Based Testing

- **Idea behind coverage-based testing:**
 - Design test cases so that certain program elements are executed (or covered).
 - Example: statement coverage, path coverage, etc.
- **Idea behind fault-based testing:**
 - Design test cases that focus on discovering certain types of faults.
 - Example: Mutation testing.

Statement Coverage

- Statement coverage methodology:
 - Design test cases so that every statement in the program is executed at least once.

Statement Coverage

- The principal idea:
 - Unless a statement is executed,
 - We have no way of knowing if an error exists in that statement.

Statement Coverage Criterion

- Observing that a statement behaves properly for one input value:
 - No guarantee that it will behave correctly for all input values.

Example

```
. int f1(int x, int y){  
. 1 while (x != y){  
. 2     if (x>y) then  
. 3         x=x-y;  
. 4     else y=y-x;  
. 5 }  
. 6 return x; }
```

Euclid's GCD Algorithm

Euclid's GCD Computation Algorithm

- By choosing the test set $\{(x=3, y=3), (x=4, y=3), (x=3, y=4)\}$
 - All statements are executed at least once.

Branch Coverage

- Test cases are designed such that:
 - Different branch conditions
 - Given true and false values in turn.

Branch Coverage

- Branch testing guarantees statement coverage:
 - A stronger testing compared to the statement coverage-based testing.

Stronger Testing

- Test cases are a **superset of a weaker testing**:
 - A stronger testing covers at least all the elements of the elements covered by a weaker testing.

Example

```
. int f1(int x,int y){  
. 1 while (x != y){  
. 2     if (x>y) then  
. 3         x=x-y;  
. 4     else y=y-x;  
. 5 }  
. 6 return x;      }
```


Example

- Test cases for branch coverage can be:
- $\{(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)\}$

Condition Coverage

- Test cases are designed such that:
 - Each component of a composite conditional expression
 - Given both true and false values.

Example

- Consider the conditional expression
 - $((c1.and.c2).or.c3):$
- Each of $c1$, $c2$, and $c3$ are exercised at least once,
 - i.e. given true and false values.

Branch Testing

- Branch testing is the simplest condition testing strategy:
 - Compound conditions appearing in different branch statements
 - Are given true and false values.

Branch testing

- Condition testing
 - Stronger testing than branch testing.
- Branch testing
 - Stronger than statement coverage testing.

statement < branch < condition

Condition coverage

- Consider a boolean expression having n components:
 - For condition coverage we require 2^n test cases.
- Condition coverage-based testing technique:
 - Practical only if n (the number of component conditions) is small.

Path Coverage

- Design test cases such that:
 - All linearly independent paths in the program are executed at least once.
- Defined in terms of
 - Control flow graph (CFG) of a program.

Path Coverage-Based Testing

- To understand the path coverage-based testing:
 - we need to learn how to draw control flow graph of a program.
- A control flow graph (CFG) describes:
 - the sequence in which different instructions of a program get executed.
 - the way control flows through the program.

How to Draw Control Flow Graph?

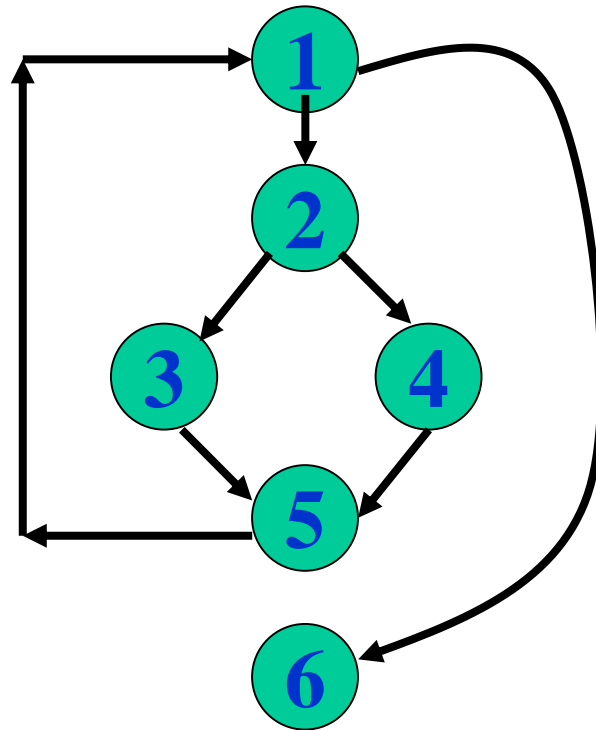
- Number all the statements of a program.
- Numbered statements:
 - Represent nodes of the control flow graph.
- An edge from one node to another node exists:
 - If execution of the statement representing the first node
 - Can result in transfer of control to the other node.

IMPORTANT

Example

```
. int f1(int x,int y){  
. 1 while (x != y){  
. 2     if (x>y) then  
. 3         x=x-y;  
. 4     else y=y-x;  
. 5 }  
. 6 return x;      }
```

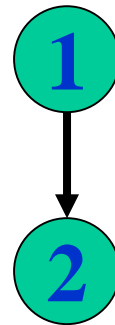
Example Control Flow Graph



How to draw Control flow graph?

. Sequence:

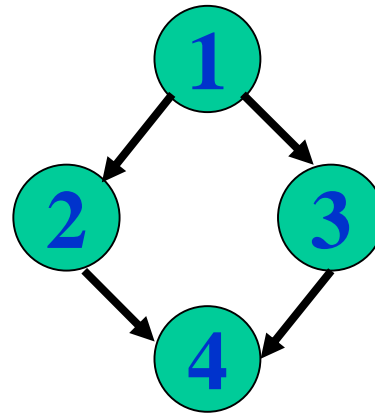
- 1 a=5;
- 2 b=a*b-1;



How to draw Control flow graph?

. Selection:

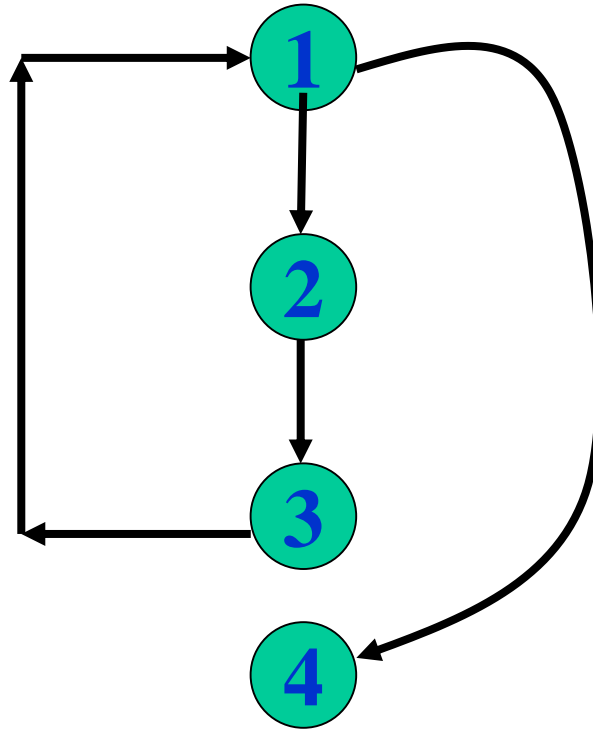
- 1 if(a>b) then
- 2 c=3;
- 3 else c=5;
- 4 c=c*c;



How to draw Control flow graph?

. Iteration:

- 1 while(a>b){
- 2 b=b*a;
- 3 b=b-1;}
- 4 c=b+d;



Path

- A **path** through a program:
 - A node and edge sequence from the starting node to a terminal node of the control flow graph.
 - There may be several terminal nodes for program.

Linearly Independent Path

- Any path through the program:
 - Introducing at least one new edge:
 - That is not included in any other independent paths.

Independent path

- It is straight forward:
 - To identify linearly independent paths of simple programs.
- For complicated programs:
 - It is not so easy to determine the number of independent paths.

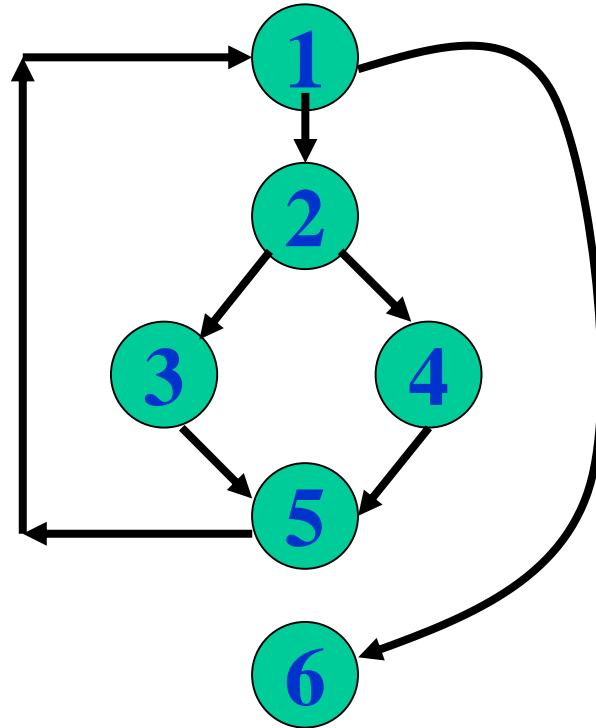
McCabe's Cyclomatic Metric

- An upper bound:
 - For the number of linearly independent paths of a program
- Provides a practical way of determining:
 - The maximum number of linearly independent paths in a program.

McCabe's Cyclomatic Metric

- Given a control flow graph G , cyclomatic complexity $V(G)$:
 - $V(G) = E - N + 2$
 - N is the number of nodes in G
 - E is the number of edges in G

Example Control Flow Graph

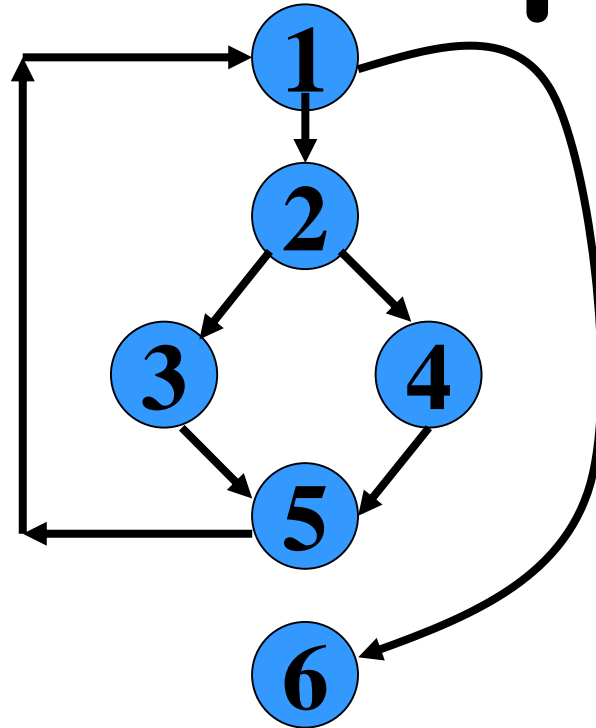


Cyclomatic complexity =
 $7 - 6 + 2 = 3.$

Cyclomatic Complexity

- Another way of computing cyclomatic complexity:
 - inspect control flow graph
 - determine number of bounded areas in the graph
- $V(G) = \text{Total number of bounded areas} + 1$
 - Any region enclosed by a nodes and edge sequence.

Example Control Flow Graph



Example

- From a visual examination of the CFG:
 - the number of bounded areas is 2.
 - cyclomatic complexity = $2+1=3$.

Cyclomatic complexity

- McCabe's metric provides:
 - A quantitative measure of testing difficulty and the ultimate reliability
- Intuitively,
 - Number of bounded areas increases with the number of decision nodes and loops.

Cyclomatic Complexity

- The first method of computing $V(G)$ is amenable to automation:
 - You can write a program which determines the number of nodes and edges of a graph
 - Applies the formula to find $V(G)$.

Cyclomatic complexity

- The cyclomatic complexity of a program provides:
 - A lower bound on the number of test cases to be designed
 - To guarantee coverage of all linearly independent paths.

Cyclomatic Complexity

- Defines the number of independent paths in a program.
- Provides a lower bound:
 - for the number of test cases for path coverage.

Cyclomatic Complexity

- Knowing the number of test cases required:
 - Does not make it any easier to derive the test cases,
 - Only gives an indication of the minimum number of test cases required.

Path Testing

- The **tester proposes**:
 - An initial set of test data using his experience and judgement.
- A **dynamic program analyzer** is used:
 - To indicate which parts of the program have been tested
 - The **output of the dynamic analysis**
 - used to **guide the tester in selecting additional test cases.**

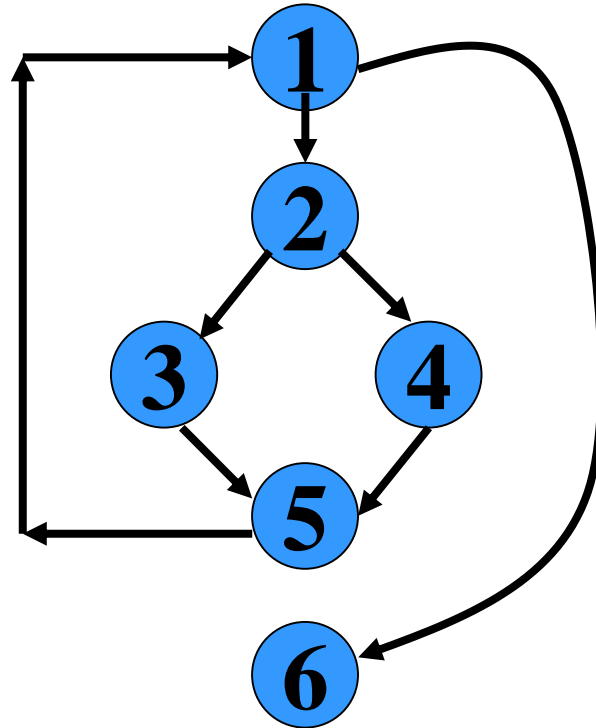
Derivation of Test Cases

- Let us discuss the steps:
 - to derive path coverage-based test cases of a program.
- Draw control flow graph.
- Determine $V(G)$.
- Determine the set of linearly independent paths.
- Prepare test cases:
 - to force execution along each path.

Example

```
. int f1(int x,int y){  
. 1 while (x != y){  
. 2     if (x>y) then  
. 3         x=x-y;  
. 4     else y=y-x;  
. 5 }  
. 6 return x;      }
```

Example Control Flow Diagram



Derivation of Test Cases

- Number of independent paths: 3
 - 1,6 test case (x=1, y=1)
 - 1,2,3,5,1,6 test case(x=1, y=2)
 - 1,2,4,5,1,6 test case(x=2, y=1)

An interesting application of cyclomatic complexity

- Relationship exists between:
 - McCabe's metric
 - The number of errors existing in the code,
 - The time required to find and correct the errors.

Cyclomatic Complexity

- Cyclomatic complexity of a program:
 - Also indicates the psychological complexity of a program.
 - Difficulty level of understanding the program.

Cyclomatic Complexity

- From maintenance perspective,
 - limit cyclomatic complexity
 - of modules to some reasonable value.
 - Good software development organizations:
 - restrict cyclomatic complexity of functions to a maximum of ten or so.

Summary

- Exhaustive testing of non-trivial systems is impractical:
 - We need to design an optimal set of test cases
 - Should expose as many errors as possible.
- If we select test cases randomly:
 - many of the selected test cases do not add to the significance of the test set.

Summary

- There are two approaches to testing:
 - black-box testing and
 - white-box testing.
- Designing test cases for black box testing:
 - does not require any knowledge of how the functions have been designed and implemented.
 - Test cases can be designed by examining only SRS document.

Summary

- White box testing:
 - requires knowledge about internals of the software.
 - Design and code is required.
- We have discussed a few white-box test strategies.
 - Statement coverage
 - branch coverage
 - condition coverage
 - path coverage

Summary

- A stronger testing strategy:
 - provides more number of significant test cases than a weaker one.
 - Condition coverage is strongest among strategies we discussed.
- We discussed McCabe's Cyclomatic complexity metric:
 - provides an upper bound for linearly independent paths
 - correlates with understanding, testing, and debugging difficulty of a program.