White-Box Testing

Path Coverage

A test suite achieves path coverage if it executes each linearly independent paths (or basis paths) at least once. A linearly independent path can be defined in terms of the control flow graph (CFG) of a program.

Control flow graph (CFG)

A CFG is a directed graph consisting of a set of nodes and edges (N, E), such that each node n ∈ N corresponds to a unique program statement and an edge exists between two nodes if control can transfer from one node to the other. We can easily draw the CFG for any program, if we know how to represent the sequence, selection, and iteration types of statements in the CFG.

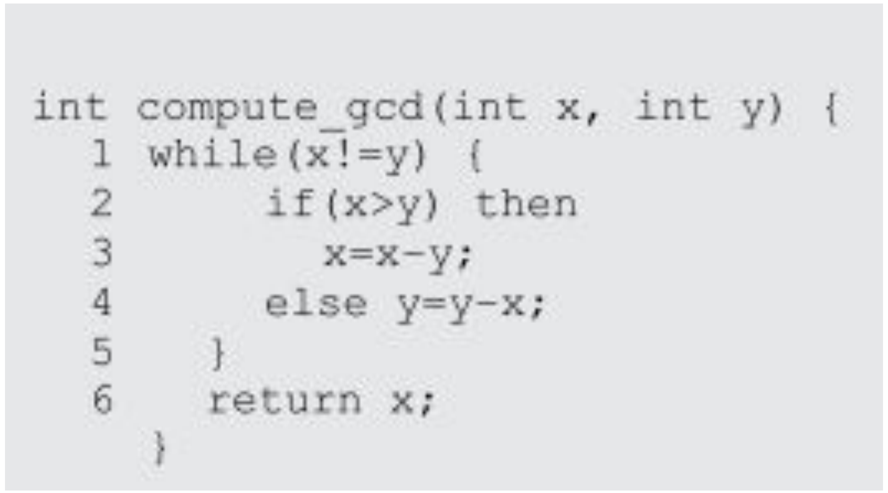
- A control flow graph describes how the control flows through the program.
- In order to draw the control flow graph of a program, we need to first number all the statements of a program.
- The different numbered statements serve as nodes of the control flow graph.
- There exists an edge from one node to another, if the execution of the statement representing the first node can result in the transfer of control to the other node.

Control flow graph (CFG)

- The CFG for any program can be easily drawn, if it is known how to represent the sequence, selection, and iteration types of statements in the CFG. After all, every program is constructed by using these three types of constructs only.
- The CFG representation of the sequence and decision types of statements is straight forward.
- For iteration type of constructs such as the while construct, the loop condition is tested only at the beginning of the loop and therefore always control flows from the last statement of the loop to the top of the loop. That is, the loop construct terminates from the first statement (after the loop is found to be false) and does not at any time exit the loop at the last statement of the loop.

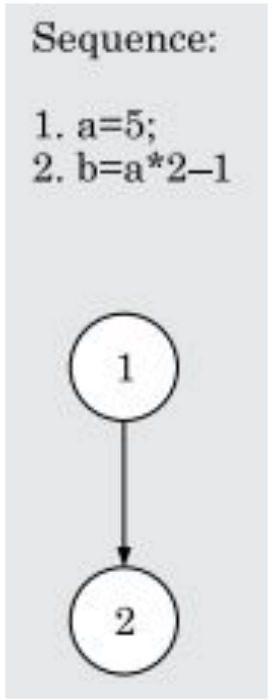
Example: Using the basic ideas, the CFG of the program given in Figure(a) can be drawn as shown in Figure (b).

Control flow graph (CFG)



```
int compute_gcd(int x, int y) {
    while(x!=y) {
        if(x>y) then
          x=x-y;
        else y=y-x;
    }
    return x;
  }
```

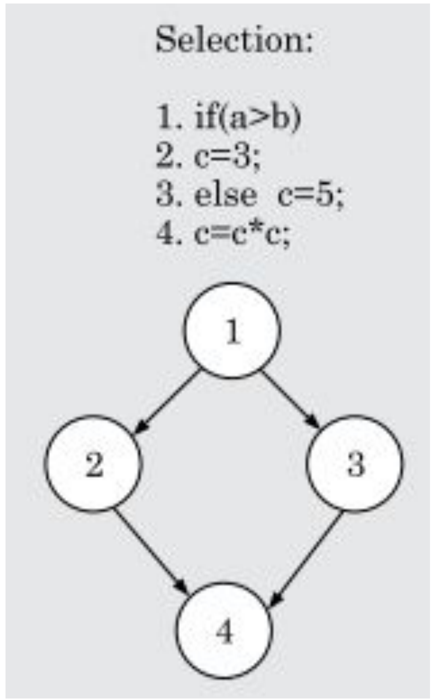
(a)



Sequence:

1. a=5;
2. b=a*2–1

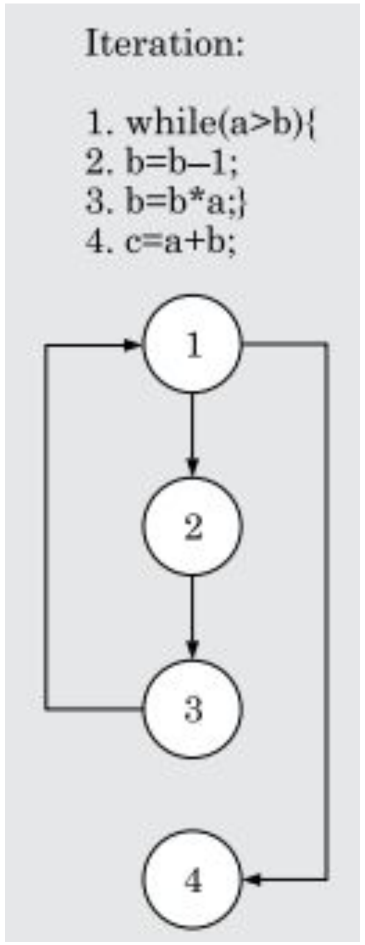
(b)



Selection:

1. if(a>b)
2. c=3;
3. else c=5;
4. c=c*c;

(c)



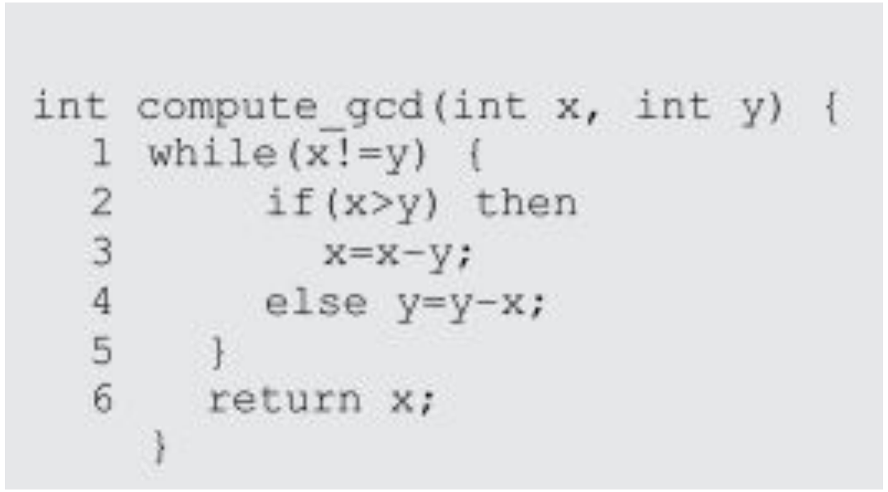
Iteration:

1. while(a>b){
2. b=b–1;
3. b=b*a;}
4. c=a+b;

(d)

Figure(a): Example Program.
Figure(b): CFG for Sequence of statements.
Figure(c): CFG for Selection Construct.
Figure(d): CFG for Iterative Construct.

Control flow graph (CFG)



```
int compute_gcd(int x, int y) {
  1 while(x!=y) {
  2       if(x>y) then
  3          x=x-y;
  4       else y=y-x;
  5    }
  6    return x;
   }
```

Figure (a): Example program.

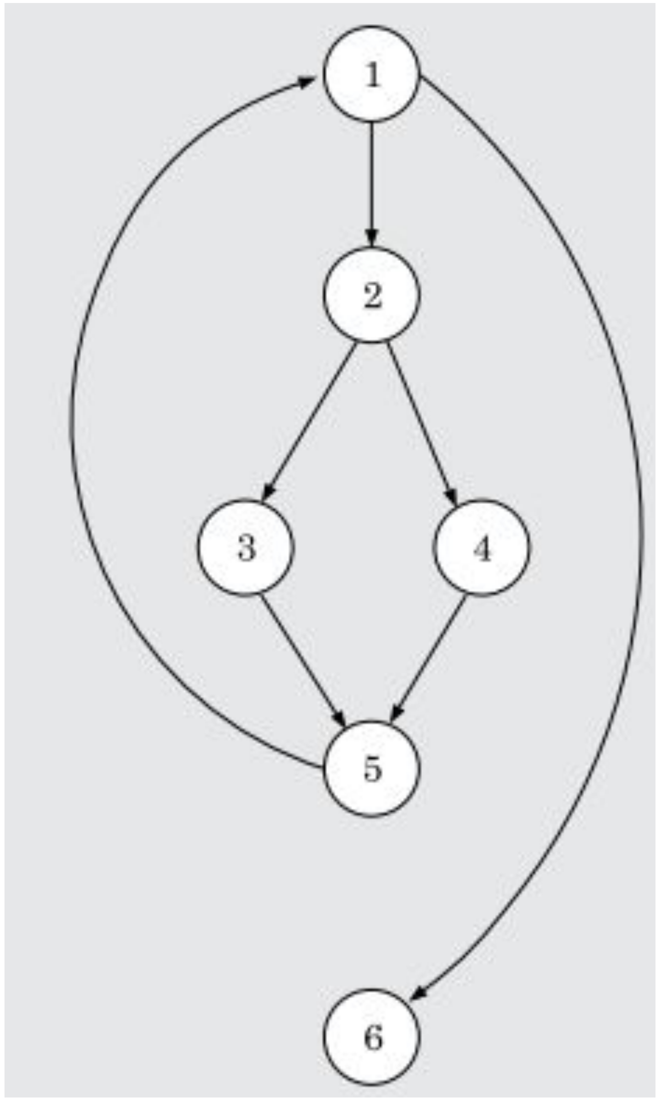


Figure (b): Control-flow graph.

Path

- A path through a program is any node and edge sequence from the start node to a terminal node of the control flow graph of a program.
- A program can have more than one terminal nodes when it contains multiple exit or return type of statements.
- Writing test cases to cover all paths of a typical program is impractical since there can be an infinite number of paths through a program in presence of loops.
- If coverage of all paths is attempted, then the number of test cases required would become infinitely large.
- Testing of all paths is impractical.
- Path coverage testing does not try to cover all paths, but only a subset of paths called linearly independent paths (or basis paths) is tested.

Linearly independent set of paths (or basis path set)

A set of paths for a given program is called linearly independent set of paths (or the set of basis paths or simply the basis set), if each path in the set introduces at least one new edge that is not included in any other path in the set. If a path has one new node compared to all other linearly independent paths, then the path is also linearly independent.

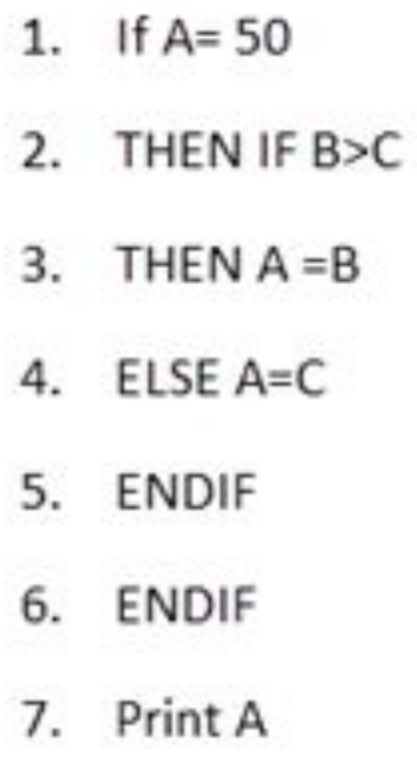
In the above example, we can see there are few conditional statements that is executed depending on what condition it suffice. Here there are 3 paths or condition that need to be tested to get the output.

The paths are-

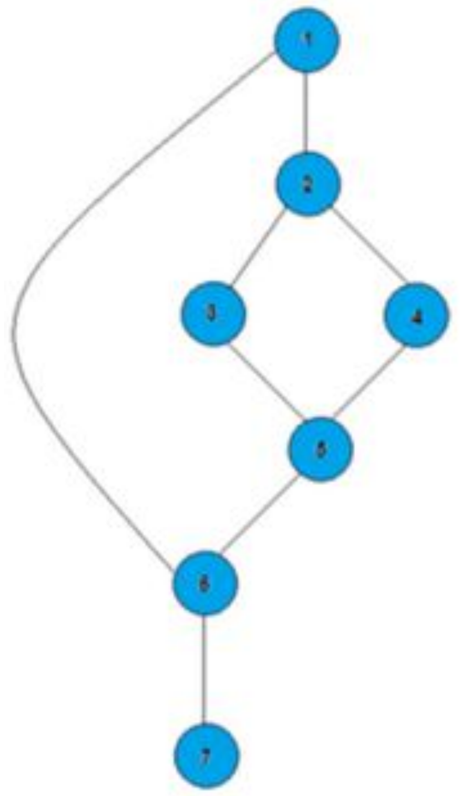
Path 1: 1,2,3,5,6, 7

Path 2: 1,2,4,5,6, 7

Path 3: 1, 6, 7



```
1.  If A= 50
2.  THEN IF B>C
3.  THEN A =B
4.  ELSE A=C
5.  ENDIF
6.  ENDIF
7.  Print A
```



McCabe's Cyclomatic Complexity Metric

- It is easy and straight forward to identify the linearly independent paths for a simple program.
- For complex programs it is not easy to determine the number of independent paths.
- McCabe's cyclomatic complexity metric is an important result that helps us to compute the number of linearly independent paths for any arbitrary program.
- McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program.
- McCabe's metric does not directly identify the linearly independent paths, but it provides us with a practical way of determining approximately how many paths to look for.
- McCabe's metric is only an upper bound and does not give the exact number of paths.

McCabe’s Cyclomatic Complexity Metric

McCabe obtained his results by applying graph-theoretic techniques to the control flow graph of a program. McCabe’s cyclomatic complexity metric defines an upper bound on the number of independent paths in a program.

We have three different ways to compute the cyclomatic complexity. For structured programs, the results computed by all the three methods are guaranteed to agree.

Method 1: Given a control flow graph G of a program, the cyclomatic complexity V (G) can be computed as:

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

Where, N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

For the CFG of the example program shown in previous figure, $E = 7$ and $N = 6$. Therefore, the value of the Cyclomatic complexity $= 7 - 6 + 2 = 3$.

McCabe’s Cyclomatic Complexity Metric

Method 2: An alternate way of computing the cyclomatic complexity of a program is based on a visual inspection of the control flow graph is as follows—

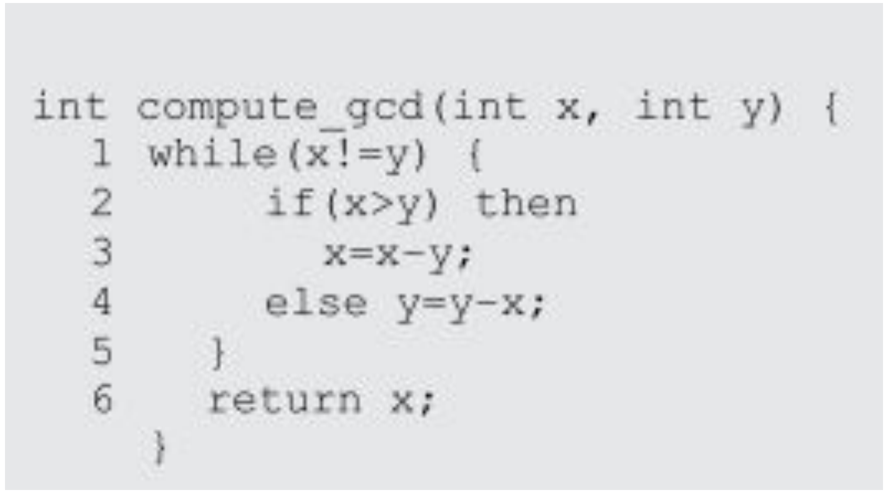
In this method, the cyclomatic complexity V (G) for a graph G is given by the following expression:

V(G) = Total number of non-overlapping bounded areas + 1

- In the program’s control flow graph G, any region enclosed by nodes and edges can be called as a bounded area.
- From a visual examination of the CFG given earlier, the number of bounded areas is 2. Therefore, the cyclomatic complexity, computed with this method is also 2+1=3.
- The number of bounded areas in a CFG increases with the number of decision statements and loops.

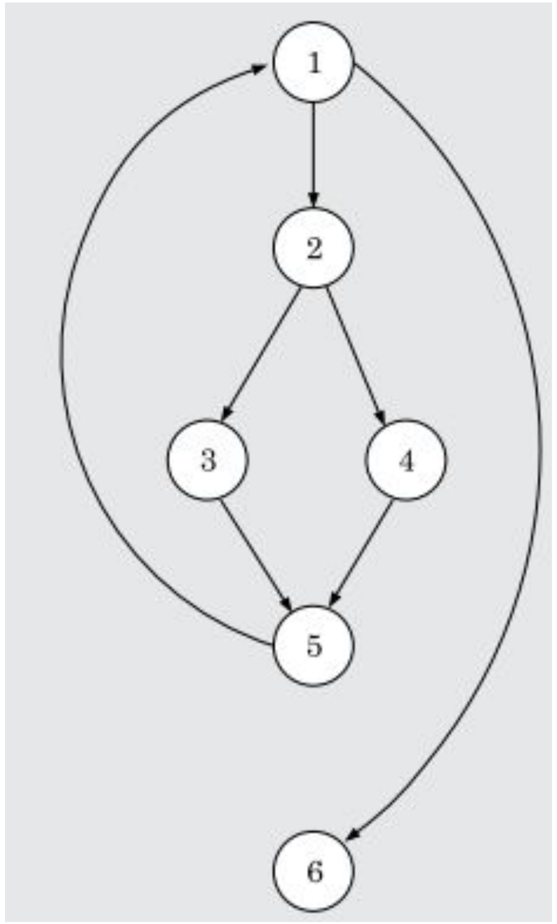
McCabe’s Cyclomatic Complexity Metric

Method 3: The cyclomatic complexity of a program can also be easily computed by computing the number of decision and loop statements of the program. If N is the number of decision and loop statements of a program, then the McCabe’s metric is equal to N + 1.



```
int compute_gcd(int x, int y) {
  1 while(x!=y) {
  2       if(x>y) then
  3          x=x-y;
  4       else y=y-x;
  5    }
  6    return x;
     }
```

In the given example, the total number of loop and decision statements is 2. So, the McCabe’s cyclomatic complexity is 2+1=3.



Steps to carry out path coverage-based testing

The following is the sequence of steps that need to be undertaken for deriving the path coverage-based test cases for a program:

1. Draw control flow graph for the program.

2. Determine the McCabe's metric V(G).

3. Determine the cyclomatic complexity. This gives the minimum number of test cases required to achieve path coverage.

4. repeat Test using a randomly designed set of test cases. Perform dynamic analysis to check the path coverage achieved. until at least 90 per cent path coverage is achieved.