B.Tech (CSE)– IV-I Sem
(19A05702T) SOFTWARE TESTING
[REG-19]
2022-23

UNIT-1

Transaction Flow Testing: Transaction Flows, Transaction Flow Testing Techniques.

Dataflow testing: Basics of Dataflow Testing, Strategies in Dataflow Testing, Application of Dataflow Testing

FLOWGRAPHS AND PATH TESTING:

This unit gives an in-depth overview of path testing and its applications.

At the end of this unit, the student will be able to:

- Understand the concept of path testing.
- Identify the components of a control flow diagram and compare the same with a flowchart.
- Represent the control flow graph in the form of a Linked List notation.
- Understand the path testing and selection criteria and their limitations.
- Interpret a control flow-graph and demonstrate the complete path testing to achieve C1+C2.
- Classify the predicates and variables as dependant/independent and correlated/uncorrelated.
- Understand the path sensitizing method and classify whether the path is achievable or not.
- Identify the problem due to co-incidental correctness and choose a path instrumentation method to overcome the problem.

BASICS OF PATH TESTING:

TOP

•PATH TESTING:

- Path Testing is the name given to a family of test techniques based on judiciously selecting a set of test paths through the program.
- If the set of paths are properly chosen then we have achieved some measure of test thoroughness. For example, pick enough paths to assure that every source statement has been executed at least once.
- Path testing techniques are the oldest of all structural test techniques.
- Path testing is most applicable to new software for unit testing. It is a structural technique.
- It requires complete knowledge of the program's structure.

- It is most often used by programmers to unit test their own code.
- The effectiveness of path testing rapidly deteriorates as the size of the software aggregate under test increases.

- **THE BUG ASSUMPTION:**
  - The bug assumption for the path testing strategies is that something has gone wrong with the software that makes it take a different path than intended.
  - As an example "GOTO X" where "GOTO Y" had been intended.
  - Structured programming languages prevent many of the bugs targeted by path testing: as a consequence the effectiveness for path testing for these languages is reduced and for old code in COBOL, ALP, FORTRAN and Basic, the path testing is indispensable.
- **CONTROL FLOW GRAPHS:**
  - The control flow graph is a graphical representation of a program's control structure. It uses the elements named process blocks, decisions, and junctions.
  - The flow graph is similar to the earlier flowchart, with which it is not to be confused.
  - **Flow Graph Elements:**A flow graph contains four different types of elements. (1) Process Block (2) Decisions (3) Junctions (4) Case Statements
    1. **Process Block:**
       - A process block is a sequence of program statements uninterrupted by either decisions or junctions.
       - It is a sequence of statements such that if any one of statement of the block is executed, then all statement thereof is executed.
       - Formally, a process block is a piece of straight line code of one statement or hundreds of statements.
       - A process has one entry and one exit. It can consist of a single statement or instruction, a sequence of statements or instructions, a single entry/exit subroutine, a macro or function call, or a sequence of these.
    2. **Decisions:**
       - A decision is a program point at which the control flow can diverge.
       - Machine language conditional branch and conditional skip instructions are examples of decisions.
       - Most of the decisions are two-way but some are three way branches in control flow.
    3. **Case Statements:**
       - A case statement is a multi-way branch or decisions.
       - Examples of case statement are a jump table in assembly language, and the PASCAL case statement.
       - From the point of view of test design, there are no differences between Decisions and Case Statements
    4. **Junctions:**
       - A junction is a point in the program where the control flow can merge.

- Examples of junctions are: the target of a jump or skip instruction in ALP, a label that is a target of GOTO.

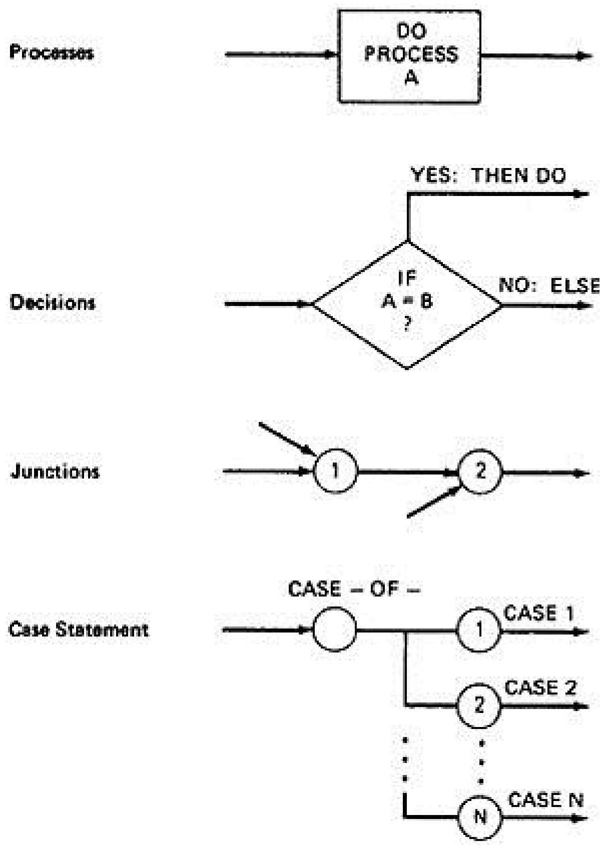


Figure 2.1: Flow graph Elements

CONTROL FLOW GRAPHS Vs FLOWCHARTS:

A program's flow chart resembles a control flow graph.

In flow graphs, we don't show the details of what is in a process block. In flow charts every part of the process block is drawn.

The flowchart focuses on process steps, where as the flow graph focuses on control flow of the program.

The act of drawing a control flow graph is a useful tool that can help us clarify the control flow and data flow issues.

NOTATIONAL EVOULTION:

The control flow graph is simplified representation of the program's structure. The notation changes made in creation of control flow graphs:

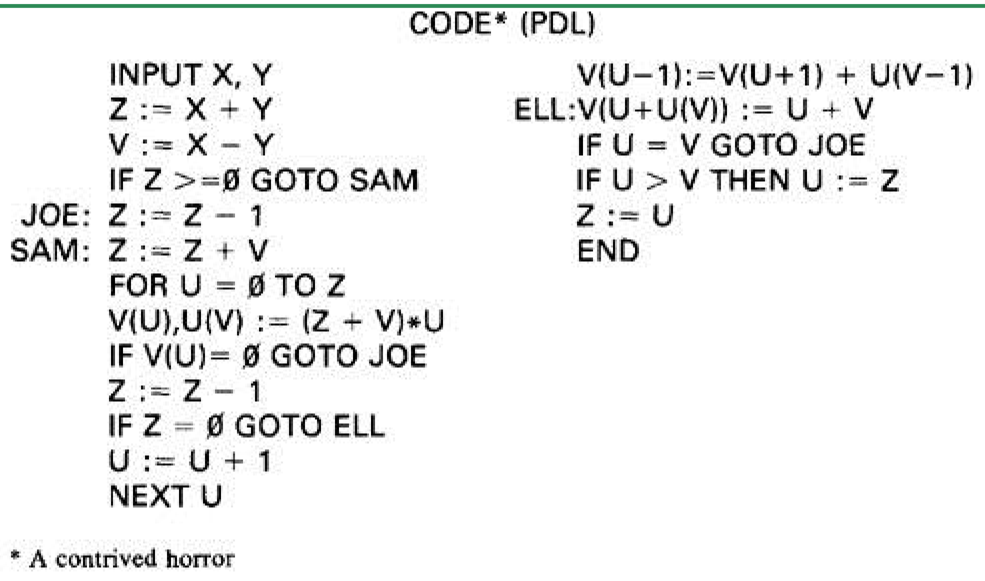
▪The process boxes weren't really needed. There is an implied process on every line joining junctions and decisions.

▪We don't need to know the specifics of the decisions, just the fact that there is a branch.

▪The specific target label names aren't important-just the fact that they exist. So we can replace them by simple numbers.

▪To understand this, we will go through an example (Figure 2.2) written in a FORTRAN like programming language called **Programming Design Language (PDL)**. The program's corresponding flowchart (Figure 2.3) and flowgraph (Figure 2.4) were also provided below for better understanding.

▪The first step in translating the program to a flowchart is shown in Figure 2.3, where we have the typical one- for-one classical flowchart. Note that complexity has increased, clarity has decreased, and that we had to add auxiliary labels (LOOP, XX, and YY), which have no actual program counterpart. In Figure 2.4 we merged the process steps and replaced them with the single process box. We now have a control flow graph. But this representation is still too busy. We simplify the notation further to achieve Figure 2.5, where for the first time
we can really see what the control flow looks like.



```
                        CODE* (PDL)

      INPUT X, Y
      Z := X + Y
      V := X - Y
      IF Z >=Ø GOTO SAM
JOE:  Z := Z - 1
SAM:  Z := Z + V
      FOR U = Ø TO Z
      V(U),U(V) := (Z + V)*U
      IF V(U)= Ø GOTO JOE
      Z := Z - 1
      IF Z = Ø GOTO ELL
      U := U + 1
      NEXT U
      V(U-1):=V(U+1) + U(V-1)
ELL:V(U+U(V)) := U + V
      IF U = V GOTO JOE
      IF U > V THEN U := Z
      Z := U
      END
```

* A contrived horror

Figure 2.2: Program Example (PDL)

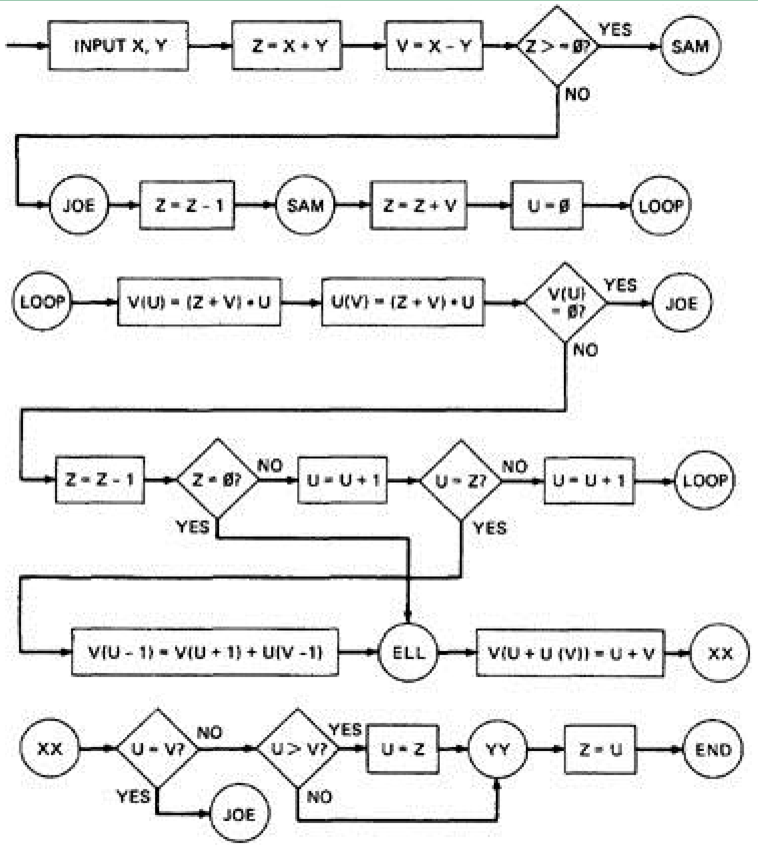


Figure 2.3: One-to-one flowchart for example program in Figure 2.2

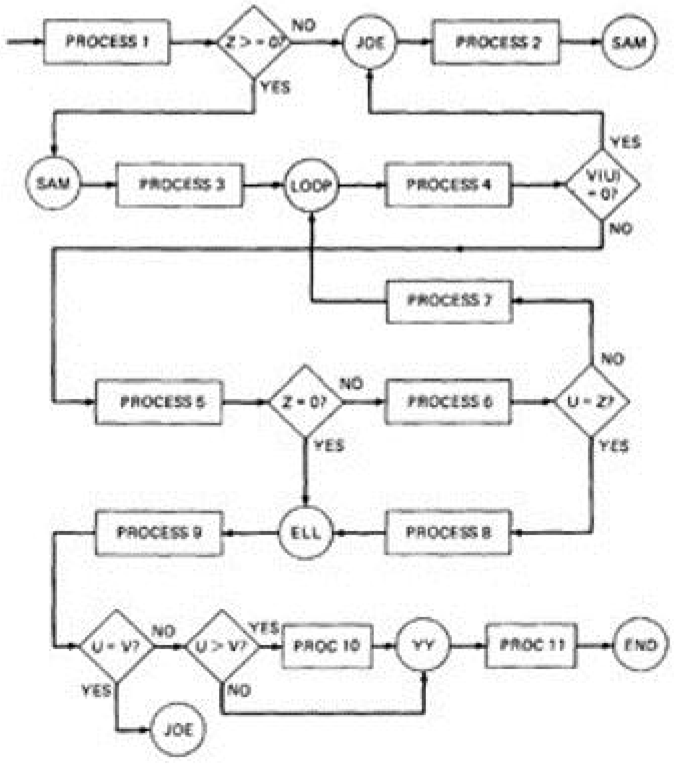


Figure 2.4: Control Flowgraph for example in Figure 2.2

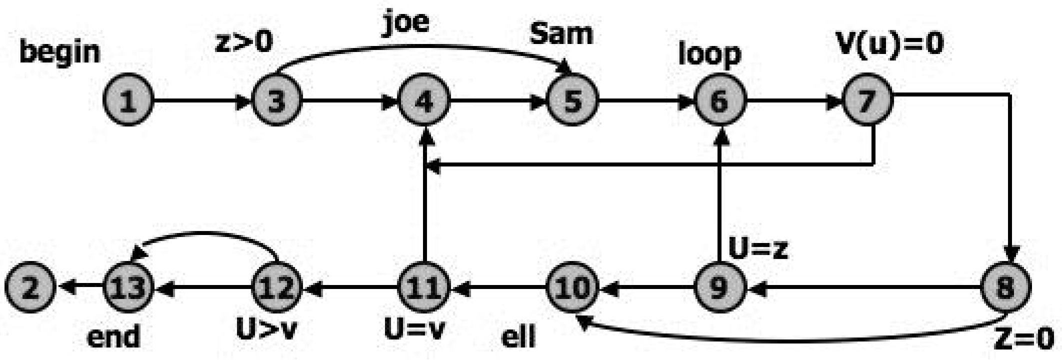


Figure 2.5: Simplified Flowgraph Notation

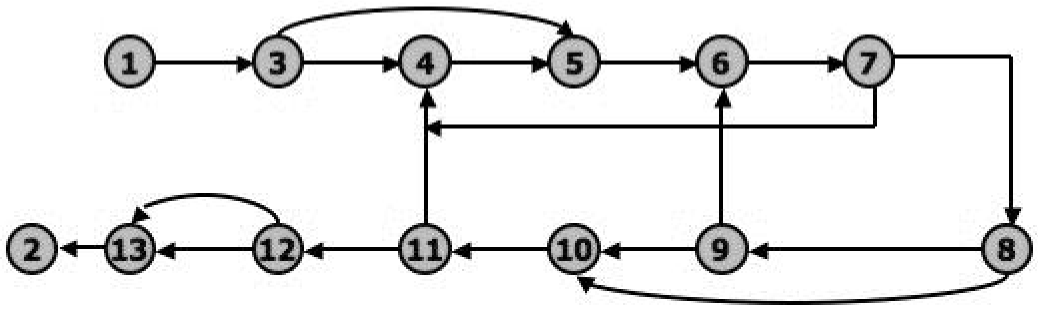
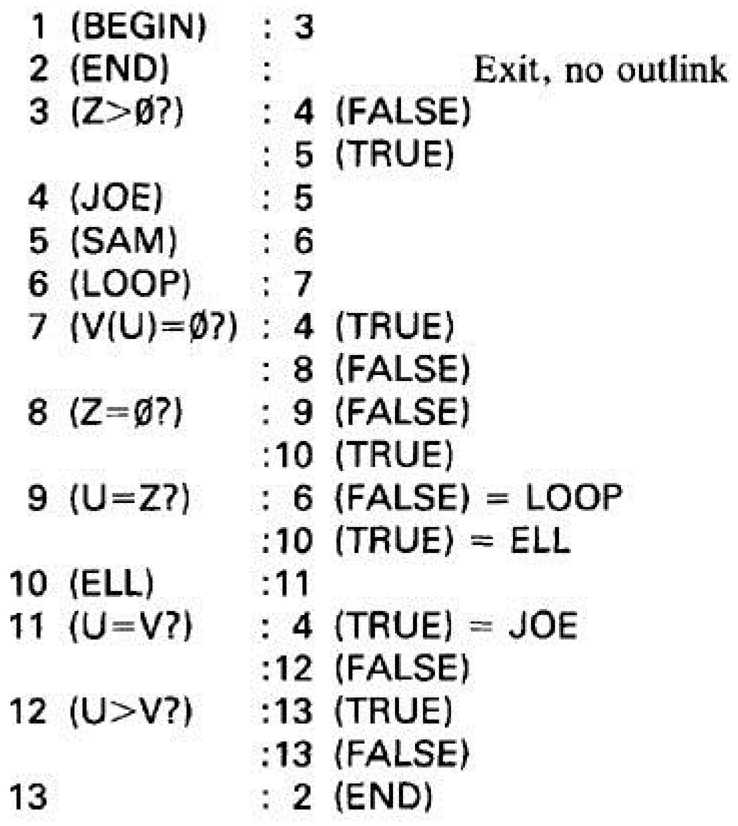


Figure 2.6: Even Simplified Flowgraph Notation

The final transformation is shown in Figure 2.6, where we've dropped the node numbers to achieve an even simpler representation. The way to work with control flowgraphs is to use the simplest possible representation
- that is, no more information than you need to correlate back to the source program or PDL.

LINKED LIST REPRESENTATION:
Although graphical representations of flowgraphs are revealing(=HELPFUL), the details of the control flow inside a program they are often inconvenient.
In linked list representation, each node has a name and there is an entry on the list for each link in the flow graph. only the information pertinent(=RELATED) to the control flow is shown.
Linked List representation of Flow Graph:



```
 1  (BEGIN)     :  3
 2  (END)       :              Exit, no outlink
 3  (Z>Ø?)      :  4  (FALSE)
                :  5  (TRUE)
 4  (JOE)       :  5
 5  (SAM)       :  6
 6  (LOOP)      :  7
 7  (V(U)=Ø?)   :  4  (TRUE)
                :  8  (FALSE)
 8  (Z=Ø?)      :  9  (FALSE)
                :10  (TRUE)
 9  (U=Z?)      :  6  (FALSE) = LOOP
                :10  (TRUE) = ELL
10  (ELL)       :11
11  (U=V?)      :  4  (TRUE) = JOE
                :12  (FALSE)
12  (U>V?)      :13  (TRUE)
                :13  (FALSE)
13              :  2  (END)
```

Figure 2.7: Linked List Control Flowgraph Notation

FLOWGRAPH - PROGRAM CORRESPONDENCE:

A flow graph is a pictorial representation of a program and not the program itself, just as a topographic map.

You can’t always associate the parts of a program in a unique way with flowgraph parts because many program structures, such as if-then-else constructs, consists of a combination of decisions, junctions, and processes.

The translation from a flowgraph element to a statement and vice versa is not always unique. (See Figure 2.8)

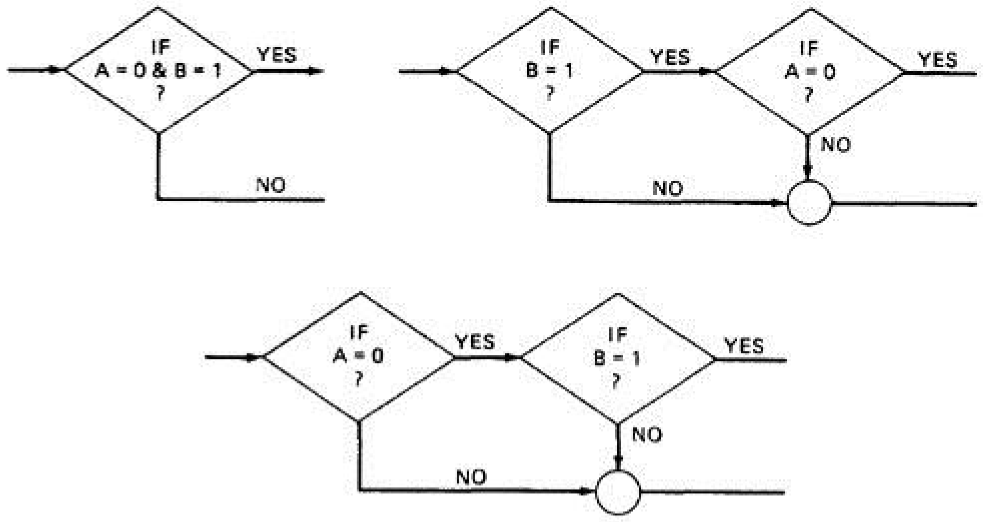


Figure 2.8: Alternative Flowgraphs for same logic (Statement "IF (A=0) AND (B=1) THEN . . .").

An improper translation from flowgraph to code during coding can lead to bugs and improper translation during the test design lead to missing test cases and causes undiscovered bugs.

FLOWGRAPH AND FLOWCHART GENERATION:

Flowcharts can be

1.Handwritten by the programmer.

2.Automatically produced by a flowcharting program based on a mechanical analysis of the source code.

3.Semi automatically produced by a flow charting program based in part on structural analysis of the source code and in part on directions given by the programmer.

There are relatively few control flow graph generators.

PATH TESTING - PATHS, NODES AND LINKS:

Path: a path through a program is a sequence of instructions or statements that starts at an entry, junction, or decision and ends at another, or possibly the same junction, decision, or exit.

A path may go through several junctions, processes, or decisions, one or more times. Paths consist of segments.

The segment is a link - a single process that lies between two nodes.

A path segment is succession of consecutive links that belongs to some path.

The length of path measured by the number of links in it and not by the number of the instructions or statements executed along that path.

The name of a path is the name of the nodes along the path.

FUNDAMENTAL PATH SELECTION CRITERIA:

There are many paths between the entry and exit of a typical routine.

Every decision doubles the number of potential paths. And every loop multiplies the number of potential paths by the number of different iteration values possible for the loop.

Defining complete testing:

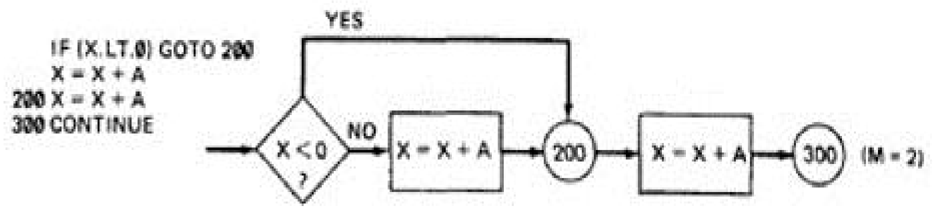
1.Exercise(=WORK OUT) every path from entry to exit

2.Exercise every statement or instruction at least once

3.Exercise every branch and case statement, in each direction at least once

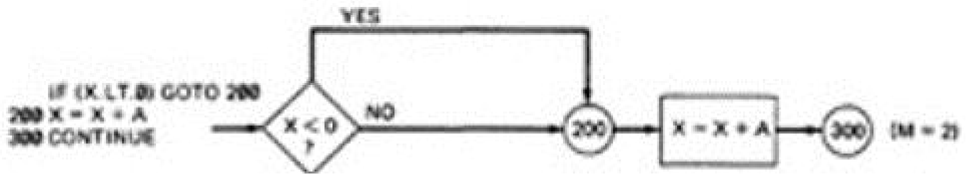
If prescription 1 is followed then 2 and 3 are automatically followed. But it is impractical for most routines. It can be done for the routines that have no loops, in which it is equivalent to 2 and 3 prescriptions.

***EXAMPLE:* Here is the correct version.**

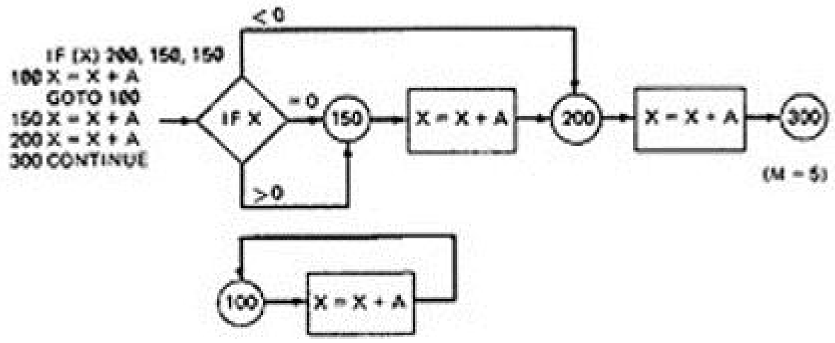


```
    IF (X.LT.0) GOTO 200
    X = X + A
200 X = X + A
300 CONTINUE
```

For X negative, the output is X + A, while for X greater than or equal to zero, the output is X + 2A. Following prescription 2 and executing every statement, but not every branch, would not reveal the bug in the following incorrect version:



A negative value produces the correct answer. Every statement can be executed, but if the test cases do not force each branch to be taken, the bug can remain hidden. The next example uses a test based on executing each branch but does not force the execution of all statements:



The hidden loop around label 100 is not revealed by tests based on prescription 3 alone because no test forces the execution of statement 100 and the following GOTO statement. Furthermore, label 100 is not flagged by the compiler as an unreferenced label and the subsequent GOTO does not refer to an undefined label.

A **Static Analysis** (that is, an analysis based on examining the source code or structure) cannot determine whether a piece of code is or is not reachable. There could be subroutine calls with parameters that are subroutine labels, or in the above example there could be a GOTO that targeted label 100 but could never achieve a value that would send the program to that label.

Only a **Dynamic Analysis** (that is, an analysis based on the code's behavior while running - which is to say, to all intents and purposes, testing) can determine whether code is reachable or not and therefore distinguish between the ideal structure we think we have and the actual, buggy structure.

PATH TESTING CRITERIA:

Any testing strategy based on paths must at least both exercise every instruction and take branches in all directions.

A set of tests that does this is not complete in an absolute sense, but it is complete in the sense that anything less must leave something untested.

So we have explored three different testing criteria or strategies out of a potentially infinite family of strategies.

0. **Path Testing (P_{inf})**:
   - ▪Execute all possible control flow paths through the program: typically, this is restricted to all possible entry/exit paths through the program.
   - ▪If we achieve this prescription, we are said to have achieved 100% path coverage. This is the strongest criterion in the path testing strategy family: it is generally impossible to achieve.
1. **Statement Testing (P_1)**:
   - ▪Execute all statements in the program at least once under some test. If we do enough tests to achieve this, we are said to have achieved 100% statement coverage.
   - ▪An alternate equivalent characterization is to say that we have achieved 100% node coverage. We denote this by C1.
   - ▪This is the weakest criterion in the family: testing less than this for new software is unconscionable (unprincipled or cannot be accepted) and should be criminalized.
2. **Branch Testing (P_2)**:
   - Execute enough tests to assure that every branch alternative has been exercised at least once under some test.
   - If we do enough tests to achieve this prescription, then we have achieved 100% branch coverage.
   - An alternative characterization is to say that we have achieved 100% link coverage.
   - For structured software, branch testing and therefore branch coverage strictly includes statement coverage.
   - We denote branch coverage by C2.

Commonsense and Strategies:

▪Branch and statement coverage are accepted today as the minimum mandatory testing requirement.

▪The question "why not use a judicious sampling of paths?, what is wrong with leaving some code, untested?" is ineffectual in the view of common sense and experience since: **(1.)** Not testing a piece of a code leaves a residue of bugs in the program in proportion to the size of the untested code and the probability of bugs. **(2.)** The high probability paths are always thoroughly tested if only to demonstrate that the system works properly.

▪**Which paths to be tested?** You must pick enough paths to achieve C1+C2. The question of what is the fewest number of such paths is interesting to the designer of test tools that help automate the path testing, but it is not crucial to the pragmatic (practical) design of tests. It is better to make many simple paths than a few complicated paths.

▪**Path Selection Example:**

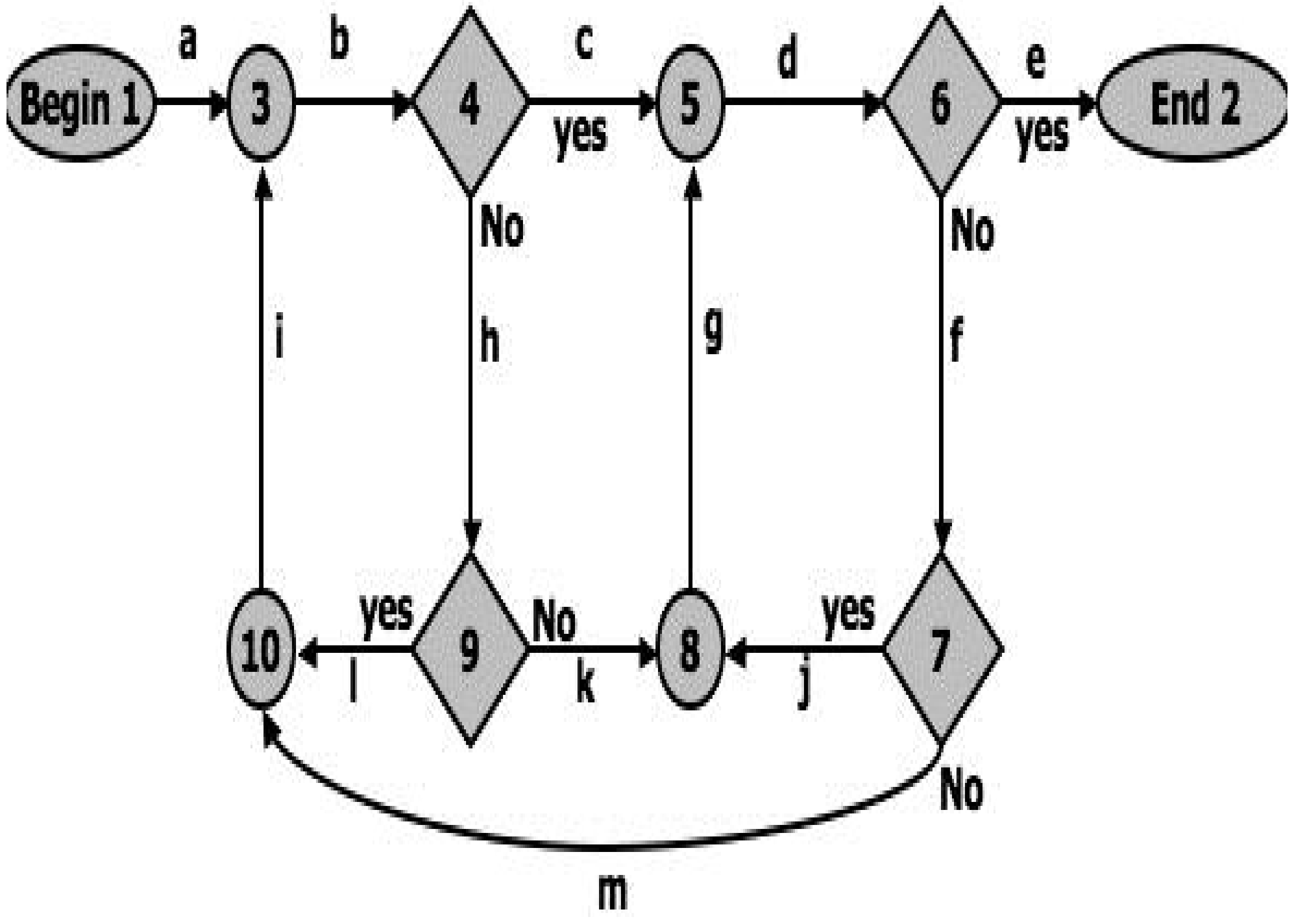


Figure 2.9: An example flowgraph to explain path selection

- **Practical Suggestions in Path Testing:**
  1. Draw the control flow graph on a single sheet of paper.
  2. Make several copies - as many as you will need for coverage (C1+C2) and several more.
  3. Use a yellow highlighting marker to trace paths. Copy the paths onto master sheets.
  4. Continue tracing paths until all lines on the master sheet are covered, indicating that you appear to have achieved C1+C2.
  5. As you trace the paths, create a table that shows the paths, the coverage status of each process, and each decision.
  6. The above paths lead to the following table considering Figure 2.9:

After you have traced a covering path set on the master sheet and filled in the table for every path, check the following:

1. Does every decision have a YES and a NO in its column? (C2)
2. Has every case of all case statements been marked? (C2)
3. Is every three - way branch (less, equal, greater) covered? (C2)
4. Is every link (process) covered at least once? (C1)

6. Revised Path Selection Rules:

- Pick the simplest, functionally sensible entry/exit path.
- Pick additional paths as small variation from previous paths. Pick paths that do not have loops rather than paths that do. Favor short paths that make sense over paths that don't.
- Pick additional paths that have no obvious functional meaning only if it's necessary to provide coverage.
- Be comfortable with your chosen paths. Play your hunches (guesses) and give your intuition free reign as long as you achieve C1+C2.
- Don't follow rules slavishly (blindly) - except for coverage.

LOOPS:

▪**Cases for a single loop:** A Single loop can be covered with two cases: Looping and Not looping. But, experience shows that many loop-related bugs are not discovered by C1+C2. Bugs hide themselves in corners and congregate at boundaries - in the cases of loops, at or around the minimum or maximum number of times the loop can be iterated. The minimum number of iterations is often zero, but it need not be.

CASE 1: Single loop, Zero minimum, N maximum, No excluded values

1. Try bypassing the loop (zero iteration). If you can't, you either have a bug, or zero is not the minimum and you have the wrong case.
2. Could the loop-control variable be negative? Could it appear to specify a negative number of iterations? What happens to such a value?
3. One pass through the loop.
4. Two passes through the loop.
5. A typical number of iterations, unless covered by a previous test.
6. One less than the maximum number of iterations.
7. The maximum number of iterations.
8. Attempt one more than the maximum number of iterations. What prevents the loop-control variable from having this value? What will happen with this value if it is forced?

CASE 2: Single loop, Non-zero minimum, No excluded values

8. Try one less than the expected minimum. What happens if the loop control variable's value is less than the minimum? What prevents the value from being less than the minimum?
9. The minimum number of iterations.
10. One more than the minimum number of iterations.
11. Once, unless covered by a previous test.
12. Twice, unless covered by a previous test.
13. A typical value.
14. One less than the maximum value.
15. The maximum number of iterations.
16. Attempt one more than the maximum number of iterations.

CASE 3: Single loops with excluded values

▪Treat single loops with excluded values as two sets of tests consisting of loops without excluded values, such as case 1 and 2 above.

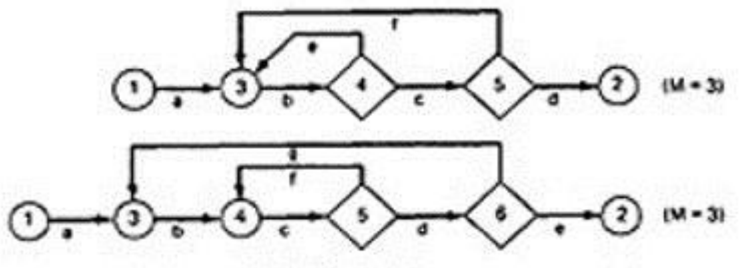
▪Example, the total range of the loop control variable was 1 to 20, but that values 7,8,9,10 were excluded. The two sets of tests are 1-6 and 11-20.

▪The test cases to attempt would be 0,1,2,4,6,7 for the first range and 10,11,15,19,20,21 for the second range.

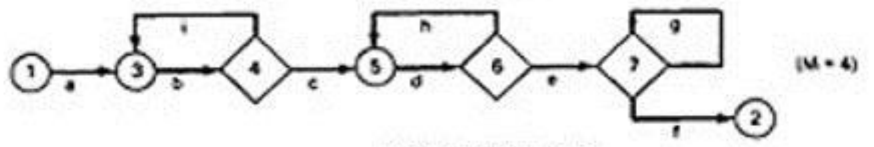
▪**Kinds of Loops:** There are only three kinds of loops with respect to path testing:

- **Nested Loops:**
  - The number of tests to be performed on nested loops will be the exponent of the tests performed on single loops.
  - As we cannot always afford to test all combinations of nested loops' iterations values. Here's a tactic used to discard some of these values:
    1. Start at the inner most loop. Set all the outer loops to their minimum values.
    2. Test the minimum, minimum+1, typical, maximum-1 , and maximum for the innermost loop, while holding the outer loops at their minimum iteration parameter values. Expand the tests as required for out of range and excluded values.
    3. If you've done the outmost loop, GOTO step 5, else move out one loop and set it up as in step 2 with all other loops set to typical values.
    4. Continue outward in this manner until all loops have been covered.
    5. Do all the cases for all loops in the nest simultaneously.
- **Concatenated Loops:**
  - Concatenated loops fall between single and nested loops with respect to test cases. Two loops are concatenated if it's possible to reach one after exiting the other while still on a path from entrance to exit.
  - If the loops cannot be on the same path, then they are not concatenated and can be treated as individual loops.
- **Horrible Loops:**
  - A horrible loop is a combination of nested loops, the use of code that jumps into and out of loops, intersecting loops, hidden loops, and cross connected loops.
  - Makes iteration value selection for test cases an awesome and ugly task, which is another

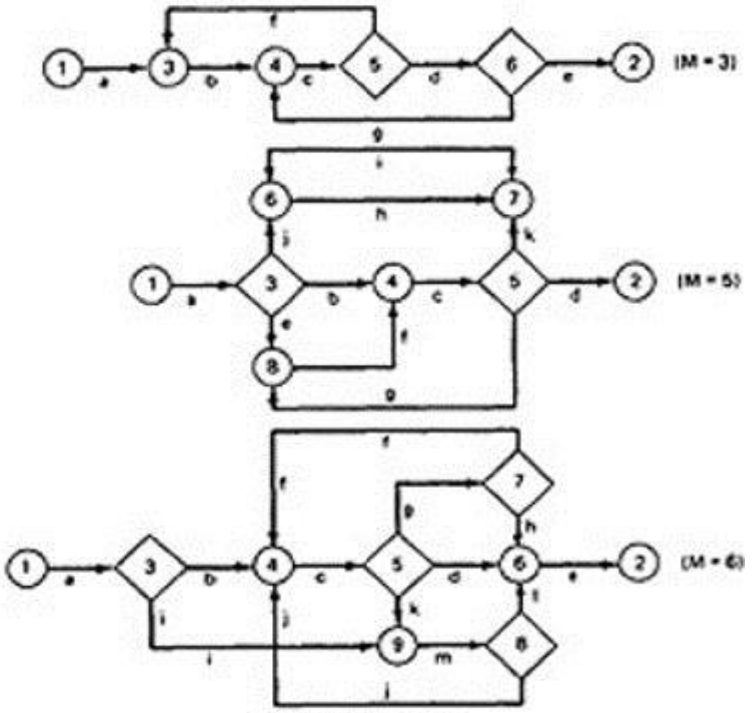
reason such structures should be avoided.



a, b) Nested Loops



c) Concatenated Loops



d, e, f) Horrible Loops