17.4.3 Deriving Test Cases

The basis path testing method can be applied to a procedural design or to source code. In this section, we present basis path testing as a series of steps. The procedure *average,* depicted in PDL in Figure 17.4, will be used as an example to illustrate each step in the test case design method. Note that *average,* although an extremely simple algorithm, contains compound conditions and loops. The following steps can be applied to derive the basis set:

**1.Using the design or code as a foundation, draw a corresponding flow graph.**

A flow graph is created using the symbols and construction rules presented in Section 16.4.1. Referring to the PDL for average in Figure 17.4, aflow graph is created by numbering those PDL statements that will bemapped into corresponding flow graph nodes. The corresponding flow graphis in Figure 17.5.

**2. Determine the cyclomatic complexity of the resultant flow graph.**

The cyclomatic complexity, *V*(*G*), is determined by applying the algorithms described in Section 17.5.2. It should be noted that *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. Referring to Figure 17.5,

*V*(*G*) = 6 regions

*V*(*G*) = 17 edges \_ 13 nodes + 2 = 6

*V*(*G*) = 5 predicate nodes + 1 = 6



`zzzzzzzz AASAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

**3. Determine a basis set of linearly independent paths.** The value of *V*(*G*) provides the number of linearly independent paths through the program control structure. In the case of procedure *average,* we expect to specify six paths:

path 1: 1-2-10-11-13

path 2: 1-2-10-12-13

path 3: 1-2-3-10-11-13

path 4: 1-2-3-4-5-8-9-2-. . .

path 5: 1-2-3-4-5-6-8-9-2-. . .

path 6: 1-2-3-4-5-6-7-8-9-2-. .

**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. Test cases that satisfy the basis set just described are

**Path 1 test case:**

value(*k*) = valid input, where *k* < *i* for 2 ≤ *i* ≤ 100

value(*i*) = \_999 where 2 ≤ *i* ≤ 100

*Expected results:* Correct average based on *k* values and proper totals.

*Note:* Path 1 cannot be tested stand-alone but must be tested as part of path 4, 5, and 6 tests.

**Path 2 test case:**

value(1) = \_999

*Expected results:* Average = \_999; other totals at initial values.

**Path 3 test case:**

Attempt to process 101 or more values.

First 100 values should be valid.

*Expected results:* Same as test case 1.

**Path 4 test case:**

value(*i*) = valid input where i < 100

value(*k*) < minimum where *k* < *i*

*Expected results:* Correct average based on *k* values and proper totals

**Path 5 test case:**

value(*i*) = valid input where *i* < 100

value(*k*) > maximum where *k* <= *i*

*Expected results:* Correct average based on *n* values and proper totals.

**Path 6 test case:**

value(*i*) = valid input where *i* < 100

*Expected results:* Correct average based on *n* values and proper totals.

17.4.4 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. A simple example of a flow graph and its corresponding graph matrix [BEI90] is shown in Figure 17.6. Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge *b.*



17.5 CONTROL STRUCTURE TESTING

17.5.3 Loop Testing

*Loop testing* is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops [BEI90] can be defined: simple loops, concatenated loops, nested loops, and unstructured loops (Figure 17.8).

**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.

**Nested loops.** If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer [BEI90] suggests an approach that will help to reduce the number of tests:

**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.



**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.

**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.