4.4 Paths in a control flow graph

For the convenience of discussion, we assume that a control flow graph has exactly one *entry* node and exactly one *exit* node. Each node is tagged with a unique integer value. Also, the two branches of a decision node are labelled with *True*(T) or *False*(F) depending on the control flow. We are interested in identifying entry-exit paths in a control flow graph.

A path is represented as a sequence of computation and decision nodes from the entry node to the exit node. While including a decision node in a path, we also specify whether control exits the node via its *True* or *False* branch.

In Table 4.1, we show a few paths from the control flow graph of Figure 4.7. The reader may note that we have arbitrarily chosen these paths without applying any path selection criterion. In **Path 3**, we have unfolded the loop just once, whereas **Path 4** unfolds the same loop twice, and these are two distinct paths.

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(T)-7(T)-8-9-** **3(T)-4(T)-5-6(T)-7(T)-8-9-3(F)-10(T)-12-13**

Table 4.1: Examples of path in the control flow graph of Figure 4.7.

4.5 Path Selection Criteria

The objectives of defining path selection criteria are as follows.

- All program constructs are exercised at least once. The programmer needs to observe the outcome of executing each and every program construct, such as statements, Boolean conditions, and returns, to name a few of them.
- We do not generate test inputs which execute the same path again and again. Executing the same path several times is a waste of resources. However, if each execution of a program path potentially updates the *state* of the system, e.g. the database state, then multiple executions of the same path may not be identical.
- We know what program features we have tested and what we have not tested. For example, we may execute an `if` statement only

once so that it evaluates to *True*. If we do not execute it once again for its *False* evaluation, we are at least aware that we have not observed the program's outcome with a *False* evaluation of the `if`.

Now we explain the following well-known 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

4.5.1 All-path coverage criterion (P_∞)

If *all* paths are selected, then one can detect all faults, except those due to missing path errors. However, a program may contain a large number of paths, or even an infinite number of paths. The small, loop-free `openfiles()` function shown in Figure 4.3 contains more than 25 paths. Though only eight of all those paths are feasible, one does not know whether or not a path is feasible at the time of selecting those paths. If one selects all possible paths in a program, then we say that the *all-path* selection crietrion has been satisfied. We denote the *all-path* selection criterion by P_∞.

Now we consider the example of `openfiles()` function. This function tries to open the three files "file1", "file2", and "file3." The function returns an integer representing the number of files it has successfully opened. A file is said to be successfully opened with "read" access if the file exists. Therefore, there are three conceptual inputs variables to the function, namely the existence of those three files. The existence of a file is either "Yes" or "No". Thus, the input domain of the function consists of eight combinations of the existence of the three files as shown in Table 4.2.

For each input, that is each row of Table 4.2, of `openfiles()`, we can trace a path in the control flow graph of Figure 4.5. Ideally, we identify test inputs to execute a certain path in a program–and this will be explained in a subsequent part of this chapter. We give three examples of what test input executes what path in Table 4.5.1. In this manner, we can identify eight possible paths in Figure 4.5. Though the all-paths selection criterion is desirable from the viewpoint of detecting faults, it remains as an ideal criterion and may not be achievable in practice.

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

Table 4.2: The input domain of *openfiles()*.

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**
$< Yes, YesYes >$	**1-2-3(T)-4(F)-6-8-9(T)-10(T)-11-13(F)-14-15(T)-16(T)-18-20-21**

Table 4.3: Inputs and paths in *openfiles()*.

4.5.2 Statement coverage criterion (C1)

Statement coverage refers to executing individual program statements and observing its outcome. If *all* statements have been executed at least once, we say that 100% statement coverage has been achieved. Complete statement coverage is denoted by **C1**. Complete statement coverage is the *weakest* coverage criterion in program testing. Any test suite that achieves less than **C1** for new software is not acceptable. This is because the programmer must execute the program under various test scenarios to observe the outcome of executing each and every statement in the program.

All program statements are represented in some form in a control flow graph. Referring to the `ReturnAverage()` function and its control flow graph in Figures 4.6 and 4.7, respectively, the four assignment statements

```
i   = 0;
ti  = 0;
```

```
tv  = 0;
sum = 0;
```

have been represented by node **2**. The `while` statement has been represented as a loop, where the condition for loop control

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

has been represented by nodes **3** and **4**. Thus, covering a statement in a program means visiting one or more nodes representing the statement. To be precise, covering a statement means selecting a **feasible** entry-exit path that includes those nodes. Since a single entry-exit path includes many nodes, we need to select just a few paths to cover all the nodes of a control flow graph.

Therefore, the basic problem is to select a few feasible paths to cover all the nodes of a control flow graph in order to achieve the complete statement coverage criterion **C1**. While selecting paths, we follow these rules:

- Select short paths.
- Select paths of increasingly longer length. Unfold a loop several times, if there is a need.
- Select arbitrarily long, "complex" paths.

One can select the two paths shown in Figure 4.4 to achieve complete statement coverage.

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

Table 4.4: Paths for statement coverage of the flow graph of Figure 4.7.

4.5.3 Branch coverage criterion (C2)

Syntactically, a branch is an outgoing edge from a node. All the rectangle nodes have at most one outgoing branch (edge)–the exit node of a control flow graph does not have an outgoing branch. All the diamond nodes have two outgoing branches or edges. Covering a branch means selecting a path that includes the branch. Complete branch coverage, denoted by **C2**, means selecting a number of paths such that every branch is

included in at least one path. Statement coverage guarantees that all the outgoing branches from the rectangle nodes are covered, but there is no guarantee that all the outgoing branches from the diamond nodes will be covered.

In a preceeding discussion, we showed that one can select two paths, **SCPath 1** and **SCPath 2** in Table 4.4, to achieve complete statement coverage. These two paths cover *all* the nodes (statements) and most of the branches of the control flow graph shown in Figure 4.7. The branches which are not covered by these two paths have been highlighted by bold dotted lines in Figure 4.8. These uncovered branches correspond to the three independent conditions

```
value[i] != -999,
value[i] >= MIN,
value[i] <= MAX
```

evaluating to *False*. The meaning of these uncovered branches is that as a programmer we have not observed the outcome of program execution as a result of the conditions evaluating to *False*. Thus, complete branch coverage means selecting enough number of paths such that every condition evaluates to *True* at least once and to *False*. at least once.

To cover the branches highlighted by the bold dotted lines in Figure 4.8, we need to select more paths. A set of paths for complete branch coverage is shown in Table 4.5.

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**

Table 4.5: Paths for branch coverage of the flow graph of Figure 4.7.

4.5.4 Predicate coverage criterion (C3)

To explain the concept of predicate coverage, first we refer to the partial control flow graph of Figure 4.9(a). *OB1*, *OB2*, *OB3*, and *OB* are four Boolean variables. First, the program computes the values of the individual variables *OB1*, *OB2* and *OB3*–details of their computation are irrelevant to our discussion and have been omitted. Next, *OB* is computed as shown in the control flow graph. The control flow graph checks

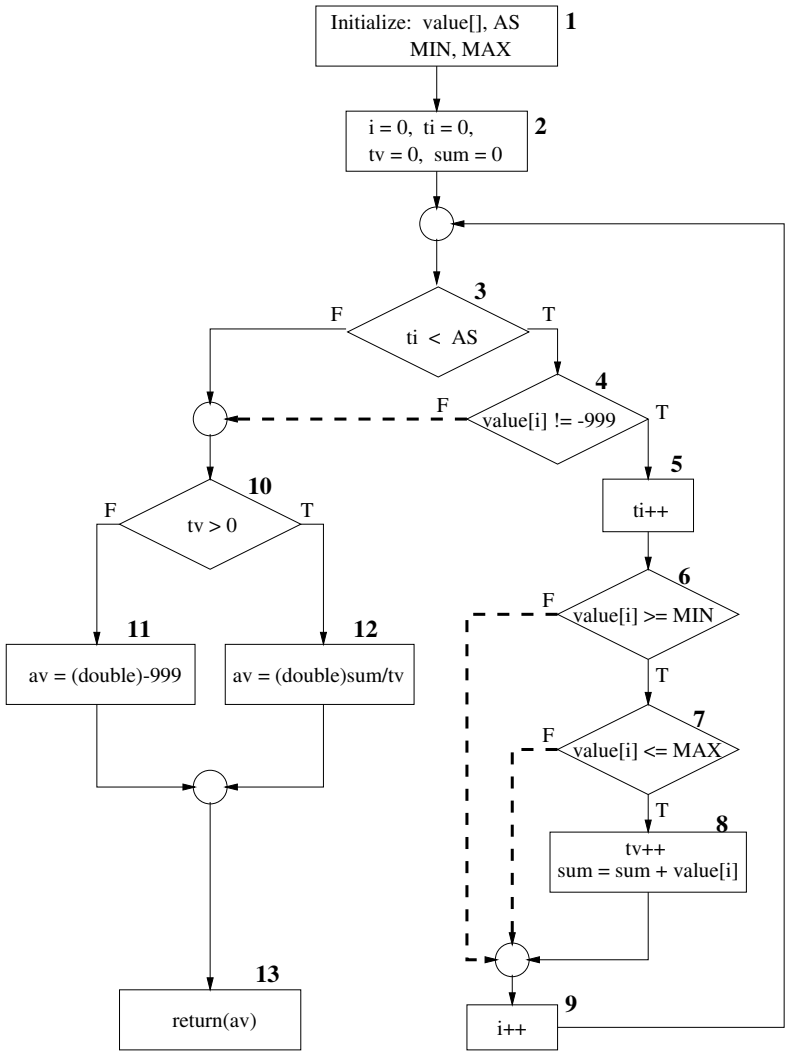

Figure 4.8: Branches not covered by statement coverage in *ReturnAverage()*.

the value of *OB* and executes either *OBlock1* or *OBlock2* depending on whether *OB* evaluates to *True* or *False*, respectively.

We need to design just two test cases to achieve both statement coverage and branch coverage. To achieve this, we select inputs such that the four Boolean conditions in Figure 4.9(a) evaluate to values shown in Table 4.6. The reader may note that we have shown just one way of forcing *OB* to *True*. As a programmer if we select inputs so that these two cases hold, then we are not observing the effect of the computations taking place in nodes **2** and **3**. There may be faults in the computation parts of nodes **2** and **3** such that *OB2* and *OB3* always evaluate to *False*.

Therefore, there is a need to design test cases such that a path is executed under all possible conditions. Referring to Figure 4.9(a), the *False* branch of node **5** is executed under exactly one condition, namely when *OB1* = *False*, *OB2* = *False*, and *OB3* = *False*, whereas the *True* branch executes under *seven* conditions.

If all possible combinations of truth values of the conditions affecting a selected path have been explored under some tests, then we say that *predicate coverage* has been achieved. Therefore, the path taking the *True* branch of node **5** in Figure 4.9(a) must be executed for all *seven* possible combinations of truth values of *OB1*, *OB2*, and *OB3* which result in *OB* = *True*.

A similar situation holds for the partial control flow graph shown in Figure 4.9(b), where *AB1*, *AB2*, *AB3*, and *AB* are Boolean variables.

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

Table 4.6: Two cases for complete statement and branch coverage of the control flow graph of Figure 4.9(a).

4.6 Generating test input

In Section 4.5, we explained the concept of path selection criteria to cover certain aspects of a program with a set of paths. The program aspects we considered were all statements, true and false evaluations of each condition, and combinations of conditions affecting execution of a path. Now, having identified a path, the question is how to select input

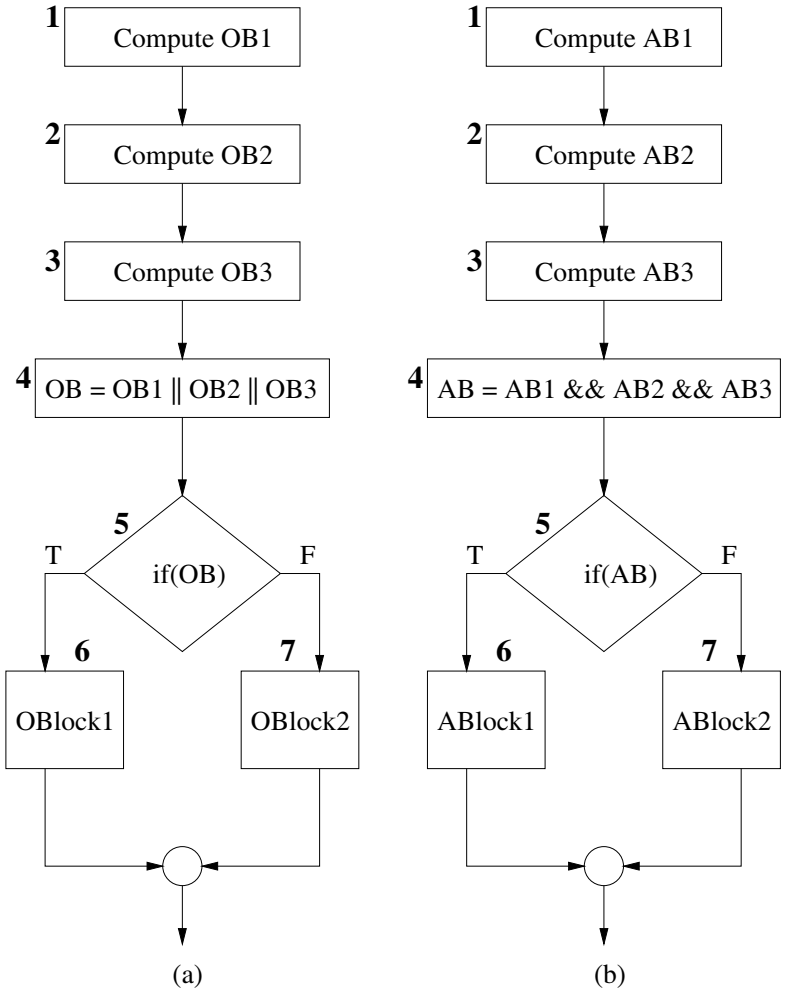

Figure 4.9: Partial control flow graph with OR operation (a) and AND operation (b).

values such that when the program is executed with the selected inputs, we essentially force the program to execute the selected path. In other words, we need to identify path forcing inputs. In the following, we define a few terms and give an example of generating test inputs for a selected path.

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 of a routine can take different forms as listed below.

- Input arguments to a routine
- Global variables and constants
- Files
- Contents of registers in Assembly language programming
- Network connections
- Timers

A file is a complex input element. In one case, mere existence of a file can be considered as an input, whereas in another case, contents of the file are considered to be inputs. Thus, the idea of an input vector is more general than the concept of input arguments of a function.

Example 1: Referring to `openfiles()` function of Figure 4.3, an input vector consists of individual presence or absence of the files "file1", "file2", and "file3".

Example 2: The input vector of `ReturnAverage()` method shown in Figure 4.6 is < `value[], AS, MIN, MAX` >.

Predicate: A predicate is a logical function evaluated at a decision point.

Example 1: Referring to the decision node **3** of Figure 4.7, the predicate is `ti < AS`.

Example 2: Referring to the decision node **5** of Figure 4.9, the predicate is `OB`.

Path predicate: A path predicate is the set of predicates associated with a path.

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

Figure 4.10: An example of a path from 4.7.

Example: Referring to Figure 4.7, consider the following path:

The above path clearly indicates that nodes **3**, **4**, **6**, **7**, and **10** are decision nodes. The predicate associated with node **3** appears twice in the path; in the first instance it evaluates to *True* and in the second instance it evaluates to *False*. The path predicate associated with the path under consideration is shown in Figure 4.11.

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

Figure 4.11: Path predicate for the path shown in 4.10.

While identifying a path predicate, we also specify the indended evaluation of the component predicates as found in the path specification. For instance, in the above path predicate we specify that `value[i] != -999` must evaluate to *True*. We keep this additional information for the following two reasons.

- In the absence of this additional information denoting the intended evaluation of a predicate, we will have no way to distinguish between the two instances of the predicate `ti < AS`, namely **3(T)** and **3(F)**, associated with node **3**.
- In order to generate path forcing inputs, we must know whether the individual component predicates of a path predicate evaluate to *True* or *False*.

Predicate interpretation: The path predicate shown in Figure 4.11 is composed of elements of the input vector < `value[]`, `AS`, `MIN`, `MAX` >, a vector of local variables < `i`, `ti`, `tv` >, and the constant `-999`.

```
public static int SymSub(int x1, int x2){
int y;
y = x2 + 7;
if (x1 + y >= 0)
     return (x2 + y);
else return (x2 - y);
}
```

Figure 4.12: A Java method to explain symbolic substitution.

Local variables are not visible outside a function. Rather, local variables are used to:

- hold intermediate results,
- point to array elements, and
- control loop iterations.

In other words, they play no roles in selecting path forcing inputs. Therefore, we can easily substitute all the local variables in a predicate with the elements of the input vector by using the idea of symbolic substitution. Let us consider the method shown in Figure 4.12. The input vector for the method in Figure 4.12 given by $< \texttt{x1}, \texttt{x2} >$. The method defines a local variable `y` and also uses the constants `7` and `0`.

The predicate

```
x1 + y >= 0
```

can be rewritten as

```
x1 + x2 + 7 >= 0
```

by symbolically substituting `y` with `x2 + 7`. The rewritten predicate

```
x1 + x2 + 7 >= 0
```

has been expressed solely in terms of the input vector $< \texttt{x1}, \texttt{x2} >$ and the constant vector $< 0, 7 >$.

Thus, *predicate interpretation* is defined as the process of symbolically substituting operations along a path in order to express the predicates solely in terms of the input vector and a constant vector.

In a control flow graph, there may be several different paths leading up to a decision point from the initial node, with each path doing different computations. Therefore, a predicate may have different interpretations depending on how control reaches the predicate under consideration.

Path Predicate Expression: An interpreted path predicate is called a path predicate expression. A path predicate expression has the following properties.

- A path predicate expression is void of local varibles. and is solely composed of elements of the input vector and possibly a vector of constants.
- A path predicate expression is a set of constraints constructed from the elements of the input vector and possibly a vector of constants.
- By solving the set of constraints in the path predicate expression of a path, one can produce input values which when applied to the routine will cause the selected path to be executed.
- If the set of constraints cannot be solved, there exist no input which can cause the selected path to execute. In other words, the selected path is said to be *infeasible*.
- An infeasible path does not imply that one or more components of a path predicate expression are unsatisfiable. It simply means that the total combination of all the components in a path predicate expression is unsatisfiable.
- Infeasibility of a path predicate expression suggests that one considers other paths in an effort to meet a chosen path selection criterion.

Example 1: Consider the path shown in Figure 4.10 from the control flow graph of Figure 4.7. In Table 4.7, we have shown the nodes of the path in column 1, the corresponding description of each node in column 2, and the interpretation of each node in column 3. For the purpose of convenience, we have highlighted each interpreted predicate with a box. The intended evaluation of

each interpreted predicate can be found in column 1 of the same row.

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++ sum = sum + value[i]	tv = 0 + 1 = 1 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])

Table 4.7: Interpretation of path predicate of path shown in Fig. 4.10.

For the sake of clarity, we show the path predicate expression of the path under consideration in Figure 4.13. The rows of Figure 4.13 have been obtained from Table 4.11 by combining each interpreted predicate in column 3 with its intended evaluation found in column 1. Now the reader may compare Figure 4.11 and 4.13 to note that the predicates in Figure 4.13 are interpretations of the

corresponding predicates in Figure 4.11.

```
          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)
```

Figure 4.13: Path predicate expression for the path shown in 4.10.

Example 2: Now we give an example of an infeasible path. In Figure 4.14 we show a path appearing in the control flow graph of Figure 4.7. The path predicate and its interpretation are shown in Table 4.8, and the path predicate expression has been shown in Figure 4.15. The path predicate expression is unsolvable because the constraint `0 > 0` ≡ *True* is unsatisfiable. Therefore, the path shown in Figure 4.14 is an infeasible path.

1-2-3(T)-4(F)-10(T)-12-13.

Figure 4.14: Another example of path from 4.7.

```
          0 < AS   ≡ True   ..... (1)
value[0] != -999   ≡ True   ..... (2)
           0 > 0   ≡ True   ..... (3)
```

Figure 4.15: Path predicate expression for the path shown in Figure 4.14.

Generating Input Data from a Path Predicate Expression: In order to generate input data which can force program to execute a selected path, we must solve the corresponding path predicate expression. Let us consider the path predicate expression shown in Figure 4.13. We observe that constraint (1) is always satisfied. Constraints (1) and (5) must be solved together to obtain `AS = 1`. Similarly, constraints (2), (3), and (4) must be solved together. We note that `MIN <= value[0] <= MAX` and `value[0] != -999`.

Therefore, we have many choices to select values of `MIN`, `MAX`, and `value[0]`. An instance of the solutions of the constraints of Figure 4.13 is shown in Figure 4.16.

```
      AS  = 1
     MIN  = 25
     MAX  = 35
value[0]  = 30
```

Figure 4.16: Input data satisfying the constraints of Figure 4.13.

4.7 Examples of test data selection

In this section, we give examples of selected test data to achieve complete statement and branch coverage. In Table 4.9, we show four sets of test data. The first two data sets cover all statements of the control flow graph in Figure 4.7. However, for complete branch coverage, we need all four sets of test data.

If we execute the method `ReturnAverage` shown in Figure 4.6 with the four sets of test input data shown in Figure 4.9, then we notice that each and every statement of the method is executed at least once, and every Boolean condition once evaluates to *True* and once evaluates to *False*. In the sense of complete branch coverage, we have thoroughly tested the method. However, we can easily introduce very simple faults in the method which go undetected when we execute the method with the above four sets of test data. Two examples of fault insertion are given below.

Example 1: Suppose that we replace the correct statement

```
av = (double) sum/tv;
```

with a faulty statement

```
av = (double) sum/ti;
```

in the method. Here the fault is that the method computes the average of the total number of inputs, denoted by `ti`, rather than the total number of valid inputs, denoted by `tv`.

Example 2: Suppose that we replace the correct statement

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(F)	value[i] != -999	value[0] != -999
10(T)	tv > 0	0 > 0
12	av = (double) sum/tv	av = (double) value[0]/0
13	return(av)	return((double) value[0]/0)

Table 4.8: Interpretation of path predicate of path shown in Fig. 4.14.

Test data set	Input Vector			
	AS	MIN	MAX	value[]
1	1	5	20	[10]
2	1	5	20	[-999]
3	1	5	20	[4]
4	1	5	20	[25]

Table 4.9: Test data for statement and branch coverage.

```
sum = sum + value[i];
```

with a faulty statement

```
sum = value[i];
```

in the method. Here the fault is that the method no more computes the sum of all the valid inputs in the array. In spite of the fault, the first set of test data produce the correct result due to *coincidental correctness*.

The above two examples of faults lead us to the following conclusions.

- One must generate test data to satisfy certain selection criteria, because those selection criteria identify what aspects of a program we want to cover.
- After the coverage criteria have been met, additional tests, which are much longer than the simple tests generated to meet coverage criteria, must be generated.
- Given a set of test data for a program, we can inject faults into the program which go undetected by those test cases.

4.8 Exercises

Question 1 You are given the following binary search routine in C. The input array `v[]` is assumed to be sorted in ascending order, `n` is the array size, and you want to find the index of an element `x` in the array. If `x` is not found in the array, the routine is supposed to return `-1`. Answer the questions following the routine.

```
int binsearch(int x, int v[], int n){
    int low, high, mid;
    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low + high)/2;
        if (x < v[mid])
               high = mid - 1;
        else if (x > v[mid])
               low = mid + 1;
        else
               return mid;
    }
    return -1;
}
```

Figure 4.17: Binary search routine.

(i) Draw a control flow graph for `binsearch()`.

(ii) From the control flow graph, identify a set of entry/exit paths to satisfy the complete statement coverage criterion.

(iii) Identify additional paths, if necessary, to satisfy the complete branch coverage criterion.

(iv) For each path identified above, derive their path predicate expressions.

(v) Solve the path predicate expressions to generate test input, and compute the corresponding expected outcomes.

(vi) Are all the selected paths feasible? If not, select and show that a path is infeasible, if it exists.

(vii) Can you introduce a fault in the routine so that it goes undetected by your test cases designed for complete branch coverage?

Question 2 The specification of an *insert* routine is as follows. The routine accepts four parameters: an integer array (`a`), the length of the array (`len`), the number of valid elements in the array (`n`)- the first `n` elements of the array are considered to be valid, and an integer (`x`) to be potentially inserted into the array. If `x` is a member of the array, the routine returns `n`. If `x` is not a member of `a`, and there is space in the array, the routine inserts `x` at the end of the valid part of the array and returns `n+1`. If `x` is not a member of the array and there is no space left in the array, then the routine returns `-1`. If `n < len`, there is space in the array for more elements to be inserted. An implementation of the *insert* routine is given below. The caller ensures that `len > 0`.

```
static int insert(int a[], int len, int  n, int x){
     int i; int found = 0;
     for (i = 0; i <= n-1; i++){
         if (a[i] == x){ found = 1; break; }
     }
     if (found) return n;
     else if (n < len) { a[n] = x; return n+1; }
     else return -1;
}
```

Figure 4.18: An insert routine.

(i) Draw a detailed control flow graph for the *insert()* routine.

(ii) Introduce a fault in the routine by making very small changes so that the following test cases fail to detect the fault. Explain why the fault went undetected.

Test case #1

Input: a = [0, 3, 5, 7, -1], len = 5, n = 4, x = 4
Expected returned value = 5
State of the array = [0, 3, 5, 7, 4]

Test case #2

Input: a = [0, 3, 5, 7, -1], len = 5, n = 4, x = 5
Expected returned value = 4
State of the array = [0, 3, 5, 7, -1]

Test case #3

Input: a = [0, 3, 5, 7], len = 4, n = 4, x = 6
Expected returned value = -1
State of the array = [0, 3, 5, 7]

Question 3 Answer the following questions.

(i) Compare the power of test selection methods based on statement coverage, branch coverage and predicate coverage.

(ii) Argue that the short test cases designed to achieve statement coverage and branch coverage have weak fault detection ability.

(iii) If the statement coverage and branch coverage critera produce weak test cases, why should a programmer design test cases to achieve those coverage criteria?