Software Engineering

Lecture 09
Testing Tactics

Testing Fundamental

- Software engineer attempts to build software from an abstract concept to a tangible product. Next is Testing.
- The engineer creates a series of test cases that are intended to "demolish" the software that has been built.
- In fact, testing is the one step in the software process that could be viewed as destructive rather than constructive.

Testing Objective

- Primary Objective
  - Testing is a process of executing a program with the intent of finding an error.
  - A good test case is one that has a high probability of finding an as-yet undiscovered error.
  - A successful test is one that uncovers an as-yet-undiscovered error
- Objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.
- Testing demonstrates that software functions appear to be working according to specification, that behavioral and performance requirements appear to have been met.
- 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.

Testing Principles

- **All tests should be traceable to customer requirements.**
- **Tests should be planned long before testing begins.**
- **The Pareto principle applies to software testing.**
- **Testing should begin "in the small" and progress toward testing "in the large."**
- **Exhaustive testing is not possible.**
- **To be most effective, testing should be conducted by an independent third party.**

Software Testability

- S/w testability is simply how easily system or program or product can be tested.
- Testing must exhibit set of characteristics that achieve the goal of finding errors with a minimum of effort.

Characteristics of s/w Testability:

- Operability - "The better it works, the more efficiently it can be tested"
  - Relatively few bugs will block the execution of tests.
  - Allowing testing progress without fits and starts.

- **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.
  - Incorrect output is easily identified.
  - Internal errors are automatically detected & reported.
  - Source code is accessible.
- **Controllability** - "The better we can control the software, the more the testing can be automated and optimized."
  - Software and hardware states and variables can be controlled directly by the test engineer.
  - 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.
  - Independent 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.
- **Understandability –** "The more information we have, the smarter we will test."
  - Dependencies between internal, external, and shared components are well understood.
  - Changes to the design are communicated to testers.
  - Technical documentation is instantly accessible, well organized, specific and detailed, and accurate.

Testing attributes

1. A good test has a high probability of finding an error.
   - Tester must understand the software and attempt to develop a mental picture of how the software might fail.
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
   - Ex. Valid/ invalid 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.
4. A good test should be neither too simple nor too complex.
   - sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors.
   - Each test should be executed separately

TEST CASE DESIGN

- Objectives of testing, we must design tests that have the highest likelihood of finding the most errors with a minimum amount of time and effort.
- Test case design methods provide a mechanism that can help to ensure the completeness of tests and provide the highest likelihood for uncovering errors in software.

Any product or system can be tested on two ways:

1. Knowing the specified function that a product has been designed to perform; tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function (Black Box)
2. knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed according to specifications and all internal components have been effectively exercised. (White box testing)

White box testing

- *White-box testing* of software is predicated on close examination of procedural detail.
- Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops.
- The "status of the program" may be examined at various points.
- White-box testing, sometimes called *glass-box testing*, is a test case design method that uses the control structure of the procedural design to derive test cases.

White box testing

Using this method, SE 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
4. Exercise internal data structures to ensure their validity.

Basis path testing

- *Basis path testing* is a white-box testing technique
- To derive a logical complexity measure of a procedural design.
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time.

Methods:

1. Flow graph notation
2. Independent program paths or Cyclomatic complexity
3. Deriving test cases
4. Graph Matrices

Flow Graph Notation

- Start with simple notation for the representation of *control flow (called flow graph).* It represent logical control flow.

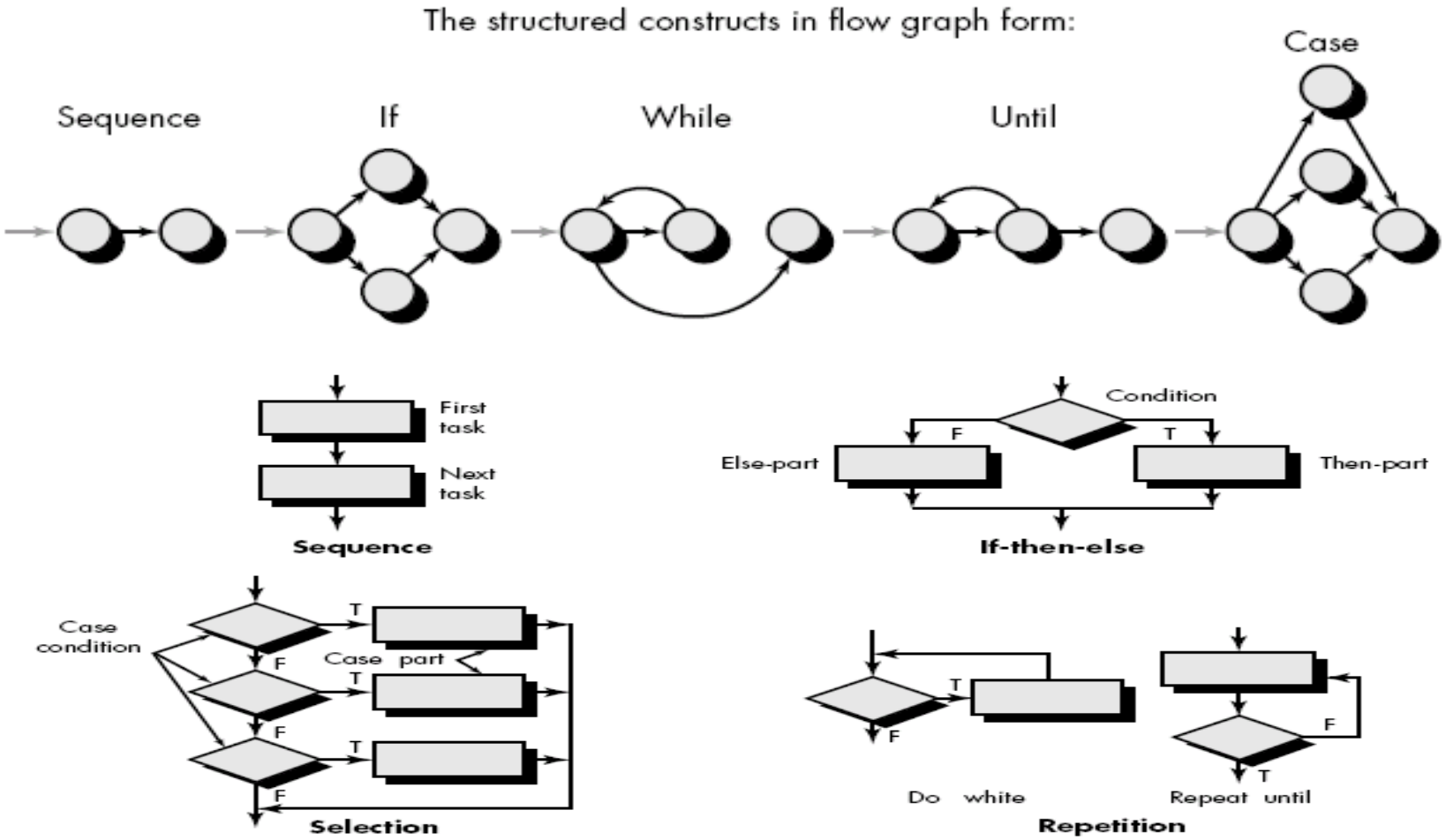
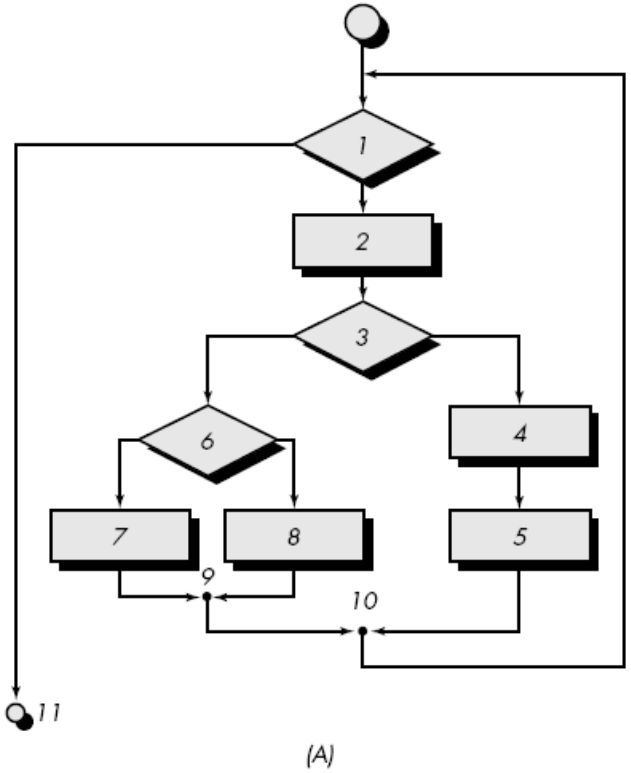
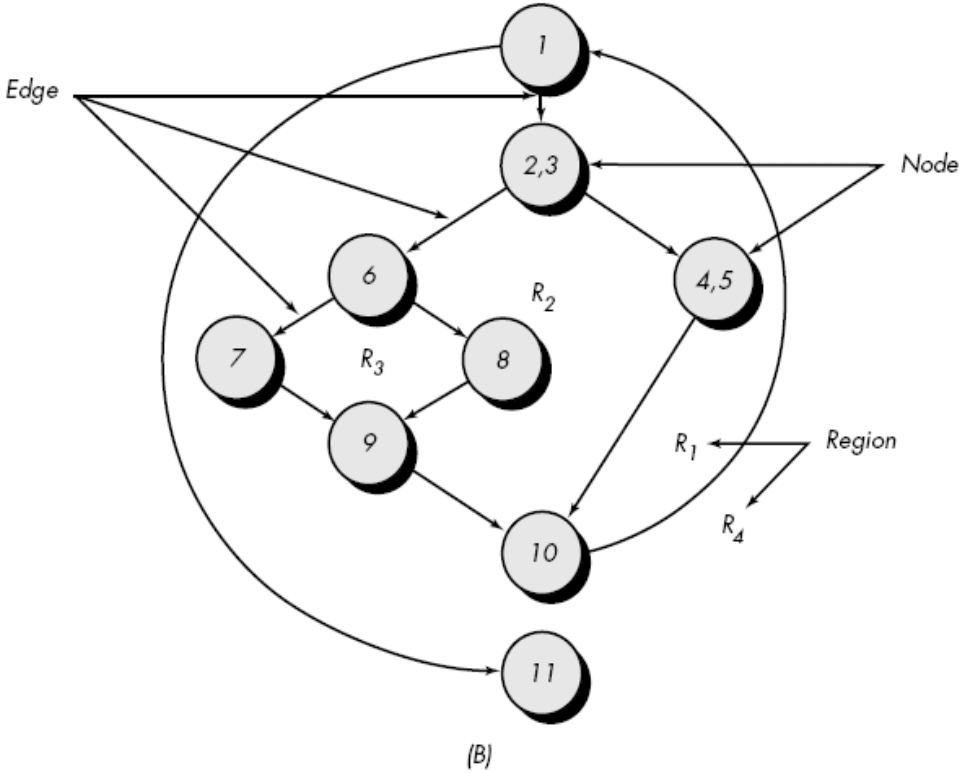


Fig. A represent program control structure and fig. B maps the flowchart into a corresponding flow graph.



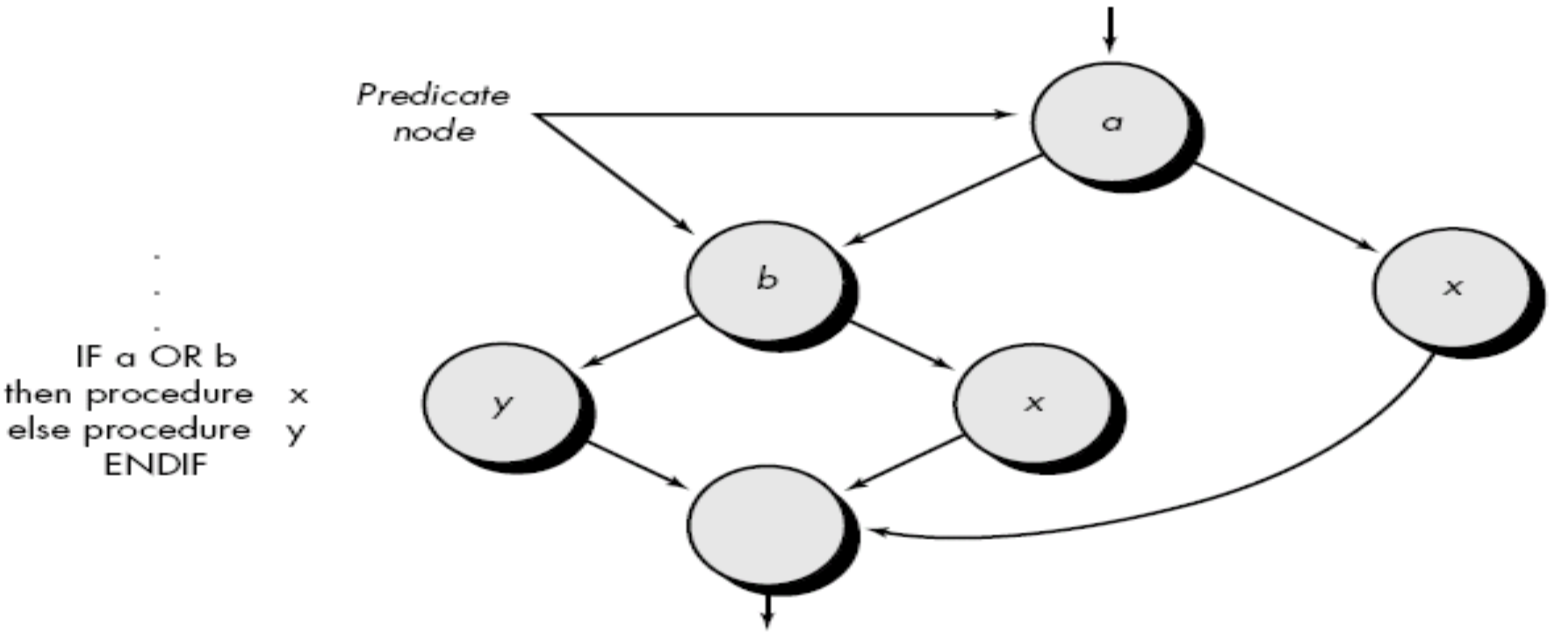
(A)



(B)

In fig. B each circle, called flow graph node, represent one or more procedural statement.

- A sequence of process boxes and decision diamond can map into a single node.
- The arrows on the flow graph, called edges or links, represent flow of control and are parallel to flowchart arrows.
- An edge must terminate at a node, even if the node does not represent any procedural statement.
- Areas bounded by edges and nodes are called *regions*. When counting regions, we include the are outside the graph as a region.
- When compound condition are encountered in procedural design, flow graph becomes slightly more complicated.



- When we translating PDL segment into flow graph, separate node is created for each condition.
- Each node that contains a condition is called *predicate node* and is characterized by two or more edges comes from it.

Independent program paths or Cyclomatic complexity

- An *independent path* is any path through the program that introduces at least one *new set of processing statement* or *new condition.*
- For example, a set of independent paths for flow graph:
  - Path 1: 1-11
  - Path 2: 1-2-3-4-5-10-1-11
  - Path 3: 1-2-3-6-8-9-1-11
  - Path 4: 1-2-3-6-7-9-1-11

  } Basis Set
- Note that each new path introduces a new edge.
- The path 1-2-3-4-5-10-1-2-3-6-8-9-1-11 is not considered to e an independent path because it is simply a combination of already specified paths and does not traverse any new edges.
- Test cases should be designed to force execution of these paths (basis set).
- Every statement in the program should be executed at least once and every condition will have been executed on its true and false.

- How do we know how many paths to looks for ?
- *Cyclomatic complexity* is a software metrics that provides a quantitative measure of the logical complexity of a program.
- It defines no. of independent paths in the basis set and also provides number of test that must be conducted.
- One of three ways to compute cyclomatic complexity:

1. The *no. of regions* corresponds 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 edges

So the value of V(G) provides us with upper bound of test cases.

Deriving Test Cases

- It is a series of steps method.
- The *procedure average* depicted in PDL.
- *Average*, an extremely simple algorithm, contains compound conditions and loops.

To derive basis set, follow the steps.

1. **Using the design or code as a foundation, draw a corresponding flow graph.**
    - A flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes.

Deriving Test Cases

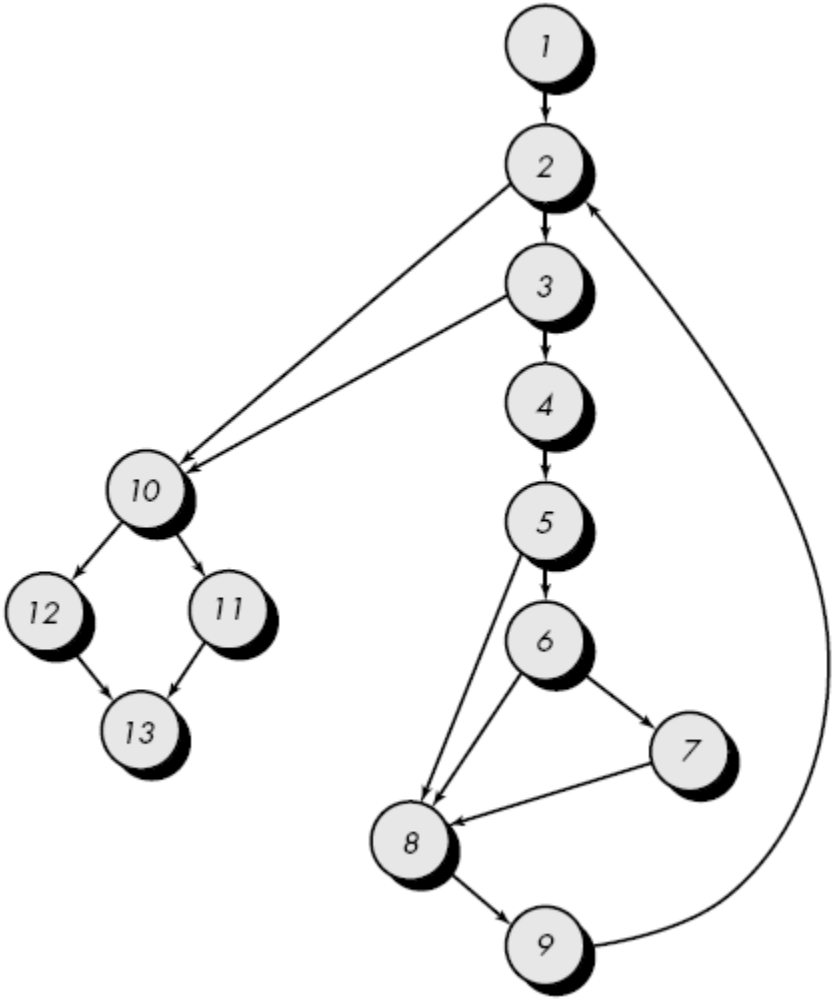
```
PROCEDURE average;

      *   This procedure computes the average of 100 or fewer
          numbers that lie between bounding values; it also computes the
          sum and the total number valid.

      INTERFACE RETURNS average, total.input, total.valid;
      INTERFACE ACCEPTS value, minimum, maximum;

      TYPE value[1:100] IS SCALAR ARRAY;
      TYPE average, total.input, total.valid;
          minimum, maximum, sum IS SCALAR;
      TYPE i IS INTEGER;
      i = 1;
      total.input = total.valid = 0;
      sum = 0;
      DO WHILE value[i] <> -999 AND total.input < 100
          increment total.input by 1;
          IF value[i] > = minimum AND value[i] < = maximum
              THEN increment total.valid by 1;
                   sum = s sum + value[i]
              ELSE skip
          ENDIF
          increment i by 1;
      ENDDO
      IF total.valid > 0
          THEN average = sum / total.valid;
          ELSE average = -999;
      ENDIF
END average
```

1 2 3 4 5 6 7 8 9 10 11 12 13



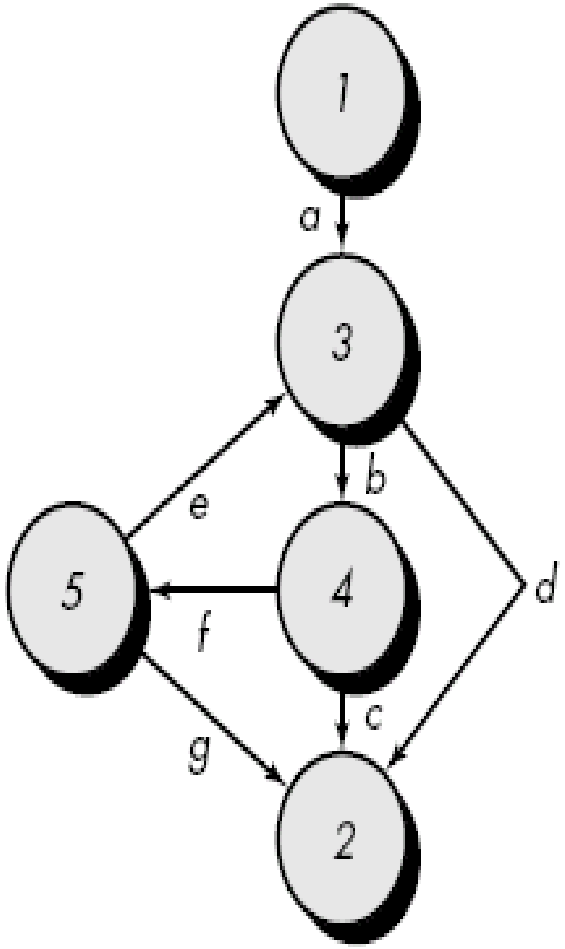
Flow graph for the procedure average

2. **Determine the cyclomatic complexity of the resultant flow graph.**
   2. *V*(*G*) can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure *average,* compound conditions count as two) and adding 1
   3. *V*(*G*) = 6 regions
   4. *V*(*G*) = 17 edges - 13 nodes + 2 = 6
   5. *V*(*G*) = 5 predicate nodes + 1 = 6
3. **Determine a basis set of linearly independent paths**
   3. The value of *V*(*G*) provides the number of linearly independent paths through the program control structure.
   4. path 1: 1-2-10-11-13
   5. path 2: 1-2-10-12-13
   6. path 3: 1-2-3-10-11-13
   7. path 4: 1-2-3-4-5-8-9-2-. . .
   8. path 5: 1-2-3-4-5-6-8-9-2-. . .
   9. path 6: 1-2-3-4-5-6-7-8-9-2-. . .
   10. The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable.

4. **Prepare test cases that will force execution of each path in the basis set.**
   - Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested.
   - Each test case is executed and compared to expected results.
   - Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

Graph Matrices

- A *graph matrix* is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph.
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- Each node on the flow graph is identify by numbers, while each edge is identify by letters.
- The graph matrix is nothing more than a tabular representation of a flow graph.
- By adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing.
- The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist).

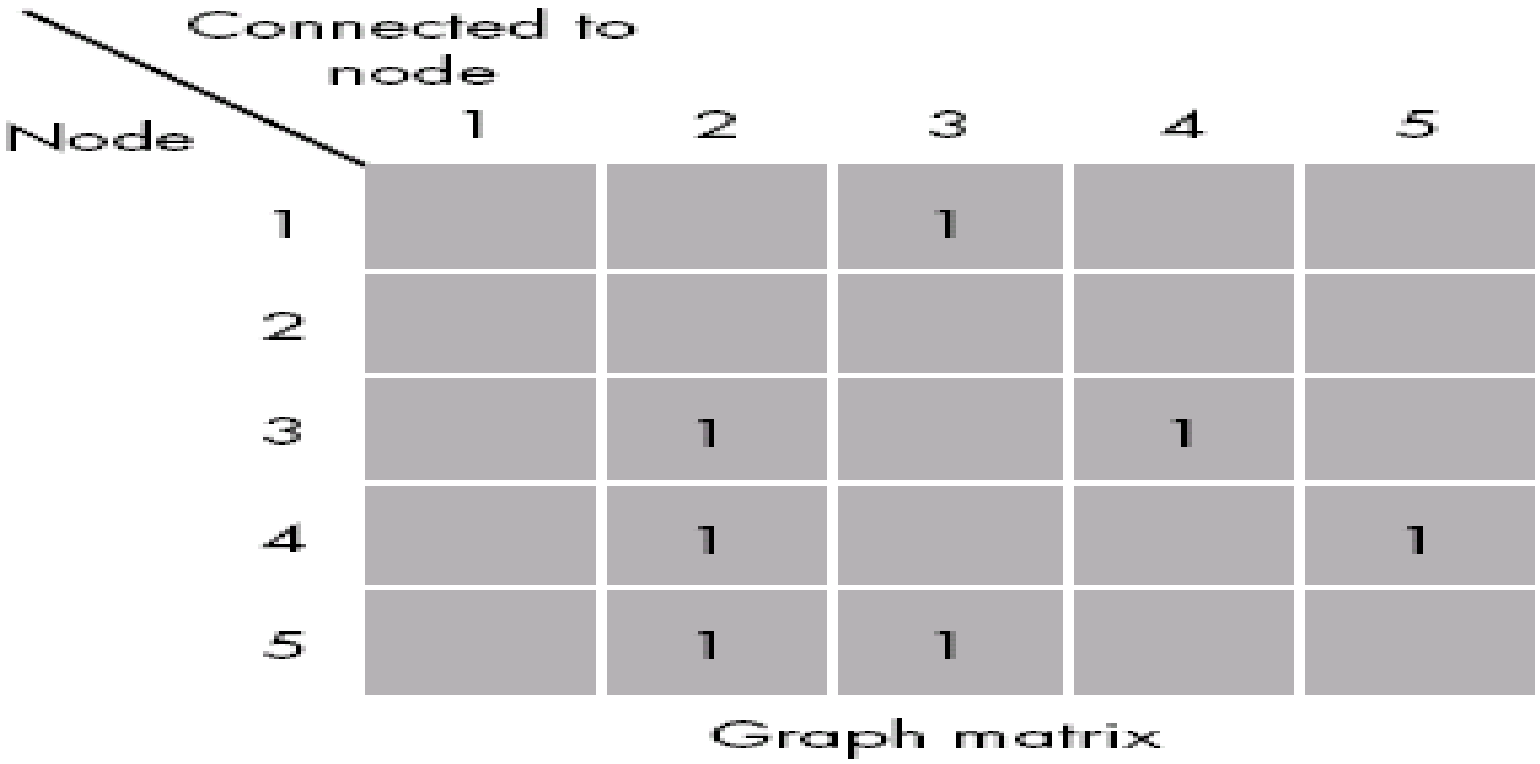


Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Connection matrix



Graph matrix

- Each letter has been replaced with a 1, indicating that a connection exists (this graph matrix is called a *connection matrix).*
- In fig.(connection matrix) each row with two or more entries represents a predicate node.
- We can directly measure cyclomatic complexity value by performing arithmetic operations
- Connections = Each row Total no. of entries – 1.
- V(G)= Sum of all connections + 1

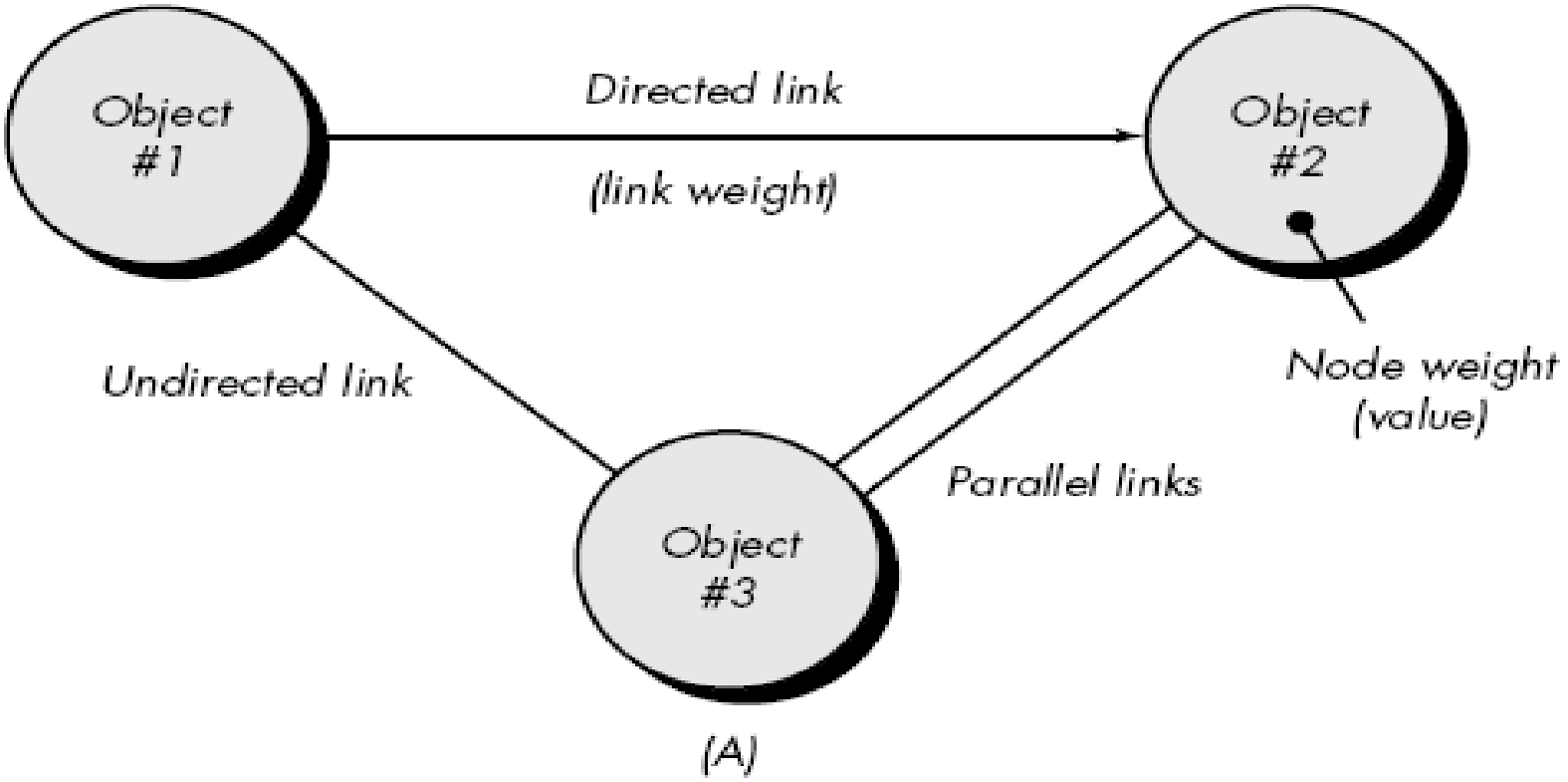
Black box testing

- Also called *behavioral testing*, focuses on the functional requirements of the software.
- It enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.
- Black-box testing is not an alternative to white-box techniques but it is complementary approach.
- Black-box testing attempts to find errors in the following categories:
  - Incorrect or missing functions,
  - Interface errors,
  - Errors in data structures or external data base access.
  - Behavior or performance errors,
  - Initialization and termination errors.

- Black-box testing purposely ignored control structure, attention is focused on the information domain. Tests are designed to answer the following questions:
  - How is functional validity tested?
  - How is system behavior and performance tested?
  - What classes of input will make good test cases?
- By applying black-box techniques, we derive a set of test cases that satisfy the following criteria
  - Test cases that reduce the number of additional test cases that must be designed to achieve reasonable testing (i.e minimize effort and time)
  - Test cases that tell us something about the presence or absence of classes of errors
- Black box testing methods
  - Graph-Based Testing Methods
  - Equivalence partitioning
  - Boundary value analysis (BVA)
  - Orthogonal Array Testing

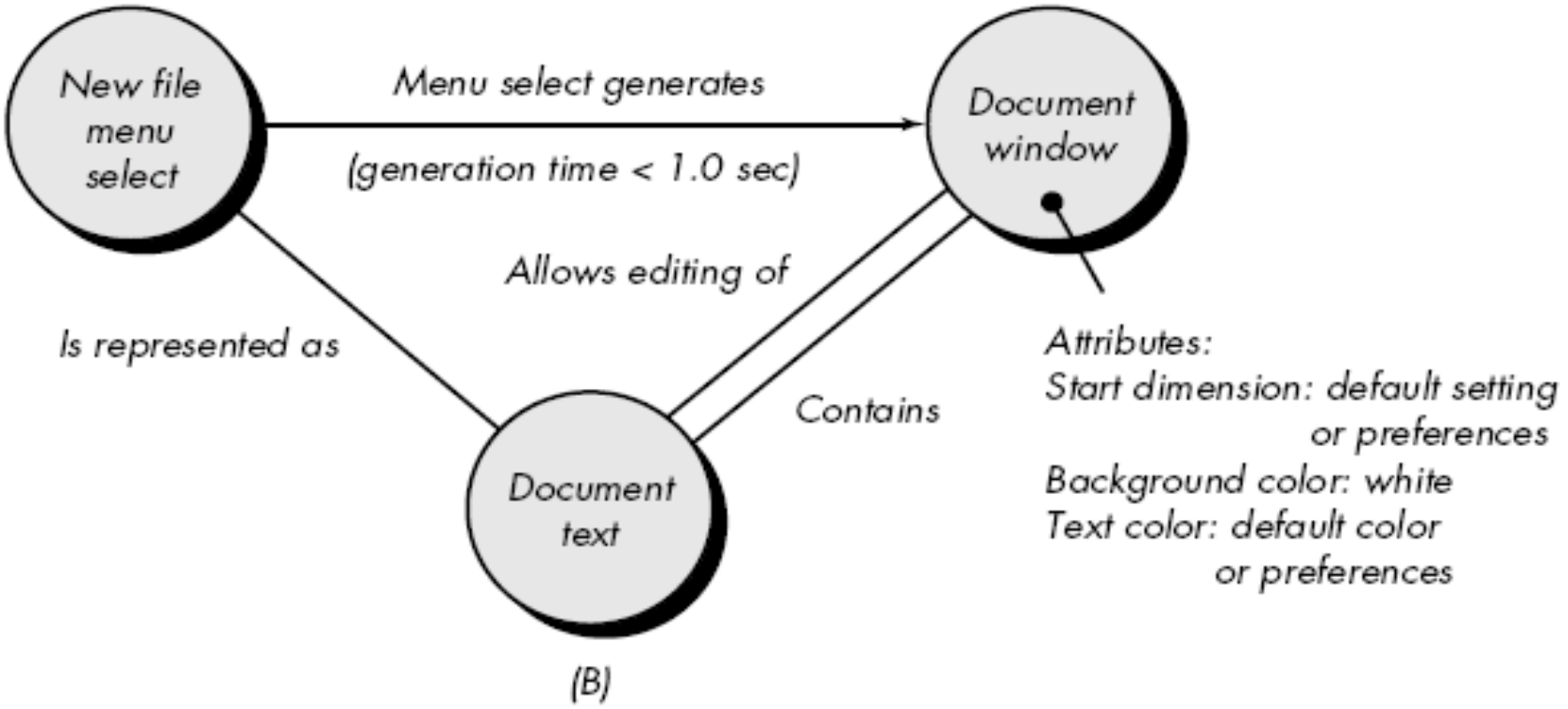
Graph-Based Testing Methods

- To understand the objects that are modeled in software and the relationships that connect these objects.
- Next step is to define a series of tests that verify “all objects have the expected relationship to one another.
- Stated in other way:
  - Create a graph of important objects and their relationships
  - Develop a series of tests that will cover the graph
- So that each object and relationship is exercised and errors are uncovered.
- Begin by creating graph –
  - a collection of *nodes* that represent objects
  - *links* that represent the relationships between objects
  - *node weights* that describe the properties of a node
  - *link weights* that describe some characteristic of a link.


Object
#1
Directed link
(link weight)
Object
#2
Undirected link
Node weight
(value)
Parallel links
Object
#3
(A)

- Nodes are represented as circles connected by links that take a number of different forms.
- A *directed link* (represented by an arrow) indicates that a relationship moves in only one direction.
- A *bidirectional link,* also called a *symmetric link*, implies that the relationship applies in both directions.
- *Parallel links* are used when a number of different relationships are established between graph nodes.

Example


New file menu select
Menu select generates
(generation time < 1.0 sec)
Document window
Allows editing of
Is represented as
Contains
Attributes:
Start dimension: default setting or preferences
Background color: white
Text color: default color or preferences
Document text
(B)

- *Object #1* = **new file menu select**
- *Object #2* = **document window**
- *Object #3* = **document text**

Referring to example figure, a menu select on **new file** generates a **document window.**

- The link weight indicates that the window must be generated in less than 1.0 second.
- The node weight of **document window** provides a list of the window attributes that are to be expected when the window is generated.
- An undirected link establishes a symmetric relationship between the **new file menu select** and **document text,**
- parallel links indicate relationships between **document window** and **document text**

- Number of behavioral testing methods that can make use of graphs:
- **Transaction flow modeling.**
  - The nodes represent steps in some transaction and the links represent the logical connection between steps
- **Finite state modeling.**
  - The nodes represent different user observable states of the software and the links represent the transitions that occur to move from state to state. (Starting point and ending point)
- **Data flow modeling.**
  - The nodes are data objects and the links are the transformations that occur to translate one data object into another.
- **Timing modeling.**
  - The nodes are program objects and the links are the sequential connections between those objects.
  - Link weights are used to specify the required execution times as the program executes.

Equivalence Partitioning

- *Equivalence partitioning* is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.
- Test case design for equivalence partitioning is based on an evaluation of *equivalence classes* for an input condition.
- An *equivalence class* represents a set of valid or invalid states for input conditions.
- Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition.

To define equivalence classes follow the guideline

1. If an input condition specifies a *range,* one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific *value,* one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a *set,* one valid and one invalid equivalence class are defined.
4. If an input condition is *Boolean,* one valid and one invalid class are defined.

Example

- area code—blank or three-digit number
- prefix—three-digit number not beginning with 0 or 1
- suffix—four-digit number
- password—six digit alphanumeric string
- commands— check, deposit, bill pay, and the like

- area code:
  - Input condition, *Boolean*—the area code may or may not be present.
  - Input condition, *value*— three digit number
- prefix:
  - Input condition, *range*—values defined between 200 and 999, with specific exceptions.
- Suffix:
  - Input condition, value—four-digit length
- password:
  - Input condition, *Boolean*—a password may or may not be present.
  - Input condition, *value*—six-character string.
- command:
  - Input condition, *set*— check, deposit, bill pay.

Boundary Value Analysis (BVA)

- Boundary value analysis is a test case design technique that complements equivalence partitioning.
- Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class.
- In other word, Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

Guidelines for BVA

1. If an input condition specifies a range bounded by values *a* and *b,* test cases should be designed with values *a* and *b* and just above and just below *a* and *b*.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions.
4. If internal program data structures have prescribed boundaries be certain to design a test case to exercise the data structure at its boundary

Orthogonal Array Testing

- The number of input parameters is small and the values that each of the parameters may take are clearly bounded.
- When these numbers are very small (e.g., three input parameters taking on three discrete values each), it is possible to consider every input permutation .
- However, as the number of input values grows and the number of discrete values for each data item increases (exhaustive testing occurs)
- *Orthogonal array testing* can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing.
- *Orthogonal Array Testing can be used to reduce the number of combinations and provide maximum coverage with a minimum number of test cases.*

Example

- Consider the *send* function for a fax application.
- Four parameters, P1, P2, P3, and P4, are passed to the *send* function. Each takes on three discrete values.
- P1 takes on values:
  - P1 = 1, send it now
  - P1 = 2, send it one hour later
  - P1 = 3, send it after midnight
- P2, P3, and P4 would also take on values of 1, 2 and 3, signifying other *send* functions.
- OAT is an array of values in which each column represents a Parameter - value that can take a certain set of values called levels.
- Each row represents a test case.
- Parameters are combined pair-wise rather than representing all possible combinations of parameters and levels

Test case	Test parameters			
	P_1	P_2	P_3	P_4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1