EDITH COWAN UNIVERSITY

CSI3105: Software Testing

Module 5a: Structural Testing Introduction

Creative thinkers made here.

Outline of the Chapter

ECU
EDITH COWAN UNIVERSITY

- Basic Idea
- Outline of Control Flow Testing
- Control Flow Graph
- Paths in a Control Flow Graph
- Path Selection Criteria
- Generating Test Input
- Containing Infeasible Paths
- Summary

Outline of the Chapter

ECU
EDITH COWAN UNIVERSITY

- Video 5b
  - Control flow graphs revisited
- Video 5c
  - Test selection criteria
- Video 5d
  - Generating test cases
- Video 5e
  - Basis paths and cyclomatic complexity

Basic Idea

ECU EDITH COWAN UNIVERSITY

Two kinds of basic program statements:

- Assignment statements (Ex. x = 2*y;)
- Conditional statements (Ex. if(), for(), while(), …)

Control flow

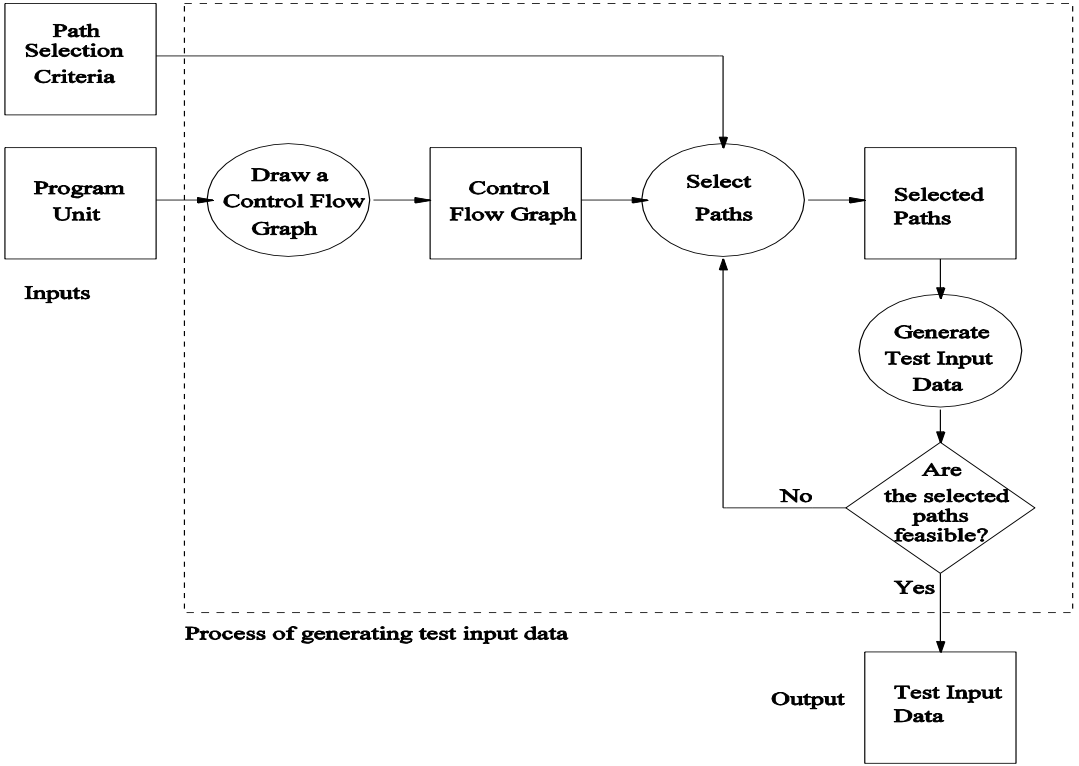
- Successive execution of program statements is viewed as flow of control.
- Conditional statements alter the default flow.

Program path

- A program path is a sequence of statements from entry to exit.
- There can be a large number of paths in a program.
- There is an (input, expected output) pair for each path.
- Executing a path requires invoking the program unit with the right test input.
- Paths are chosen by using the concepts of path selection criteria.

Tools: Automatically generate test inputs from program paths.

Outline of Control Flow Testing



The process of generating test input data for control flow testing.

Outline of Control Flow Testing

EDITH COWAN UNIVERSITY

Inputs to the test generation process

- Source code
- Path selection criteria: statement, branch, …

Generation of control flow graph (CFG)

- A CFG is a graphical representation of a program unit.
- Compilers are modified to produce CFGs. (You can draw one by hand.)

Selection of paths

- Enough entry/exit paths are selected to satisfy path selection criteria.

Generation of test input data

- Two kinds of paths
  - Executable path: There exists input so that the path is executed.
  - Infeasible path: There is no input to execute the path.
- Solve the path conditions to produce test input for each path.

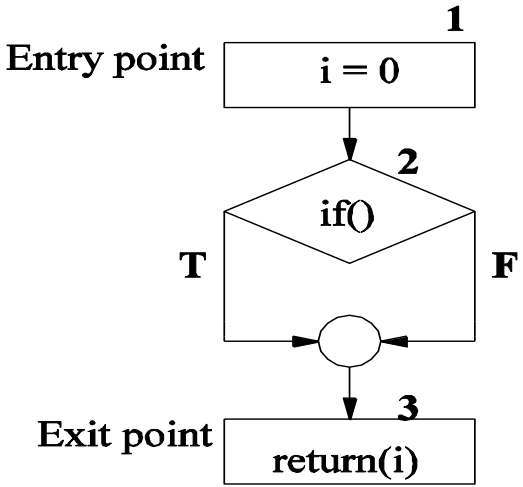
Control Flow Graph

Example code: openfiles()

```
FILE *fptr1, *fptr2, *fptr3; /* These are global variables. */
int openfiles(){
  /*
    This function tries to open files "file1", "file2", and "file3"
    for read access, and returns the number of files successfully
    opened. The file pointers of the opened files are put in the
    global variables.
  */
    int i = 0;
    if(
       ((( fptr1 = fopen("file1", "r")) != NULL) && (i++) && (0)) ||
       ((( fptr2 = fopen("file2", "r")) != NULL) && (i++) && (0)) ||
       ((( fptr3 = fopen("file3", "r")) != NULL) && (i++))
    );
    return(i);
}
```

Figure 4.3: A function to open three files.

Control Flow Graph



A high-level CFG representation of openfiles().

Control Flow Graph

Symbols in a CFG

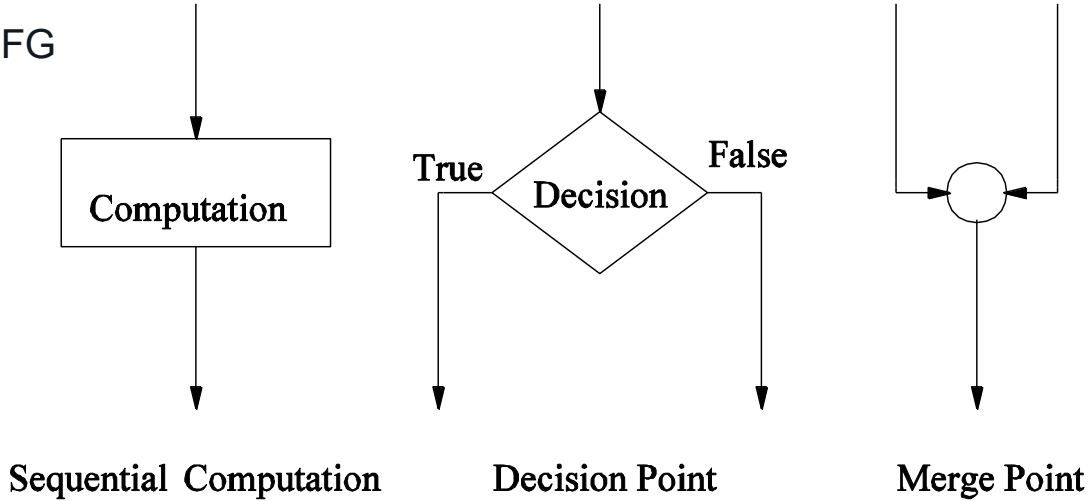


Figure 4.2: Symbols in a control flow graph

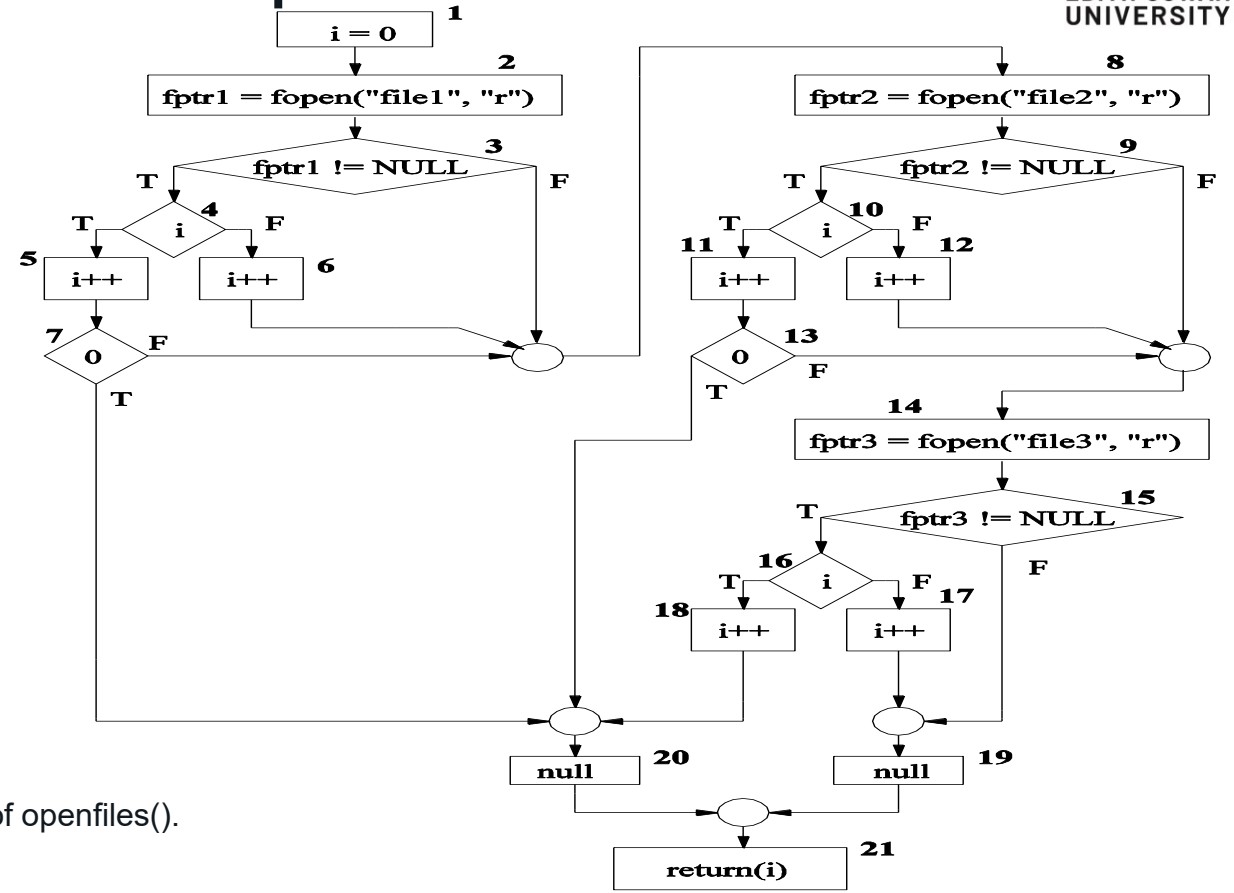
Control Flow Graph

Example code: openfiles()

```
FILE *fptr1, *fptr2, *fptr3; /* These are global variables. */
int openfiles(){
  /*
    This function tries to open files "file1", "file2", and "file3"
    for read access, and returns the number of files successfully
    opened. The file pointers of the opened files are put in the
    global variables.
  */
    int i = 0;
    if(
       ((( fptr1 = fopen("file1", "r")) != NULL) && (i++) && (0)) ||
       ((( fptr2 = fopen("file2", "r")) != NULL) && (i++) && (0)) ||
       ((( fptr3 = fopen("file3", "r")) != NULL) && (i++))
    );
    return(i);
}
```

Figure 4.3: A function to open three files.

Control Flow Graph



A detailed CFG representation of openfiles().

Control Flow Graph

Example code: ReturnAverage()
public static double ReturnAverage(int value[], int AS, int MIN, int MAX){
/* Function: ReturnAverage Computes the average of all those numbers in the input array in the positive range [MIN, MAX]. The maximum size of the array is AS. But, the array size could be smaller than AS in which case the end of input is represented by -999.
*/

```
public static double ReturnAverage(int value[],  int AS, int MIN, int MAX){
int i, ti, tv, sum;
        double av;
        i = 0; ti = 0; tv = 0; sum = 0;
        while (ti < AS && value[i] != -999) {
          ti++;
          if (value[i] >= MIN && value[i] <= MAX) {
            tv++;
            sum = sum + value[i];
          }
          i++;
        }
        if (tv > 0)
          av = (double)sum/tv;
        else
          av = (double) -999;
        return (av);
    }
```

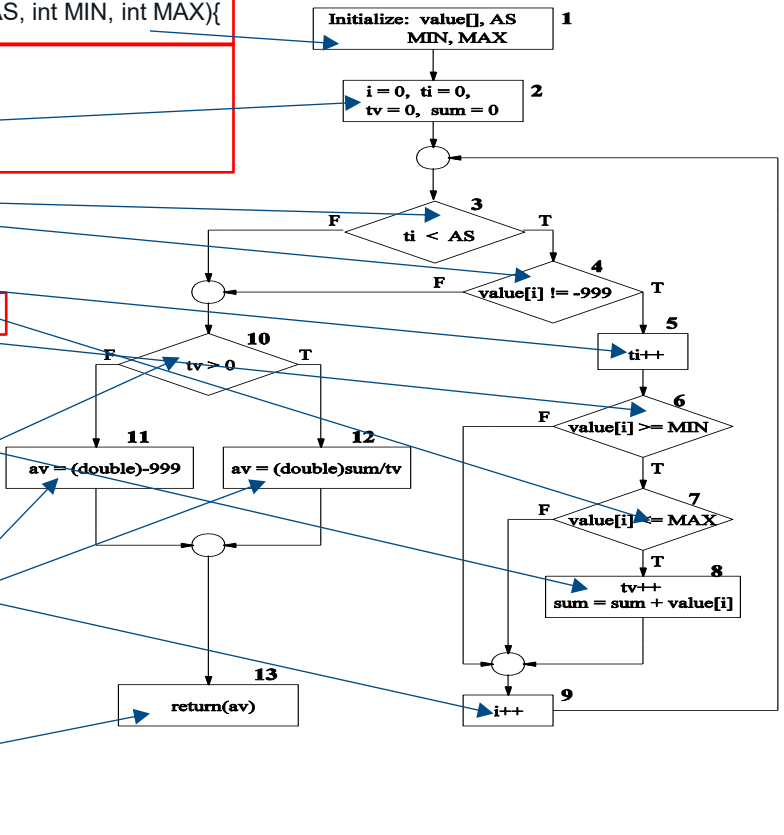
Control Flow Graph

EDITH COWAN UNIVERSITY

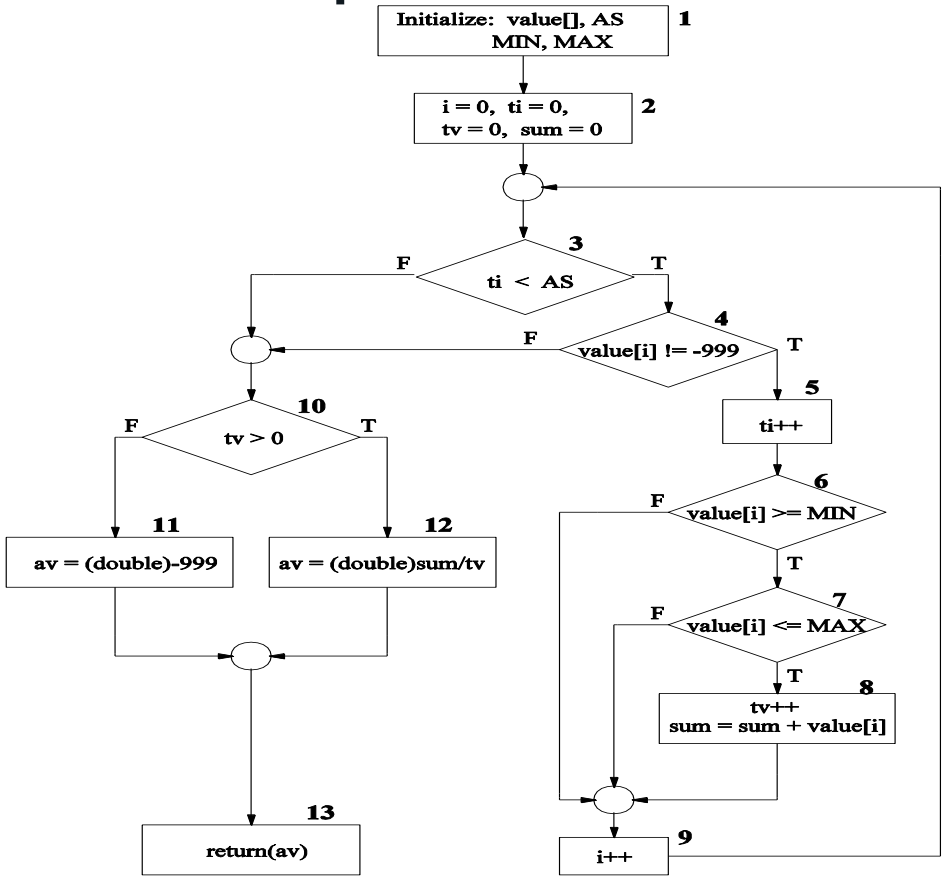
```
public static double ReturnAverage(int value[],  int AS, int MIN, int MAX){
int i, ti, tv, sum;
        double av;
        i = 0; ti = 0; tv = 0; sum = 0;
        while (ti < AS && value[i] != -999) {
          ti++;
          if (value[i] >= MIN && value[i] <= MAX) {
            tv++;
            sum = sum + value[i];
          }
          i++;
        }
        if (tv > 0)
          av = (double)sum/tv;
        else
          av = (double) -999;
        return (av);
    }
```

Control Flow Graph

```
public static double ReturnAverage(int value[],  int AS, int MIN, int MAX){
int i, ti, tv, sum;
      double av;
      i = 0; ti = 0; tv = 0; sum = 0;
      while (ti < AS && value[i] != -999) {
        ti++;
        if (value[i] >= MIN && value[i] <= MAX) {
          tv++;
          sum = sum + value[i];
        }
        i++;
      }
      if (tv > 0)
        av = (double)sum/tv;
      else
        av = (double) -999;
      return (av);
}
```



Control Flow Graph

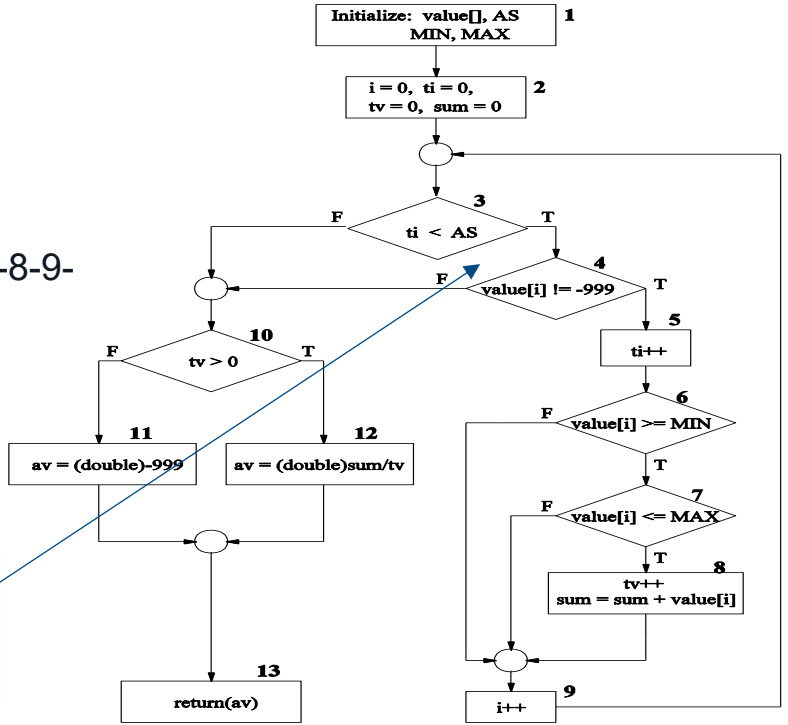


Paths in a Control Flow Graph

A few paths in Figure 4.7. (Table 4.1)

- Path 1: 1-2-3(F)-10(T)-12-13
- Path 2: 1-2-3(F)-10(F)-11-13
- Path 3: 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13
- Path 4: 1-2-3(T)-4(T)-5-6-7(T)-8-9-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12- 13

```
while (ti < AS && value[i] != -999) {
```



Path Selection Criteria

Program paths are selectively executed.

Question: What paths do I select for testing?

The concept of *path selection criteria* is used to answer the question.

Advantages of selecting paths based on defined criteria:

- Ensure that all program constructs are executed at least once.
- Repeated selection of the same path is avoided.
- One can easily identify what features have been tested and what not.

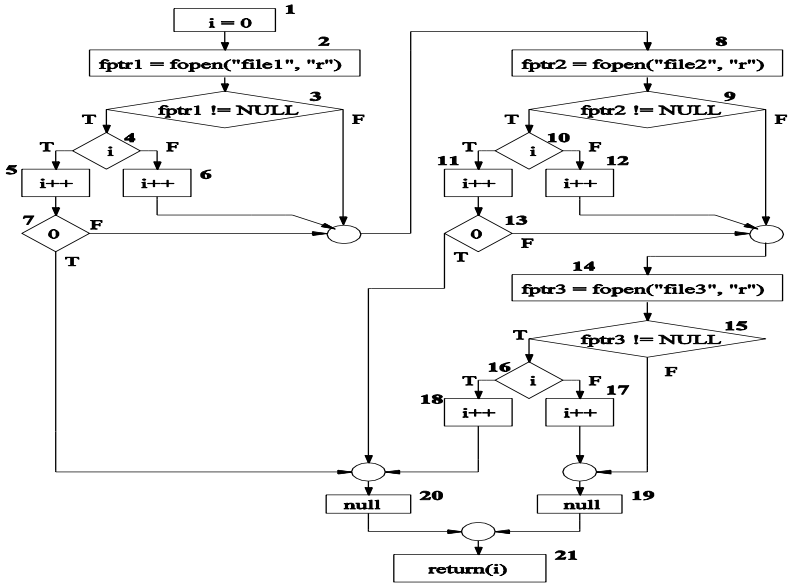
Path selection criteria

- Select all paths.
- Select paths to achieve complete statement coverage.
- Select paths to achieve complete branch coverage.
- Select paths to achieve predicate coverage.

Path Selection Criteria

All-path coverage criterion: Select all the paths in the program unit under consideration.

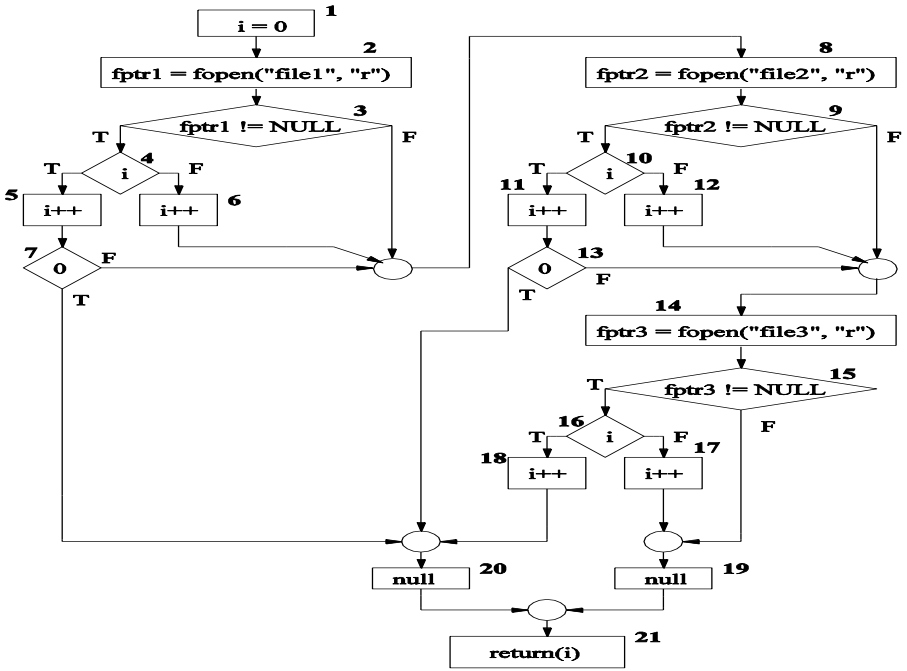
- The openfiles() unit has 25+ paths.



Existence of "file1"	Existence of "file2"	Existence of "file3"
No	No	No
No	No	Yes
No	Yes	No
No	Yes	Yes
Yes	No	No
Yes	No	Yes
Yes	Yes	No
Yes	Yes	Yes

- Selecting all the inputs will exercise all the program paths.

Path Selection Criteria



Input	Path
<No, No, No>	1-2-3(F)-8-9(F)-14-15(F)-19-21
<Yes, No, No>	1-2-3(T)-4(F)-6-8-9(F)-14-15(F)-19-21
<Yes, Yes, Yes>	1-2-3(T)-4(F)-6-8-9(T)-10(T)-11-13(F)-14- 15(T) -16(T)-18-20-21

Path Selection Criteria

EDITH COWAN UNIVERSITY

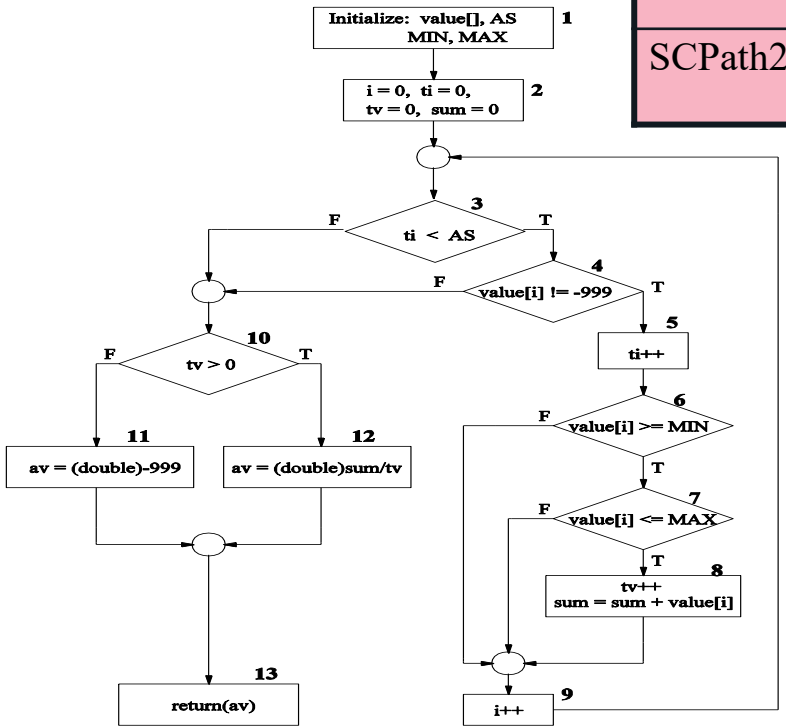
Statement coverage criterion

- Statement coverage means executing individual program statements and observing the output.
- 100% statement coverage means all the statements have been executed at least once.
  - Cover all assignment statements.
  - Cover all conditional statements.
- Less than 100% statement coverage is unacceptable.

SCPath1	1-2-3(F)-10(F)-11-13
SCPath2	1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13

Path Selection Criteria

Statement coverage criterion



SCPath1	1-2-3(F)-10(F)-11-13
SCPath2	1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13

Path Selection Criteria

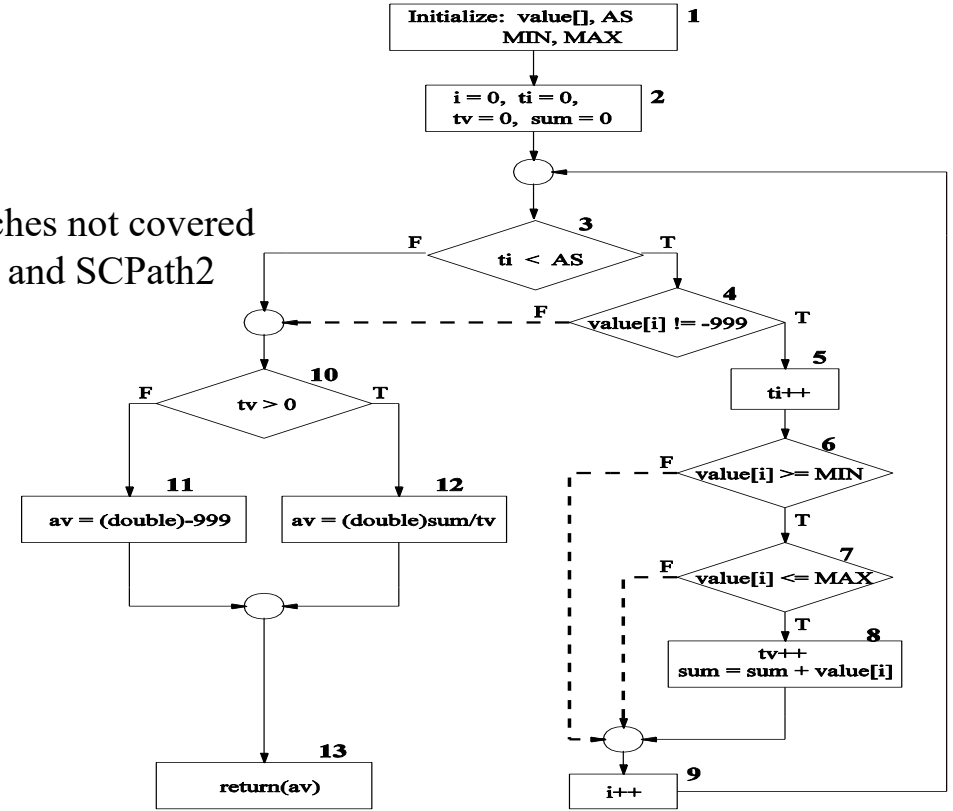
ECU EDITH COWAN UNIVERSITY

Branch coverage criterion

- A branch is an outgoing edge from a node in a CFG.
  - A condition node has two outgoing branches – corresponding to the True and False values of the condition.
- Covering a branch means executing a path that contains the branch.
- 100% branch coverage means selecting a set of paths such that each branch is included on some path.

Path Selection Criteria

The dotted arrows represent the branches not covered by the statement covered by SCPath1 and SCPath2



Path Selection Criteria

Branch coverage criterion

- A branch is an outgoing branch (edge) from a node in a CFG.
  - A condition node has two outgoing branches – corresponding to the True and False values of the condition.
- Covering a branch means executing a path that contains the branch.
- 100% branch coverage means selecting a set of paths such that each branch is included on some path.

SCPath1	1-2-3(F)-10(F)-11-13
SCPath2	1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13

- Branch coverage has the same two paths as Statement coverage, but adds in three more paths to get those branches that were not tested in Statement coverage

BCPath 1	1-2-3(F)-10(F)-11-13
BCPath 2	1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13
BCPath 3	1-2-3(T)-4(F)-10(F)-11-13
BCPath 4	1-2-3(T)-4(T)-5-6(F)-9-3(F)-10(F)-11-13
BCPath 5	1-2-3(T)-4(T)-5-6(T)-7(F)-9-3(F)-10(F)-11-13

Path Selection Criteria

Hot Tip!

In your assignments build the weakest criteria test set first

- Copy and paste to start the next strongest test set
- Repeat, repeat, repeat (build on previous criteria, don’t start again)

Path Selection Criteria

Predicate coverage criterion

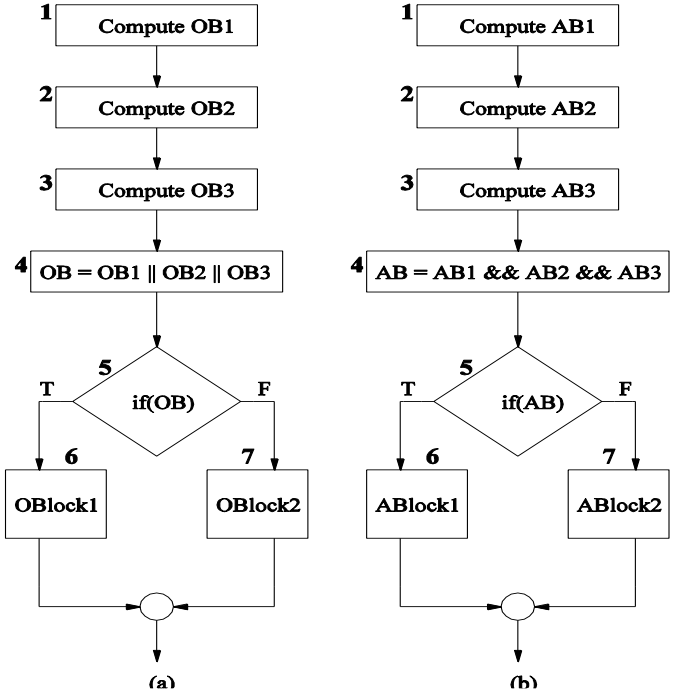
- If all possible combinations of truth values of the conditions affecting a path have been explored under some tests, then we say that predicate coverage has been achieved.

Path Selection Criteria

TABLE 4.6 Two Cases for Complete Statement and Branch Coverage of CFG of Figure 4.9a

Cases	OB1	OB2	OB3	OB
1	T	F	F	T
2	F	F	F	F

Covered! No ☹ Covered!



Partial control flow graph with (a) OR operation and (b) AND operation.

Generating Test Input

ECU
EDITH COWAN UNIVERSITY

Having identified a path, a key question is how to make the path execute, if possible.

- Generate input data that satisfy all the conditions on the path.

Key concepts in generating test input data

- Input vector
- Predicate
- Path condition
- Predicate interpretation
- Path predicate expression
- Generating test input from path predicate expression

Generating Test Input

ECU EDITH COWAN UNIVERSITY

Input vector

- An input vector is a collection of all data entities read by the routine whose values must be fixed prior to entering the routine.
- Members of an input vector can be as follows.
  - Input arguments to the routine
  - Global variables and constants
  - Files
  - Contents of registers (in Assembly language programming)
  - Network connections
  - Timers
- Example: An input vector for openfiles() consists of individual presence or absence of the files “files1,” “file2,” and “file3.”
- Example: The input vector of ReturnAverega() shown in Figure 4.6 is <value[], AS, MIN, MAX>.

Generating Test Input

ECU EDITH COWAN UNIVERSITY

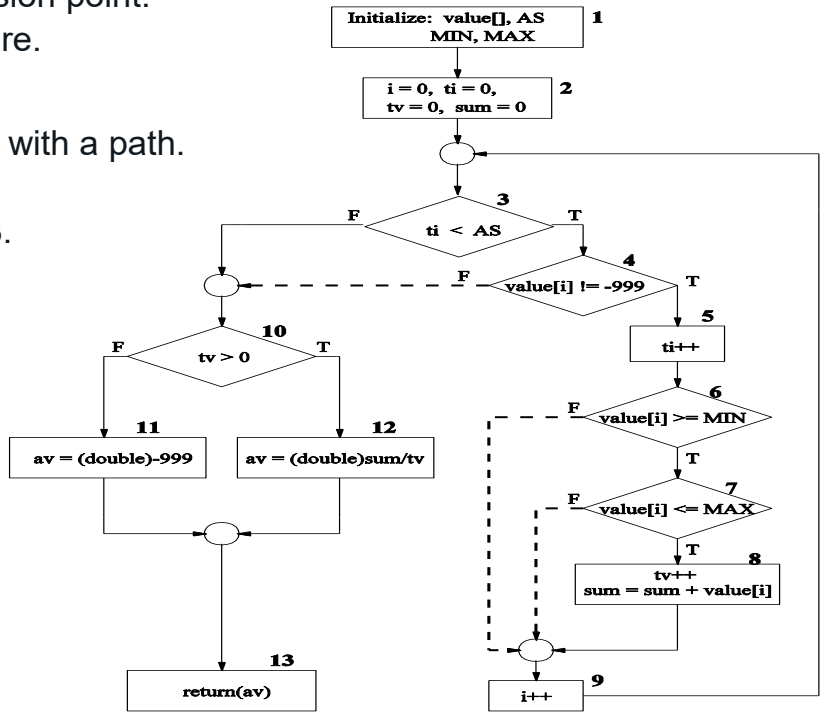
Predicate

- A predicate is a logical function evaluated at a decision point.
- Example: ti < AS is a predicate in node 3 of the figure.

Path predicate

- A path predicate is the set of predicates associated with a path.
- An example path from the CFG:
  - 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13.
- The path predicate for the path shown.

  ti < AS ≡ True
  value[i] != -999 ≡ True
  value[i] >= MIN ≡ True
  value[i] <= MAX ≡ True
  ti < AS ≡ False
  tv > 0 ≡ True

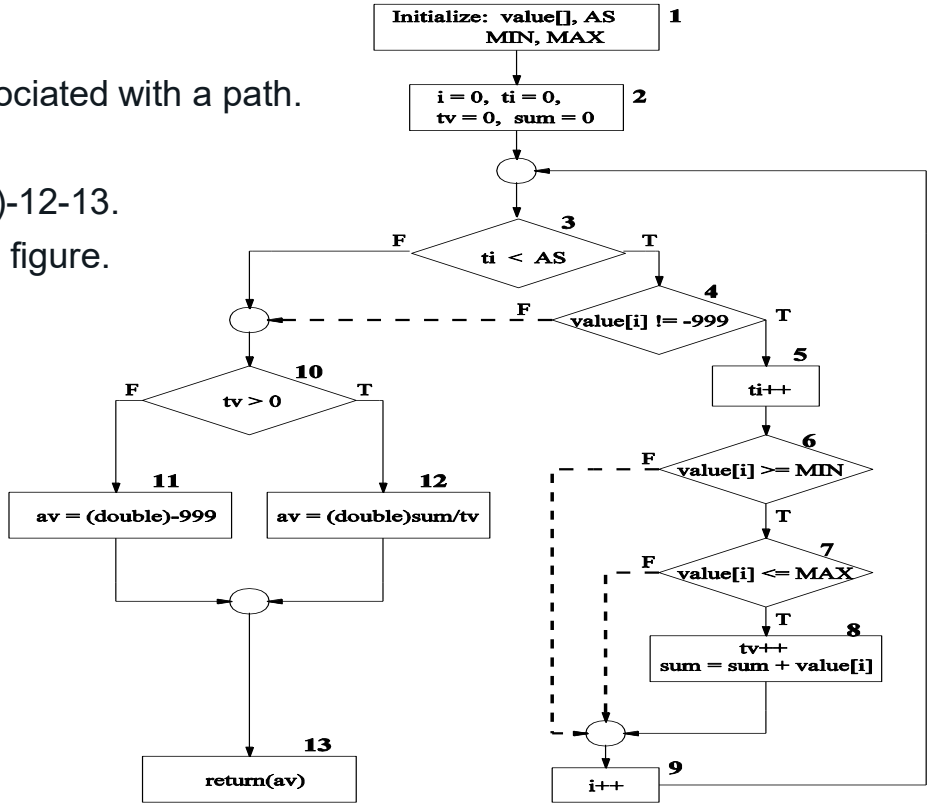


Generating Test Input

Path predicate

- A path predicate is the set of predicates associated with a path.
- **An** example path from the figre:
  - 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13.
- The path predicate for the path shown in the figure.

ti < AS ≡ True
value[i] != -999 ≡ True
value[i] >= MIN ≡ True
value[i] <= MAX ≡ True
ti < AS ≡ False
tv > 0 ≡ True



Generating Test Input

Predicate interpretation

- A path predicate may contain local variables.
- Example: <i, ti, tv> in Figure 4.11 are local variables.
- Local variables play no role in selecting inputs that force a path to execute.
- Local variables can be eliminated by a process called **symbolic execution**.
- Predicate interpretation is defined as the process of
    - symbolically substituting operations along a path in order to express the predicate solely in terms of the input vector and a constant vector.
- The path predicate for the path shown for the figure

ti < AS	≡ True
value[i] != -999	≡ True
value[i] >= MIN	≡ True
value[i] <= MAX	≡ True
ti < AS	≡ False
tv > 0	≡ True

```
int i, ti, tv, sum;
    double av;
    i = 0; ti = 0; tv = 0; sum = 0;
    while (ti < AS && value[i] != -999) {
       ti++;
       if (value[i] >= MIN && value[i] <= MAX) {
          tv++;
          sum = sum + value[i];
       }
       i++;
    }
    if (tv > 0)
       av = (double)sum/tv;
    else
       av = (double) -999;
    return (av);
```

Generating Test Input

Path predicate expression

- An interpreted path predicate is called a path predicate expression.
- A path predicate expression has the following attributes.
  - It is void of local variables.
  - It is a set of constraints in terms of the input vector, and, maybe, constants.
  - Path forcing inputs can be generated by solving the constraints.
  - If a path predicate expression has no solution, the path is infeasible.
- Path predicate expression for the path shown in Figure 4.10.

0 < AS	≡ True	…… (1)
value[0] != -999	≡ True	…… (2)
value[0] >= MIN	≡ True	…… (3)
value[0] <= MAX	≡ True	…… (4)
1 < AS	≡ False	…… (5)
1 > 0	≡ True	…… (6)

Generating Test Input

Node	Node Description	Interpreted Description
1	Input vector:	
	< value[], AS, MIN, MAX >	
2	i = 0, ti = 0,	
	tv = 0, sum = 0	
3(T)	ti < AS	0 < AS
4(T)	value[i]! = − 999	value[0]! = − 999
5	ti++	ti = 0 + 1 = 1
6(T)	value[i] > = MIN	value[0] > = MIN
7(T)	value[i] < = MAX	value[0] < = MAX
8	tv++	tv = 0 + 1 = 1
	sum = sum + value[i]	sum = 0 + value[0]
		= value[0]
9	i++	i = 0 + 1 = 1
3(F)	ti < AS	1 < AS
10(T)	tv > 0	1 > 0
12	av = (double) sum/tv	av = (double) value[0]/1
13	return(av)	return(value[0])

Note: The bold entries in column 1 denote interpreted predicates.

Generating Test Input

Path predicate expression

- 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13.

- The path predicate for the path shown.

ti < AS	≡ True
value[i] != -999	≡ True
value[i] >= MIN	≡ True
value[i] <= MAX	≡ True
ti < AS	≡ False
tv > 0	≡ True

- Path predicate expression for the path shown.

0 < AS	≡ True	…… (1)
value[0] != -999	≡ True	…… (2)
value[0] >= MIN	≡ True	…… (3)
value[0] <= MAX	≡ True	…… (4)
1 < AS	≡ False	…… (5)
1 > 0	≡ True	…… (6)

Generating Test Input

EDITH COWAN UNIVERSITY

Generating input data from a path predicate expression

- Consider the path predicate expression of Figure 4.13 (reproduced below.)

0 < AS	≡ True	…… (1)
value[0] != -999	≡ True	…… (2)
value[0] >= MIN	≡ True	…… (3)
value[0] <= MAX	≡ True	…… (4)
1 < AS	≡ False	…… (5)
1 > 0	≡ True	…… (6)

- One can solve the above equations to obtain the following test input data

AS	= 1
MIN	= 25
MAX	= 35
Value[0]	= 30

Path 1-2-3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13.
Input =

Summary

Control flow is a fundamental concept in program execution.

A program path is an instance of execution of a program unit.

Select a set of paths by considering path **selection criteria**.

- Statement coverage
- Branch coverage
- Predicate coverage
- All paths

From source code, derive a CFG (compilers are modified for this.)

Select paths from a CFG based on path selection criteria.

Extract path predicates from each path.

Solve the path predicate expression to generate test input data.

There are two kinds of paths.

- feasible
- infeasible

Path Testing

Paths derived from some graph construct.

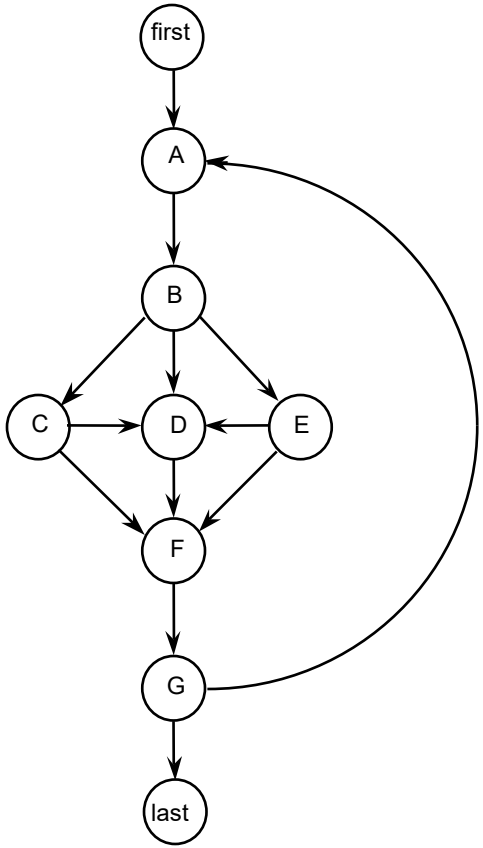
When a test case executes, it traverses a path.

Limitations:

- Huge number of paths implies simplifications are required.
- Can also have many infeasible paths.

By itself, path testing can result in a false sense of security about the testing adequacy.

Test Cases for Schach's "Program"

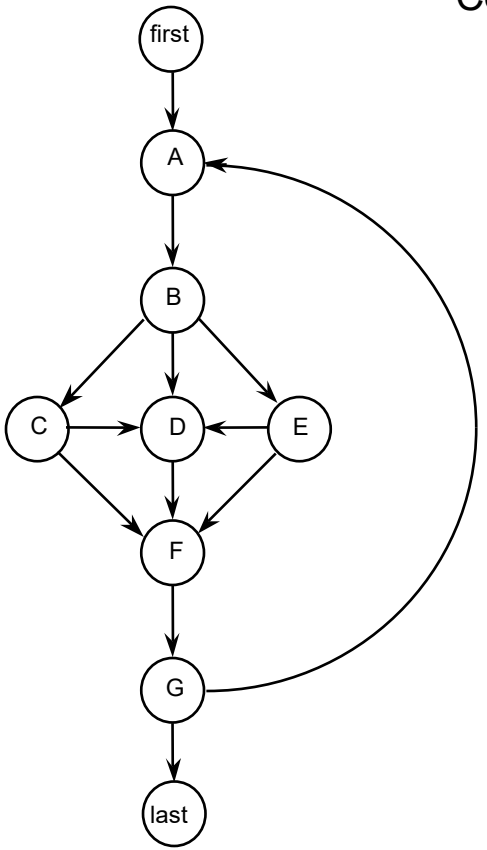


1. First-A-B-C-F-G-Last
2. First-A-B-C-D-F-G-Last
3. First-A-B-D-F-G-A-B-D-F-G-Last
4. First-A-B-E-F-G-Last
5. First-A-B-E-D-F-G-Last

These test cases cover

- Every node
- Every edge
- Normal repeat of the loop
- Exiting the loop

Common Objection to Path-Based Testing
(Trillions of Paths)



If the loop executes up to 18 times, there are 4.77 Trillion paths. Impossible, or at least, infeasible, to test them all. [Schach]

$5^0 + 5^1 + 5^2 + \ldots + 5^{18} = 4{,}768{,}371{,}582{,}030$

Stephen R. Schach, *Software Engineering,* , (2nd edition) Richard D. Irwin, Inc. and Aksen Associates, Inc. 1993

Basis Path Based

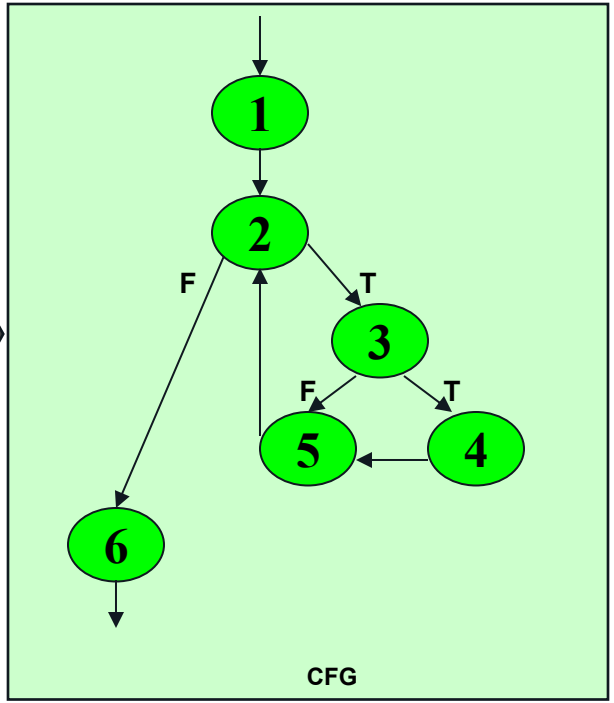
A generalized technique to find out the number of paths needed (known as *cyclomatic complexity*) to cover all arcs and nodes in CFG.

Steps:

1. Draw the CFG for the code fragment.
2. Compute the *cyclomatic complexity number* ***C***, for the CFG.
3. Find at most ***C*** paths that cover the nodes and arcs in a CFG, also known as **Basis Paths Set;**
4. Design test cases to force execution along paths in the **Basis Paths Set.**

Path Based Testing: Step 1

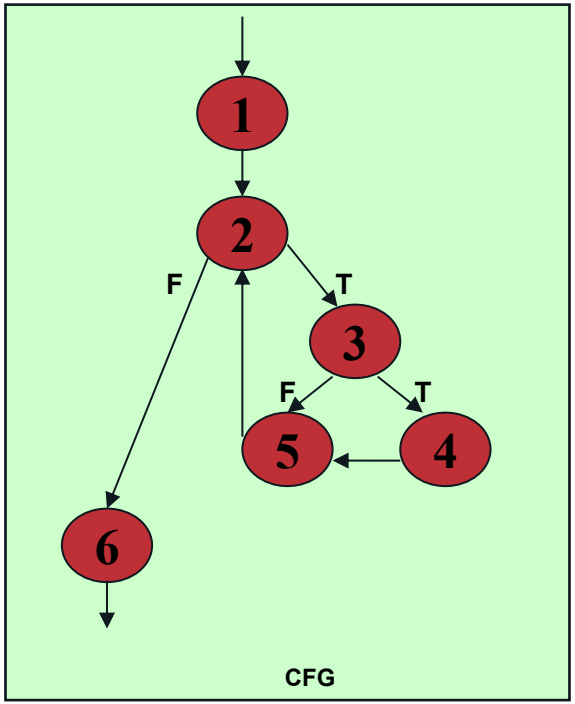
```
findSmallest(int X[], N)
{
       smallest = X[0];
       Y = 1;

     while (Y < N) {
         if (X[Y] < smallest)
             smallest = X[Y];
         Y = Y + 1;
     }
     print smallest
 }
```



Path Base Testing: Step 2

Cyclomatic complexity =

- The number of ‘regions’ in the graph; OR
- The number of predicates + 1.



SLIDES From LAMAR UNIVERSITY

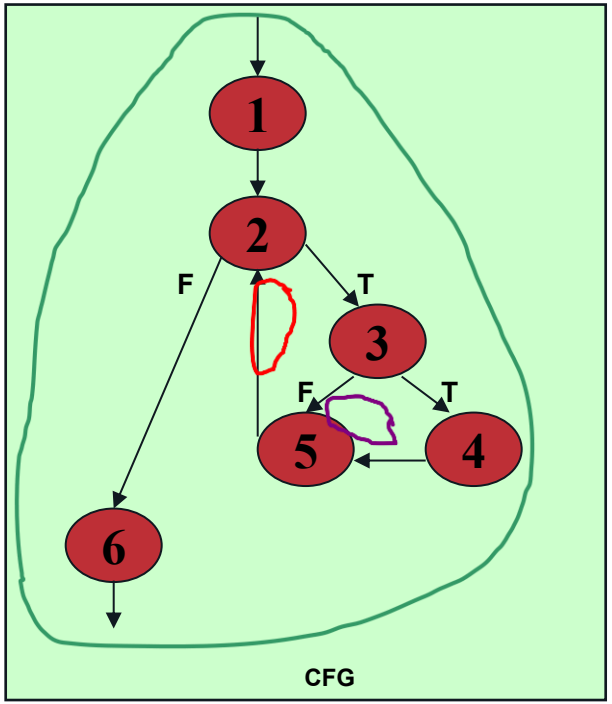
Path Base Testing: Step 2

Region: Enclosed area in the CFG.

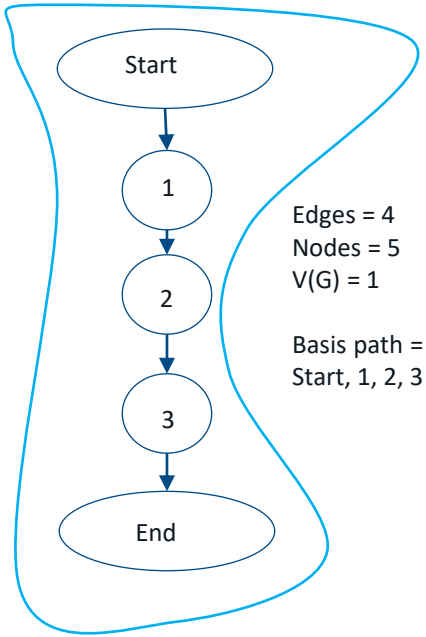
- Do not forget the outermost region.

In this example:

- 3 Regions (see the circles with different colors).
- Cyclomatic Complexity = 3

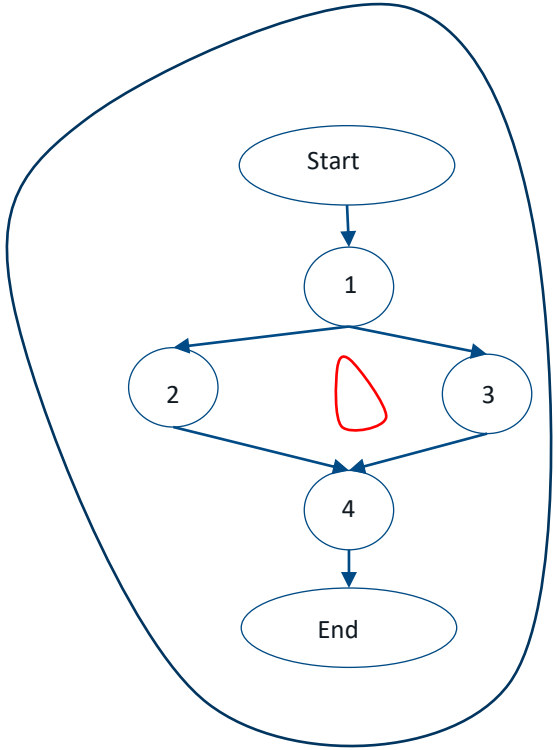


Path Base Testing: Step 2



Edges = 4
Nodes = 5
V(G) = 1

Basis path =
Start, 1, 2, 3 , End



Edges = 6
Nodes = 6
V(G) = 2

Basis path =
Start, 1, 2 ,4, end
Start, 1, 3 ,4, end

Path Base Testing: Step 2

Predicates:

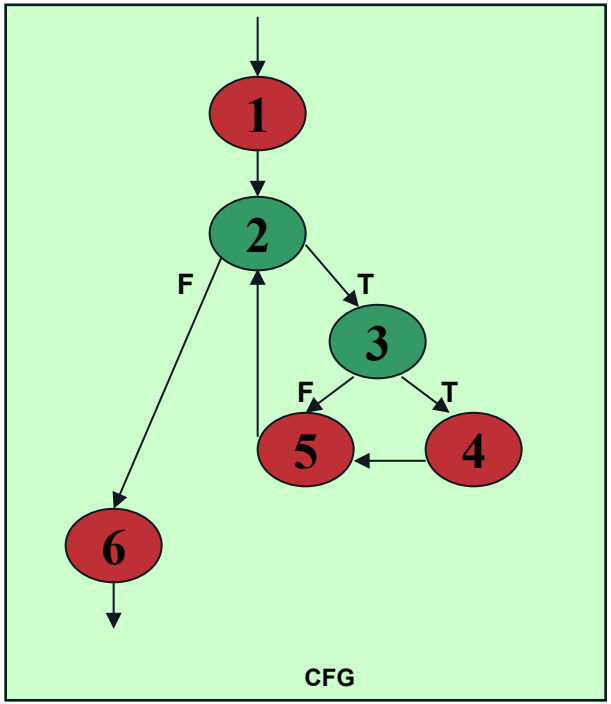
- Nodes with multiple exit arcs.
- Corresponds to branch/conditional statement in program.

In this example:

- Predicates = 2
  - (Node 2 and 3)
- Cyclomatic Complexity

  = 2 + 1

  = 3



Path Base Testing: Step 3

Independent path:

- An **executable** or **realizable path** through the graph from the start node to the end node that has not been traversed before.
- **Must** move along **at least one arc** that has not been yet traversed (an unvisited arc).
- The objective is to cover all statements in a program by independent paths.

The number of independent paths to discover <= cyclomatic complexity number.

Decide the Basis Path Set:

- It is the maximal set of *independent paths* in the flow graph.
- **NOT** a unique set.

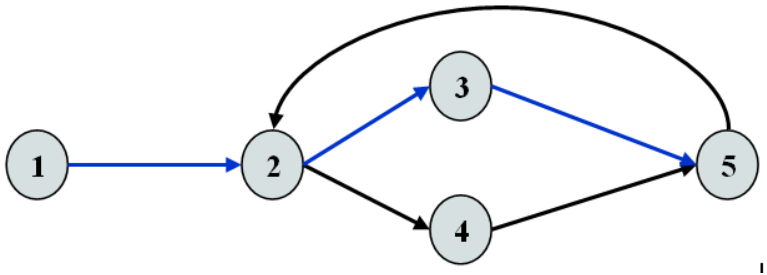
Example

1-2-3-5 can be the first independent path; 1-2-4-5 is another; 1-2-3-5-2-4-5 is one more.

There are only these 3 independent paths. The basis path set is then having 3 paths.

Alternatively, if we had identified 1-2-3-5-2-4-5 as the first independent path, there would be no more independent paths.

The number of independent paths therefore can vary according to the order we identify them.



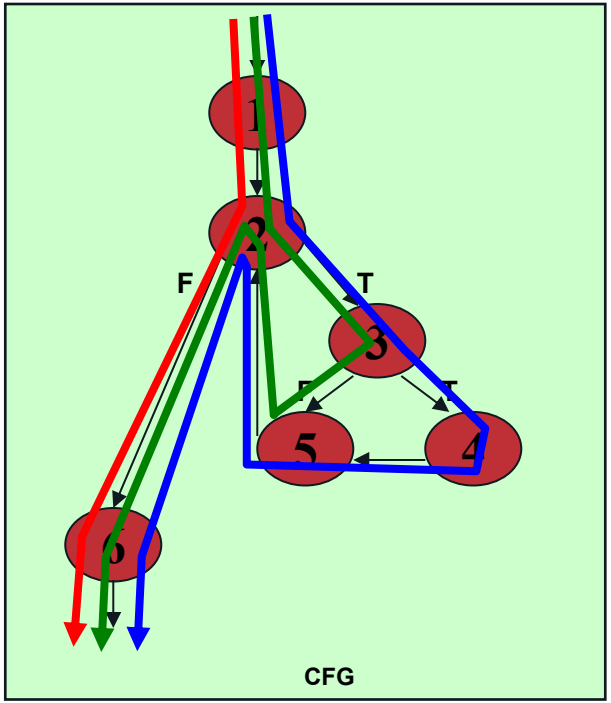
Path Base Testing: Step 3

Cyclomatic complexity = 3.

Need at most **3** independent paths to cover the CFG.

In this example:

- [1 – 2 – 6]
- [1 – 2 – 3 – 5 – 2 – 6]
- [1 – 2 – 3 – 4 – 5 – 2 – 6]



Path Base Testing: Step 4

Prepare a test case for each independent path.

In this example:

- Path: [1 – 2 – 6]
  - Test Case: X = { 5, …}, N = 1
  - Expected Output: 5
- Path: [1 – 2 – 3 – 5 – 2 – 6]
  - Test Case: X = { 5, 9, … }, N = 2
  - Expected Output: 5
- Path: [1 – 2 – 3 – 4 – 5 – 2 – 6]
  - Test Case: X = { 8, 6, … }, N = 2
  - Expected Output: 6

```
findSmallest(int X[], N)
{
       smallest = X[0];
       Y = 1;

     while (Y < N) {
         if (X[Y] < smallest)
             smallest = X[Y];
         Y = Y + 1;
     }
     print smallest
 }
```

These tests will result a complete predicate and statement coverage of the code.