Software testing techniques and strategies

Er. Rudra Nepal

What is testing

- What is Testing?
- Many people understand many definitions of testing :
- 1. Testing is the process of demonstrating that errors are not present.
- 2. The purpose of testing is to show that a program performs its intended functions correctly.
- 3. Testing is the process of establishing confidence that a program does what it is supposed to do.

A more appropriate definition is:
“Testing is the process of executing a program with
the intent of finding errors prior to delivery to the end user.”

overview

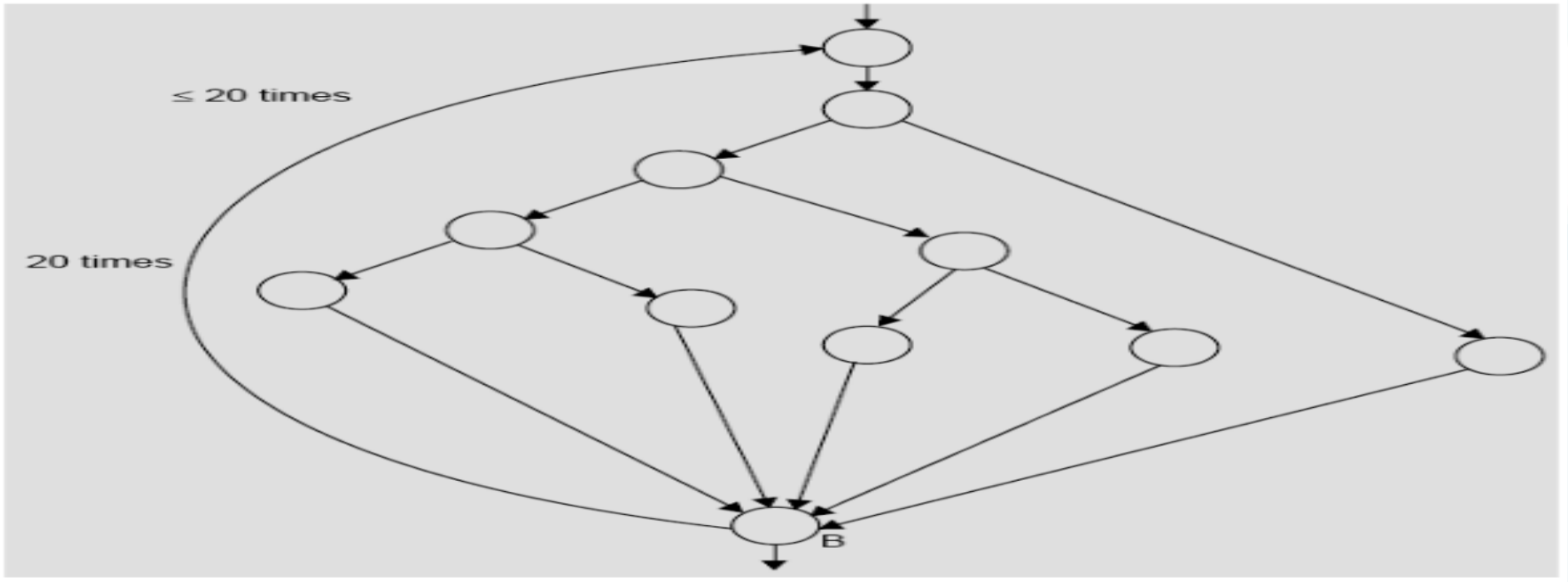
- **What is it?**
- Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to your customer.
- Software testing techniques provide systematic guidance for designing tests that
- (1) Exercise the internal logic of software components, and
- (2) Exercise the input and output domains of the program to uncover errors in program function, behavior and performance.
- **Who does it?**
- During early stages of testing, a software engineer performs all tests.
- However, as the testing process progresses, testing specialists may become involved.

- Testing requires the developers to find errors from their software.
- It is difficult for software developer to point out errors from own creations.
- Many organizations have made a distinction between development and testing phase by making different people responsible for each phase.
- A strategy for software testing is developed by the project manager,software engineers, and testing specialists.

- **Why should We Test ?**
- Although software testing is itself an expensive activity, yet launching of software without testing may lead to cost potentially much higher than that of testing, specially in systems where human safety is involved.
- In the software life cycle the earlier the errors are discovered and removed,the lower is the cost of their removal.

- **What are the steps?**
- Software is tested from two different perspectives:
- 1. Internal program logic is exercised using “white box” test case design techniques.
- 2. Software requirements are exercised using “black box” test case design techniques.

- **What should We Test ?**
- We should test the program's responses to every possible input. It means, we should test for all valid and invalid inputs. Suppose a program requires two 8 bit integers as inputs. Total possible combinations are 2 8 x2 8 . If only one second it required to execute one set of inputs, it may take 18 hours to test all combinations. Practically, inputs are more than two and size is also more than 8 bits. We have also not considered invalid inputs where so many combinations are possible. Hence, complete testing is just not possible, although, we may wish to do so.



The number of paths in the example of Fig. 1 are 10 14 or 100 trillions. It is computed from 5 ^20 + 5^19 + 5^18 + ... + 5^1 ; where 5 is the number of paths through the loop body. If only 5 minutes are required to test one test path, it may take approximately one billion years to execute every path.

- **Testing Objectives**
- 1. Testing is a process of executing a program with the intent of finding an error.
- 2. A good test case is one that has a high probability of finding an as-yet undiscovered error.
- 3. A successful test is one that uncovers an as-yet-undiscovered error.

- If testing is conducted successfully, it will uncover errors in the software. As a secondary benefit, testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met. In addition, data collected as testing is conducted provide a good indication of software reliability and some indication of software quality as a whole. But testing cannot show the absence of errors and defects, it can show only that software errors and defects are present. It is important to keep this statement in mind as testing is being conducted.

Testing Principles

- Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. Davis suggests a set1 of testing principles that have been adapted for use :
- All tests should be traceable to customer requirements. As we have seen, the objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.
- Tests should be planned long before testing begins. Test planning can begin as soon as the requirements model is complete. Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.
- The Pareto principle applies to software testing. Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.

- Testing should begin “in the small” and progress toward testing “in the large.” The first tests planned and executed generally focus on individual components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system .
- Exhaustive testing is not possible. The number of path permutations for even a moderately sized program is exceptionally large . For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.
- To be most effective, testing should be conducted by an independent third party. By most effective, we mean testing that has the highest probability of finding errors (the primary objective of testing).

Testability

- Software testability is simply how easily [a computer program] can be tested.
- The checklist that follows provides a set of characteristics that lead to testable software.
- **Operability.** "The better it works, the more efficiently it can be tested."
- The system has few bugs (bugs add analysis and reporting overhead to the test process
- No bugs block the execution of tests.
- The product evolves in functional stages (allows simultaneous development and testing).
- **Observability.** "What you see is what you test."
- Distinct output is generated for each input.
- System states and variables are visible or queriable during execution.
- Past system states and variables are visible or queriable (e.g., transaction logs).
- All factors affecting the output are visible.
- Incorrect output is easily identified.
- Internal errors are automatically detected through self-testing mechanisms.
- Internal errors are automatically reported.
- Source code is accessible.

- Controllability. "The better we can control the software, the more the testing can be automated and optimized."
- All possible outputs can be generated through some combination of input.
- All code is executable through some combination of input.
- Software and hardware states and variables can be controlled directly by the test engineer.
- Input and output formats are consistent and structured.
- Tests can be conveniently specified, automated, and reproduced.
- Decomposability. "By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting."
- The software system is built from independent modules.
- Software modules can be tested independently.

- **Simplicity.** "The less there is to test, the more quickly we can test it."
- Functional simplicity (e.g., the feature set is the minimum necessary to meet requirements).
- Structural simplicity (e.g., architecture is modularized to limit the propagation of faults).
- Code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).
- **Stability.** "The fewer the changes, the fewer the disruptions to testing.
- Changes to the software are infrequent.
- Changes to the software are controlled.
- Changes to the software do not invalidate existing tests.
- The software recovers well from failures.
- **Understandability.** "The more information we have, the smarter we will test."
- The design is well understood.
- Dependencies between internal, external, and shared components are well understood.
- Changes to the design are communicated.
- Technical documentation is instantly accessible.
- Technical documentation is well organized.
- Technical documentation is specific and detailed.
- Technical documentation is accurate.

attributes of a “good” test:

1. A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail. Ideally, the classes of failure are probed. For example, one class of potential failure in a GUI (graphical user interface) is a failure to recognize proper mouse position. A set of tests would be designed to exercise the mouse in an attempt to demonstrate an error in mouse position recognition.

2. A good test is not redundant. Testing time and resources are limited. There is no point in conducting a test that has the same purpose as another test. Every test should have a different purpose (even if it is subtly different). For example, a module of the SafeHome software is designed to recognize a user password to activate and deactivate the system. In an effort to uncover an error in password input, the tester designs a series of tests that input a sequence of passwords. Valid and invalid passwords (four numeral sequences) are input as separate tests. However, each valid/invalid password should probe a different mode of failure. For example, the invalid password 1234 should not be accepted by a system programmed to recognize 8080 as the valid password. If it is accepted, an error is present. Another test input, say 1235, would have the same purpose as 1234 and is therefore redundant. However, the invalid input 8081 or 8180 has a subtle difference, attempting to demonstrate that an error exists for passwords "close to" but not identical with the valid password.

- 3. A good test should be “best of breed”. In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

- 4. A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

- **Some Terminologies**
- Error, Mistake, Bug, Fault and Failure
- People make errors. A good synonym is mistake. This may be a syntax error or misunderstanding of specifications. Sometimes, there are logical errors.
- When developers make mistakes while coding, we call these mistakes “bugs”.
- A fault is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc. Defect is a good synonym for fault.
- A failure occurs when a fault executes. A particular fault may cause different failures, depending on how it has been exercised.

Test, Test Case and Test Suite

- Test and Test case terms are used interchangeably. In practice, both are same and are treated as synonyms. Test case describes an input description and an expected output description.

Test Case ID	
Section-I (Before Execution)	Section-II (After Execution)
Purpose :	Execution History:
Pre condition: (If any)	Result:
Inputs:	If fails, any possible reason (Optional);
Expected Outputs:	Any other observation:
Post conditions:	Any suggestion:
Written by:	Run by:
Date:	Date:

The set of test cases is called a test suite. Hence any combination of test cases may generate a test suite.

- **Verification and Validation**
- Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.
- Validation is the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements .
- Testing= Verification+Validation

TEST CASE DESIGN STRATEGIES

- Black-box or behavioral testing (knowing the specified function a product is to perform and demonstrating correct operation based solely on its specification without regard for its internal logic)

- White-box or glass-box testing (knowing the internal workings of a product, tests are performed to check the workings of all independent logic paths)

White box testing

White-box testing , sometimes called glass-box testing or code testing, is a test case design method that uses the control structure of the procedural design to derive test cases. (focus is on data flow, control flow ,information flow, coding practice , exception and error handling)Using white-box testing methods, the software engineer can derive test cases that

1. Guarantee that all independent paths within a module have been exercised at least once.

2. Exercise all logical decisions on their true and false sides,

3. Execute all loops at their boundaries and within their operational bounds, and

4. Exercise internal data structures to ensure their validity

A reasonable question might be posed at this juncture: "Why spend time and energy worrying about (and testing) logical minutiae when we might better expend effort ensuring that program requirements have been met?" Stated another way, why don't we spend all of our energy on black-box tests? The answer lies in the nature of software defects :

- Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed. Errors tend to creep into our work when we design and implement function, conditions, or control that are out of the mainstream. Everyday processing tends to be well understood (and well scrutinized), while "special case" processing tends to fall into the cracks.

- We often believe that a logical path is not likely to be executed when, in fact, it may be executed on a regular basis. The logical flow of a program is sometimes counter intuitive, meaning that our unconscious assumptions about flow of control and data may lead us to make design errors that are uncovered only once path testing commences.

- Typographical errors are random. When a program is translated into programming language source code, it is likely that some typing errors will occur. Many will be uncovered by syntax and type checking mechanisms, but others may go undetected until testing begins. It is as likely that a typo will exist on an obscure logical path as on a mainstream path.

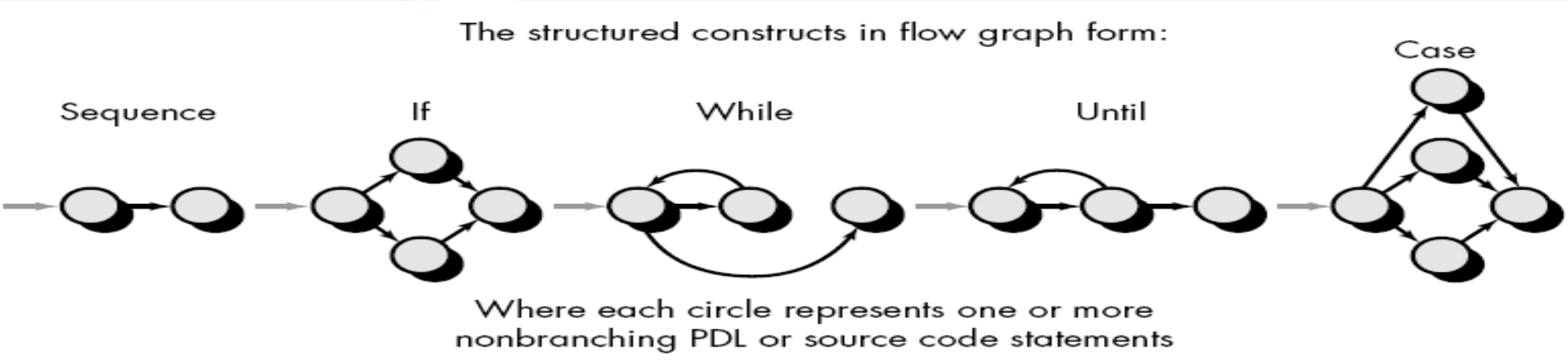
Each of these reasons provides an argument for conducting white-box tests. Blackbox testing, no matter how thorough, may miss the kinds of errors noted here. Whitebox testing is far more likely to uncover them.

Basis Path Testing

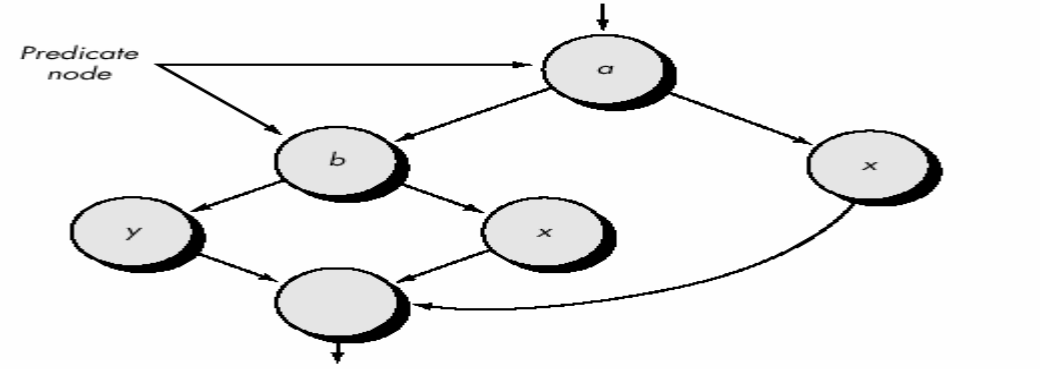
Basis path testing is a white-box testing technique. The basis path method enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

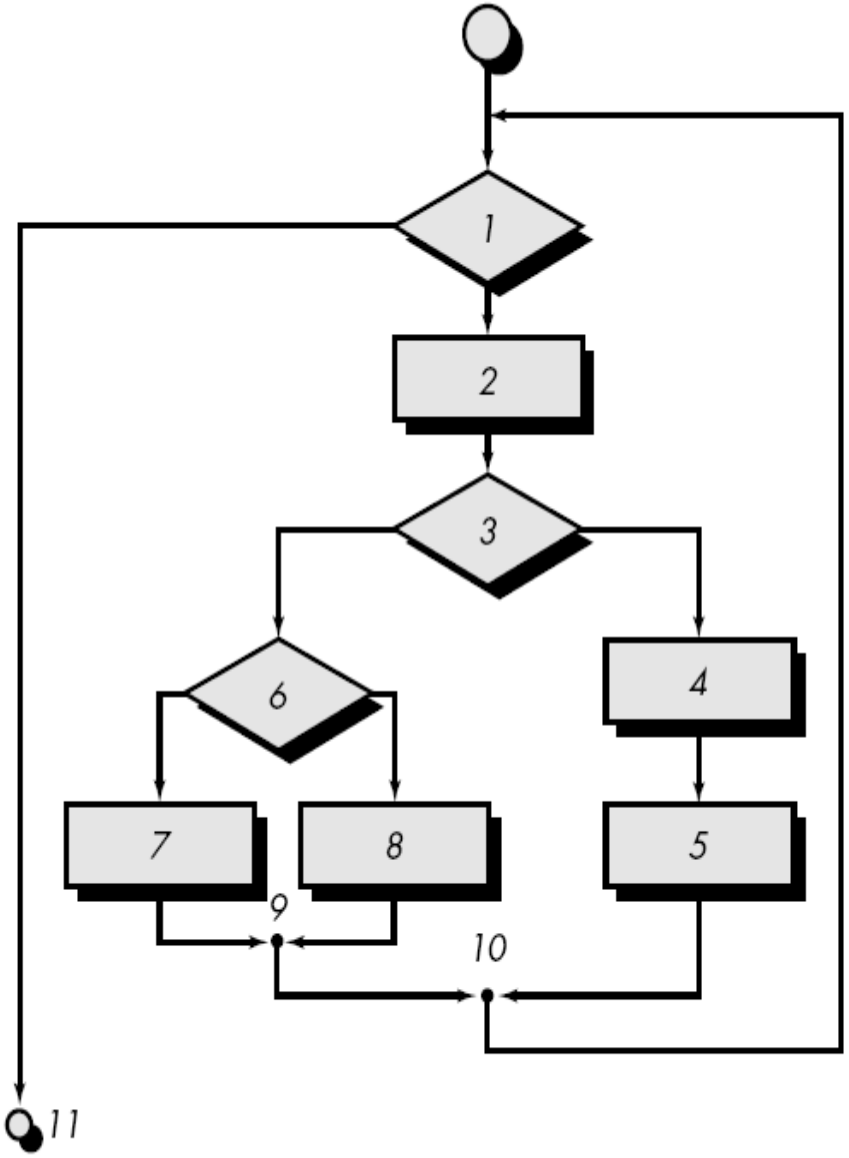
Flow Graph Notation

- **Before the basis path method can be introduced, a simple notation for the representation of control flow, called a flow graph (or program graph) must be introduced. The flow graph depicts logical control flow using the notation illustrated in figure. Each structured construct has a corresponding flow graph symbol. To illustrate the use of a flow graph, we consider the procedural design representation in figure below. Here, a flowchart is used to depict program control structure.**

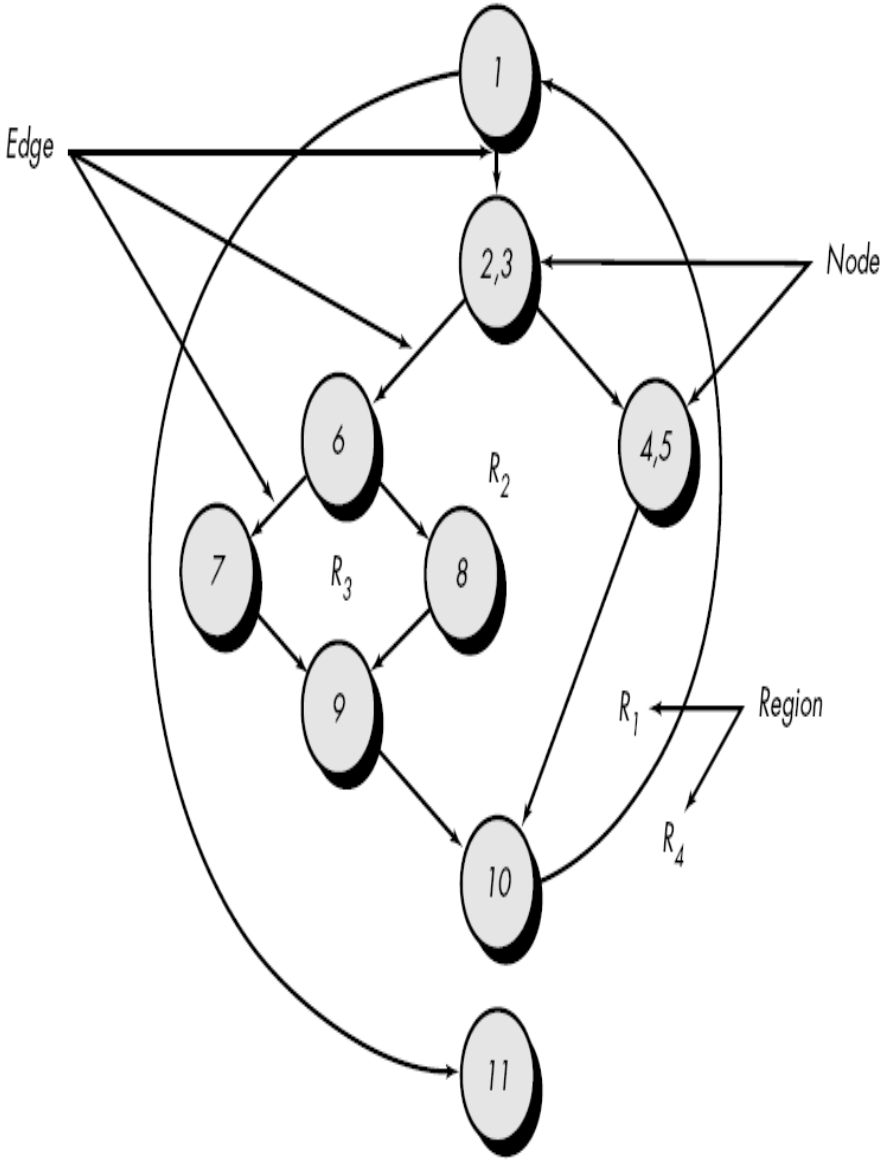


It is a simple notation for the depiction of the control flow.
Node – Each circle in the flow graph represents one or more procedural statements.
Edge – They represent the flow of control. An edge must always terminate on a node.
Region – An area bounded by edges and nodes is called a region. The area outside the graph
is also considered as a region.
Predicate Node – each node that contains a condition is called a predicate node and is characterized by two or more no of branches emerging from it.





(A)



(B)

Cyclomatic Complexity

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in figure (B) (above) is

path 1: 1-11

path 2: 1-2-3-4-5-10-1-11

path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

Note that each new path introduces a new edge. The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Complexity is computed in one of three ways:

1. The number of regions of the flow graph correspond to the cyclomatic complexity.

2. Cyclomatic complexity, V(G), for a flow graph, G, is defined as

$$V(G) = E - N + 2$$

where E is the number of flow graph edges, N is the number of flow graph nodes.

3. Cyclomatic complexity, V(G), for a flow graph, G, is also defined as

$$V(G) = P + 1$$

where P is the number of predicate nodes contained in the flow graph G.

Referring once more to the flow graph infigure (B), the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.

2. V(G) = 11 edges 9 nodes + 2 = 4.

3. V(G) = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in figure (B) is 4. More important, the value for V(G) provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

Derive Test Cases

Basis path testing can be directly enforced on the source code or it can be applied to the procedural design to derive the test cases. Let us see step by step procedure of basis path testing:

Step 1: The first step is to draw a flow graph by reviewing the source code of the program or its procedural design.

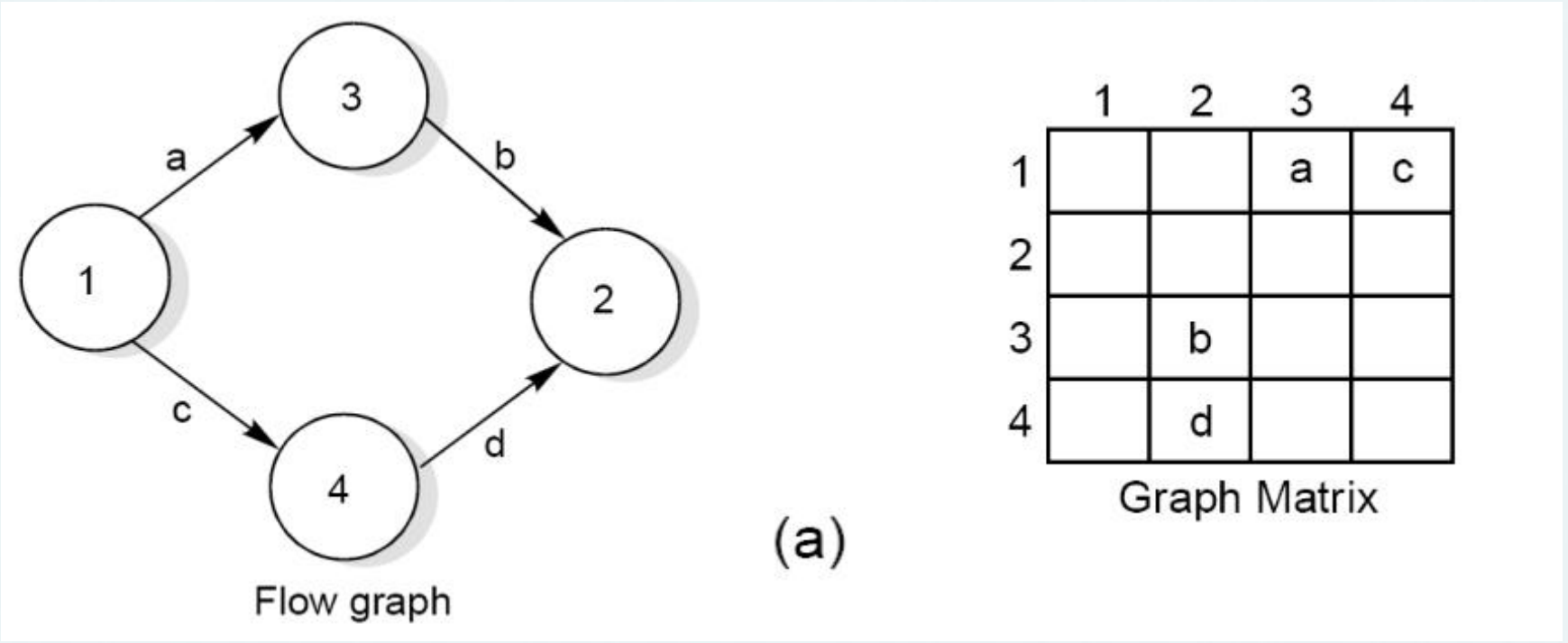
Step 2: Reviewing the flow graph you have to compute cyclomatic complexity.

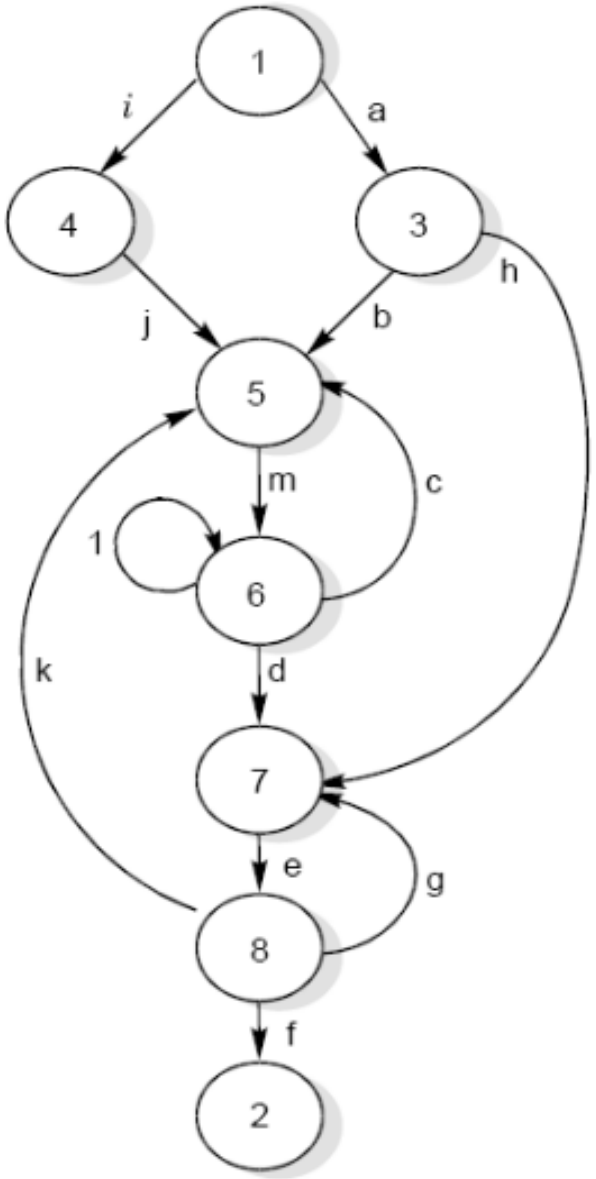
Step 3: The value computed by cyclomatic complexity provides the number up to which independent paths could be derived to determine the basis set.

Step 4: Reviewing the basis set testers design the test case that will exercise each statement in the program and every side of conditional statements in the program.

Graph Matrices

A graph matrix is a square matrix with one row and one column for every node in the graph. The size of the matrix (i.e., the number of rows and columns) is equal to the number of nodes in the flow graph. Some examples of graphs and associated matrices are shown in fig.





Flow graph

(c)

	1	2	3	4	5	6	7	8
1			a	i				
2								
3					b		h	
4					j			
5						m		
6					c	l	d	
7								e
8		f			k		g	

Graph Matrix

	1	2	3	4	5	6	7	8	Connections
1			1	1					2 – 1 = 1
2									
3					1		1		2 – 1 = 1
4					1				1 – 1 = 0
5						1			1 – 1 = 0
6					1	1	1		3 – 1 = 2
7								1	1 – 1 = 0
8		1			1		1		3 – 1 = 2
									6 + 1 = 7

Fig. Connection matrix

CONTROL STRUCTURE TESTING

The basis path testing technique is one of a number of techniques for white box testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. In this section, other variations on control structure testing are discussed. These broaden testing coverage and improve quality of white-box testing.

Condition Testing

Condition testing is a test case design method that exercises the logical conditions contained in a program module. The condition testing method focuses on testing each condition in the program. The purpose of condition testing is to detect not only errors in the conditions of a program but also other errors in the program.

Data Flow Testing

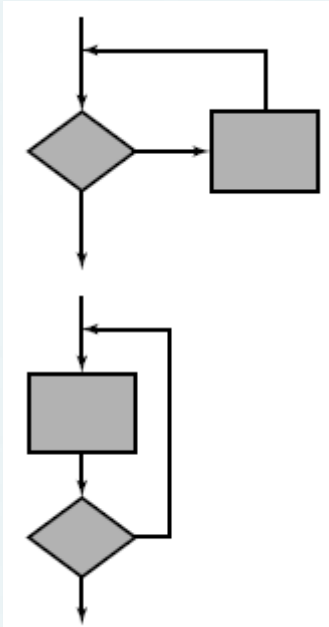
The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program.

Loop Testing

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops.

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.

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. m passes through the loop where m < n.
5. n -1, n, n + 1 passes through the loop.



Simple loops

Nested loops:

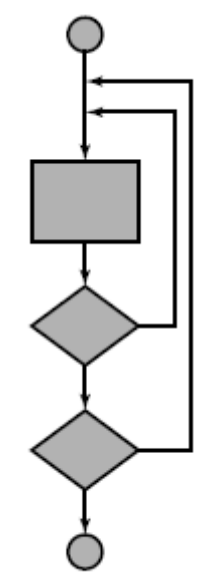
1. Start at the innermost loop. Set all other loops to minimum values.

2. 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.

3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.

4. Continue until all loops have been tested.



Nested loops

Concatenated loops:

Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent. When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured loops.

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

programming constructs.