Software Testing

Software Testing

- What is Testing?

Many people understand many definitions of testing :

1. Testing is the process of demonstrating that errors are not present.

2. The purpose of testing is to show that a program performs its intended functions correctly.

3. Testing is the process of establishing confidence that a program does what it is supposed to do.

These definitions are incorrect.

Software Testing

A more appropriate definition is:

"*Testing is theprocess of executinga program with the intent of finding errors*."

Software Testing

- Why should We Test ?

Although software testing is itself an expensive activity, yet launching of software without testing may lead to cost potentially much higher than that of testing, specially in systems where human safety is involved.

In the software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal.

Software Testing

- Who should Do the Testing ?
  - Testing requires the developers to find errors from their software.
  - It is difficult for software developer to point out errors from own creations.
  - Many organisations have made a distinction between development and testing phase by making different people responsible for each phase.

Software Testing

- What should We Test ?

We should test the program's responses to every possible input. It means, we should test for all valid and invalid inputs. Suppose a program requires two 8 bit integers as inputs. Total possible combinations are $2^8 x 2^8$. If only one second it required to execute one set of inputs, it may take 18 hours to test all combinations. Practically, inputs are more than two and size is also more than 8 bits. We have also not considered invalid inputs where so many combinations are possible. Hence, complete testing is just not possible, although, we may wish to do so.

Software Testing

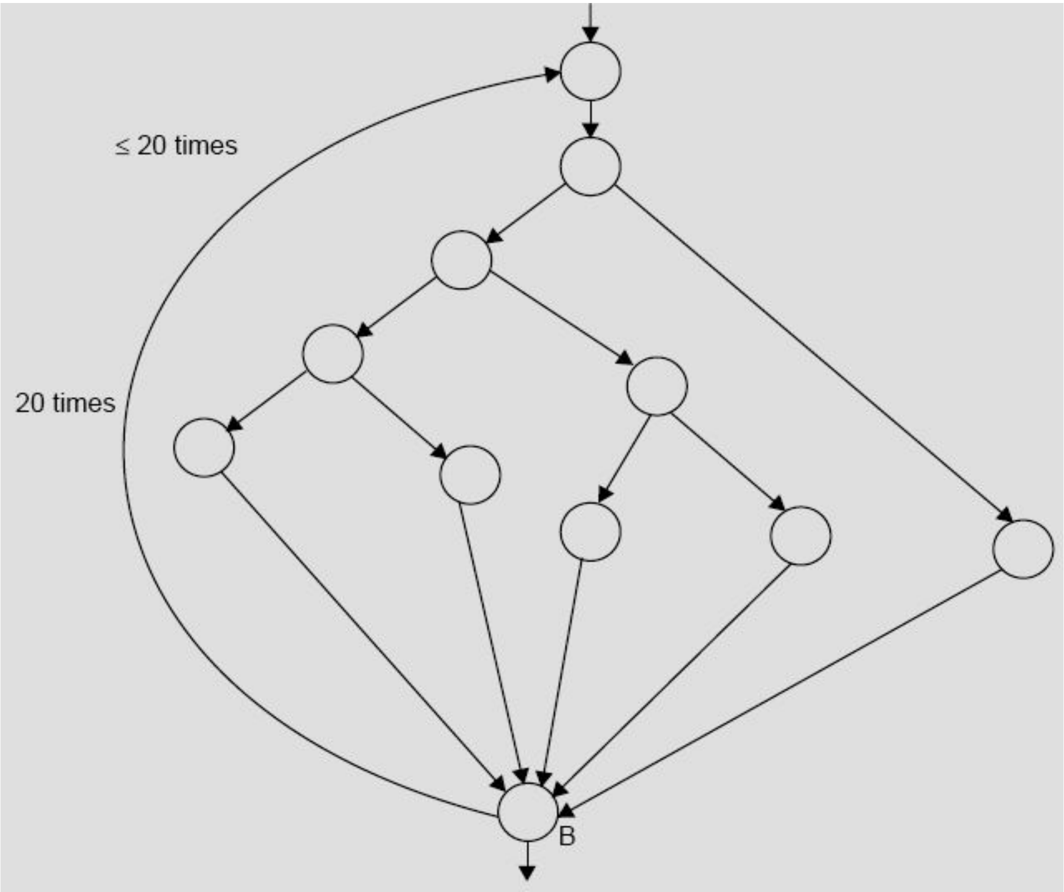


Fig. 1: Control flow graph

Software Testing

The number of paths in the example of Fig. 1 are 10^{14} or 100 trillions. It is computed from $5^{20} + 5^{19} + 5^{18} + \ldots\ldots + 5^{1}$; where 5 is the number of paths through the loop body. If only 5 minutes are required to test one test path, it may take approximately one billion years to execute every path.

Software Testing

Some Terminologies

➢ **Error, Mistake, Bug, Fault and Failure**

People make **errors**. A good synonym is **mistake**. This may be a syntax error or misunderstanding of specifications. Sometimes, there are logical errors.

When developers make mistakes while coding, we call these mistakes "**bugs**".

A **fault** is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc. Defect is a good synonym for fault.

A **failure** occurs when a fault executes. A particular fault may cause different failures, depending on how it has been exercised.

Software Testing

➢ Test, Test Case and Test Suite

Test and **Test case** terms are used interchangeably. In practice, both are same and are treated as synonyms. Test case describes an input description and an expected output description.

Test Case ID	
Section-I (Before Execution)	Section-II (After Execution)
Purpose :	Execution History:
Pre condition: (If any)	Result:
Inputs:	If fails, any possible reason (Optional);
Expected Outputs:	Any other observation:
Post conditions:	Any suggestion:
Written by:	Run by:
Date:	Date:

Fig. 2: Test case template

The set of test cases is called a **test suite**. Hence any combination of test cases may generate a test suite.

Software Testing

➢ Verification and Validation

Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Validation is the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements .

Testing= Verification+Validation

Software Testing

➢ Alpha, Beta and Acceptance Testing

The term **Acceptance Testing** is used when the software is developed for a specific customer. A series of tests are conducted to enable the customer to validate all requirements. These tests are conducted by the end user / customer and may range from adhoc tests to well planned systematic series of tests.

The terms alpha and beta testing are used when the software is developed as a product for anonymous customers.

Alpha Tests are conducted at the developer's site by some potential customers. These tests are conducted in a controlled environment. Alpha testing may be started when formal testing process is near completion.

Beta Tests are conducted by the customers / end users at their sites. Unlike alpha testing, developer is not present here. Beta testing is conducted in a real environment that cannot be controlled by the developer.

Software Testing

Functional Testing

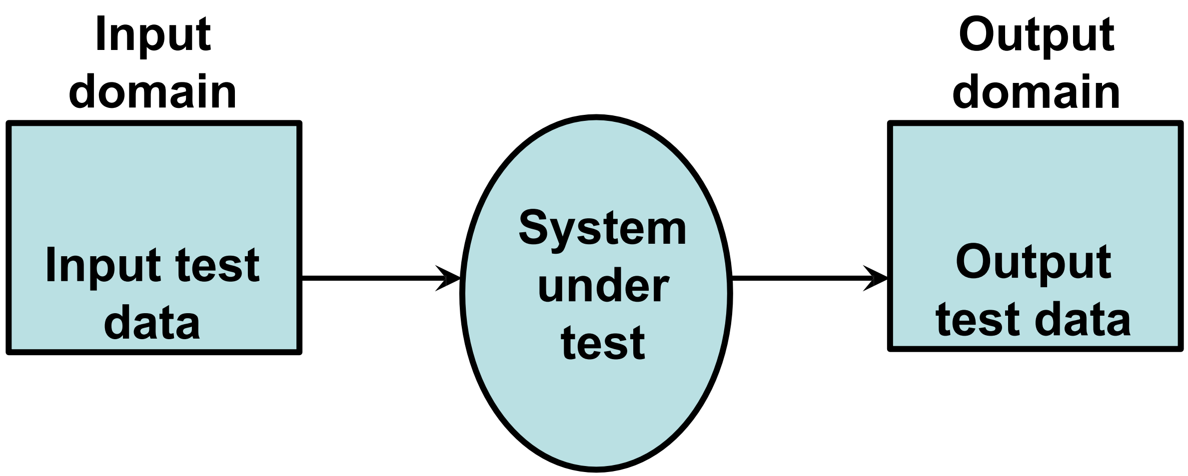


Fig. 3: Black box testing

Software Testing

Boundary Value Analysis

Consider a program with two input variables x and y. These input variables have specified boundaries as:

$$a \le x \le b$$

$$c \le y \le d$$

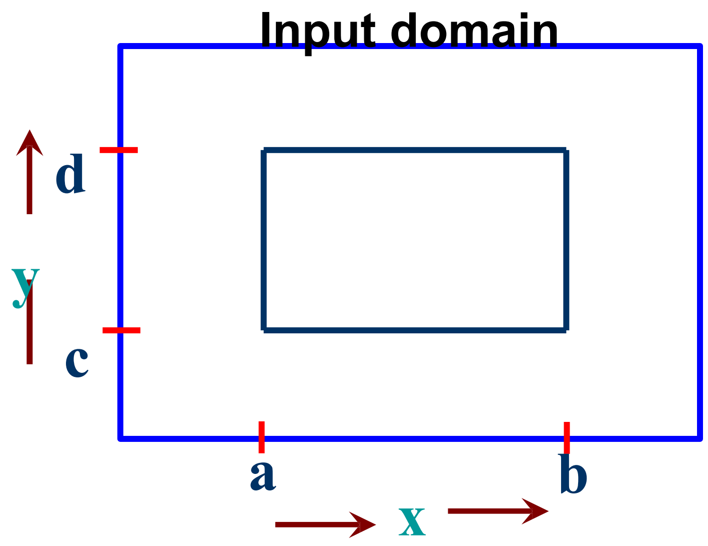


Fig.4: Input domain for program having two input variables

Software Testing

The boundary value analysis test cases for our program with two inputs variables (x and y) that may have any value from 100 to 300 are: (200,100), (200,101), (200,200), (200,299), (200,300), (100,200), (101,200), (299,200) and (300,200). This input domain is shown in Fig. 5. Each dot represent a test case and inner rectangle is the domain of legitimate inputs. Thus, for a program of n variables, boundary value analysis yield **4n + 1** test cases.

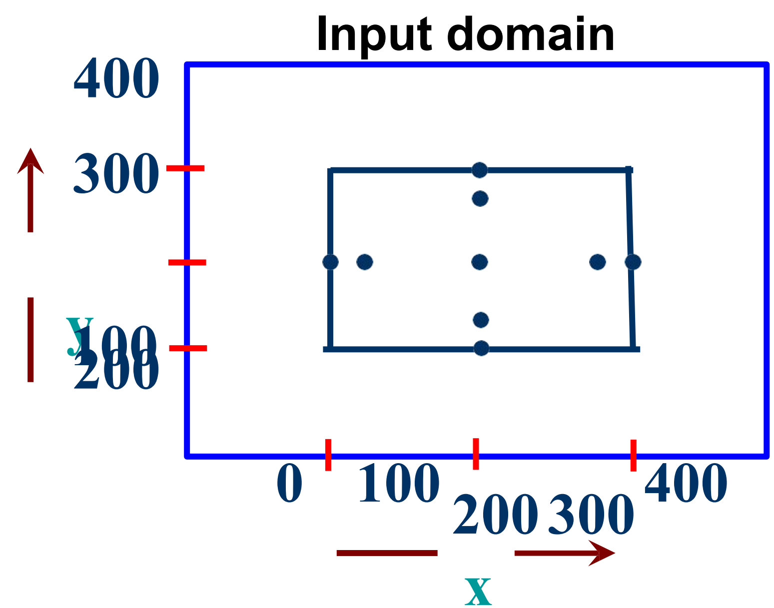


Fig. 5: Input domain of two variables x and y with boundaries [100,300] each

Software Testing

Example- I

Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values may be from interval [0,100]. The program output may have one of the following words.
[Not a quadratic equation; Real roots; Imaginary roots; Equal roots]

Design the boundary value test cases.

Software Testing

Solution

Quadratic equation will be of type:

$$ax^2+bx+c=0$$

Roots are real if $(b^2-4ac)>0$

Roots are imaginary if $(b^2-4ac)<0$

Roots are equal if $(b^2-4ac)=0$

Equation is not quadratic if $a=0$

Software Testing

The boundary value test cases are :

Test Case	*a*	*b*	*c*	*Expected output*
1	0	50	50	Not Quadratic
2	1	50	50	Real Roots
3	50	50	50	Imaginary Roots
4	99	50	50	Imaginary Roots
5	100	50	50	Imaginary Roots
6	50	0	50	Imaginary Roots
7	50	1	50	Imaginary Roots
8	50	99	50	Imaginary Roots
9	50	100	50	Equal Roots
10	50	50	0	Real Roots
11	50	50	1	Real Roots
12	50	50	99	Imaginary Roots
13	50	50	100	Imaginary Roots

Software Testing

Example – 2

Consider a program for determining the Previous date. Its input is a triple of day, month and year with the values in the range

$1 \leq month \leq 12$

$1 \leq day \leq 31$

$1900 \leq year \leq 2025$

The possible outputs would be Previous date or invalid input date. Design the boundary value test cases.

Software Testing

Solution

The Previous date program takes a date as input and checks it for validity. If valid, it returns the previous date as its output.

With single fault assumption theory, 4n+1 test cases can be designed and which are equal to 13.

Software Testing

The boundary value test cases are:

Test Case	Month	Day	Year	Expected output
1	6	15	1900	14 June, 1900
2	6	15	1901	14 June, 1901
3	6	15	1962	14 June, 1962
4	6	15	2024	14 June, 2024
5	6	15	2025	14 June, 2025
6	6	1	1962	31 May, 1962
7	6	2	1962	1 June, 1962
8	6	30	1962	29 June, 1962
9	6	31	1962	Invalid date
10	1	15	1962	14 January, 1962
11	2	15	1962	14 February, 1962
12	11	15	1962	14 November, 1962
13	12	15	1962	14 December, 1962

Software Testing

Example – 3

Consider a simple program to classify a triangle. Its inputs is a triple of positive integers (say x, y, z) and the date type for input parameters ensures that these will be integers greater than 0 and less than or equal to 100. The program output may be one of the following words:
[Scalene; Isosceles; Equilateral; Not a triangle]

Design the boundary value test cases.

Software Testing

Solution

The boundary value test cases are shown below:

Test case	x	y	z	Expected Output
1	50	50	1	Isosceles
2	50	50	2	Isosceles
3	50	50	50	Equilateral
4	50	50	99	Isosceles
5	50	50	100	Not a triangle
6	50	1	50	Isosceles
7	50	2	50	Isosceles
8	50	99	50	Isosceles
9	50	100	50	Not a triangle
10	1	50	50	Isosceles
11	2	50	50	Isosceles
12	99	50	50	Isosceles
13	100	50	50	Not a triangle

Software Testing

Robustness testing

It is nothing but the extension of boundary value analysis. Here, we would like to see, what happens when the extreme values are exceeded with a value slightly greater than the maximum, and a value slightly less than minimum. It means, we want to go outside the legitimate boundary of input domain. This extended form of boundary value analysis is called robustness testing and shown in Fig. 6

There are four additional test cases which are outside the legitimate input domain. Hence total test cases in robustness testing are 6n+1, where n is the number of input variables. So, 13 test cases are:

(200,99), (200,100), (200,101), (200,200), (200,299), (200,300)

(200,301), (99,200), (100,200), (101,200), (299,200), (300,200), (301,200)

Software Testing

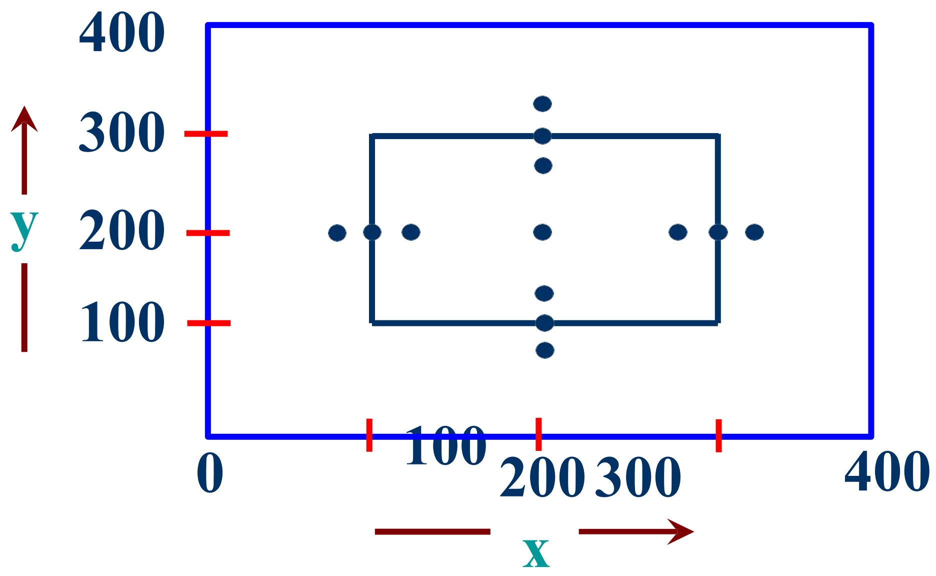


Fig. 8.6: Robustness test cases for two variables x and y with range [100,300] each

Software Testing

Worst-case testing

If we reject "single fault" assumption theory of reliability and may like to see what happens when more than one variable has an extreme value. In electronic circuits analysis, this is called "worst case analysis". It is more thorough in the sense that boundary value test cases are a proper subset of worst case test cases. It requires more effort. Worst case testing for a function of *n* variables generate 5^n test cases as opposed to $4n+1$ test cases for boundary value analysis. Our two variables example will have $5^2=25$ test cases and are given in table 1.

Software Testing

Table 1: Worst cases test inputs for two variables example

Test case number	Inputs		Test case number	Inputs	
	x	***y***		***x***	***y***
1	100	100	14	200	299
2	100	101	15	200	300
3	100	200	16	299	100
4	100	299	17	299	101
5	100	300	18	299	200
6	101	100	19	299	299
7	101	101	20	299	300
8	101	200	21	300	100
9	101	299	22	300	101
10	101	300	23	300	200
11	200	100	24	300	299
12	200	101	25	300	300
13	200	200	--		

Software Testing

Example - 8.4

Consider the program for the determination of nature of roots of a quadratic equation as explained in example 8.1. Design the Robust test case and worst test cases for this program.

Solution

Robust test cases are 6n+1. Hence, in 3 variable input cases total number of test cases are 19 as given on next slide:

Software Testing

Test case	a	b	c	Expected Output
1	-1	50	50	Invalid input`
2	0	50	50	Not quadratic equation
3	1	50	50	Real roots
4	50	50	50	Imaginary roots
5	99	50	50	Imaginary roots
6	100	50	50	Imaginary roots
7	101	50	50	Invalid input
8	50	-1	50	Invalid input
9	50	0	50	Imaginary roots
10	50	1	50	Imaginary roots
11	50	99	50	Imaginary roots
12	50	100	50	Equal roots
13	50	101	50	Invalid input
14	50	50	-1	Invalid input
15	50	50	0	Real roots
16	50	50	1	Real roots
17	50	50	99	Imaginary roots
18	50	50	100	Imaginary roots
19	50	50	101	Invalid input

Software Testing

In case of worst test case total test cases are 5^n. Hence, 125 test cases will be generated in worst test cases. The worst test cases are given below:

Test Case	***a***	***b***	***c***	***Expected output***
1	0	0	0	Not Quadratic
2	0	0	1	Not Quadratic
3	0	0	50	Not Quadratic
4	0	0	99	Not Quadratic
5	0	0	100	Not Quadratic
6	0	1	0	Not Quadratic
7	0	1	1	Not Quadratic
8	0	1	50	Not Quadratic
9	0	1	99	Not Quadratic
10	0	1	100	Not Quadratic
11	0	50	0	Not Quadratic
12	0	50	1	Not Quadratic
13	0	50	50	Not Quadratic
14	0	50	99	Not Quadratic

(Contd.)...

Software Testing

Test Case	*A*	*b*	*c*	*Expected output*
15	0	50	100	Not Quadratic
16	0	99	0	Not Quadratic
17	0	99	1	Not Quadratic
18	0	99	50	Not Quadratic
19	0	99	99	Not Quadratic
20	0	99	100	Not Quadratic
21	0	100	0	Not Quadratic
22	0	100	1	Not Quadratic
23	0	100	50	Not Quadratic
24	0	100	99	Not Quadratic
25	0	100	100	Not Quadratic
26	1	0	0	Equal Roots
27	1	0	1	Imaginary
28	1	0	50	Imaginary
29	1	0	99	Imaginary
30	1	0	100	Imaginary
31	1	1	0	Real Roots

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
32	1	1	1	Imaginary
33	1	1	50	Imaginary
34	1	1	99	Imaginary
35	1	1	100	Imaginary
36	1	50	0	Real Roots
37	1	50	1	Real Roots
38	1	50	50	Real Roots
39	1	50	99	Real Roots
40	1	50	100	Real Roots
41	1	99	0	Real Roots
42	1	99	1	Real Roots
43	1	99	50	Real Roots
44`	1	99	99	Real Roots
45	1	99	100	Real Roots
46	1	100	0	Real Roots
47	1	100	1	Real Roots
48	1	100	50	Real Roots

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
49	1	100	99	Real Roots
50	1	100	100	Real Roots
51	50	0	0	Equal Roots
52	50	0	1	Imaginary
53	50	0	50	Imaginary
54	50	0	99	Imaginary
55	50	0	100	Imaginary
56	50	1	0	Real Roots
57	50	1	1	Imaginary
58	50	1	50	Imaginary
59	50	1	99	Imaginary
60	50	1	100	Imaginary
61	50	50	0	Real Roots
62	50	50	1	Real Roots
63	50	50	50	Imaginary
64	50	50	99	Imaginary
65	50	50	100	Imaginary

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
66	50	99	0	Real Roots
67	50	99	1	Real Roots
68	50	99	50	Imaginary
69	50	99	99	Imaginary
70	50	99	100	Imaginary
71	50	100	0	Real Roots
72	50	100	1	Real Roots
73	50	100	50	Equal Roots
74	50	100	99	Imaginary
75	50	100	100	Imaginary
76	99	0	0	Equal Roots
77	99	0	1	Imaginary
78	99	0	50	Imaginary
79	99	0	99	Imaginary
80	99	0	100	Imaginary
81	99	1	0	Real Roots
82	99	1	1	Imaginary

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
83	99	1	50	Imaginary
84	99	1	99	Imaginary
85	99	1	100	Imaginary
86	99	50	0	Real Roots
87	99	50	1	Real Roots
88	99	50	50	Imaginary
89	99	50	99	Imaginary
90	99	50	100	Imaginary
91	99	99	0	Real Roots
92	99	99	1	Real Roots
93	99	99	50	Imaginary Roots
94	99	99	99	Imaginary
95	99	99	100	Imaginary
96	99	100	0	Real Roots
97	99	100	1	Real Roots
98	99	100	50	Imaginary
99	99	100	99	Imaginary
100	99	100	100	Imaginary

(Contd.)...

Software Testing

Test Case	A	b	C	Expected output
101	100	0	0	Equal Roots
102	100	0	1	Imaginary
103	100	0	50	Imaginary
104	100	0	99	Imaginary
105	100	0	100	Imaginary
106	100	1	0	Real Roots
107	100	1	1	Imaginary
108	100	1	50	Imaginary
109	100	1	99	Imaginary
110	100	1	100	Imaginary
111	100	50	0	Real Roots
112	100	50	1	Real Roots
113	100	50	50	Imaginary
114	100	50	99	Imaginary
115	100	50	100	Imaginary
116	100	99	0	Real Roots
117	100	99	1	Real Roots
118	100	99	50	Imaginary

(Contd.)...

Software Testing

Test Case	***A***	***b***	***C***	***Expected output***
119	100	99	99	Imaginary
120	100	99	100	Imaginary
121	100	100	0	Real Roots
122	100	100	1	Real Roots
123	100	100	50	Imaginary
124	100	100	99	Imaginary
125	100	100	100	Imaginary

Software Testing

Example – 5

Consider the program for the determination of previous date in a calendar as explained in example 8.2. Design the robust and worst test cases for this program.

Software Testing

Solution

Robust test cases are $6n+1$. Hence total 19 robust test cases are designed and are given on next slide.

Software Testing

Test case	Month	Day	Year	Expected Output
1	6	15	1899	Invalid date (outside range)
2	6	15	1900	14 June, 1900
3	6	15	1901	14 June, 1901
4	6	15	1962	14 June, 1962
5	6	15	2024	14 June, 2024
6	6	15	2025	14 June, 2025
7	6	15	2026	Invalid date (outside range)
8	6	0	1962	Invalid date
9	6	1	1962	31 May, 1962
10	6	2	1962	1 June, 1962
11	6	30	1962	29 June, 1962
12	6	31	1962	Invalid date
13	6	32	1962	Invalid date
14	0	15	1962	Invalid date
15	1	15	1962	14 January, 1962
16	2	15	1962	14 February, 1962
17	11	15	1962	14 November, 1962
18	12	15	1962	14 December, 1962
19	13	15	1962	Invalid date

Software Testing

In case of worst test case total test cases are 5^n. Hence, 125 test cases will be generated in worst test cases. The worst test cases are given below:

Test Case	***Month***	***Day***	***Year***	***Expected output***
1	1	1	1900	31 December, 1899
2	1	1	1901	31 December, 1900
3	1	1	1962	31 December, 1961
4	1	1	2024	31 December, 2023
5	1	1	2025	31 December, 2024
6	1	2	1900	1 January, 1900
7	1	2	1901	1 January, 1901
8	1	2	1962	1 January, 1962
9	1	2	2024	1 January, 2024
10	1	2	2025	1 January, 2025
11	1	15	1900	14 January, 1900
12	1	15	1901	14 January, 1901
13	1	15	1962	14 January, 1962
14	1	15	2024	14 January, 2024

(Contd.)...

Software Testing

Test Case	A	b	c	Expected output
15	1	15	2025	14 January, 2025
16	1	30	1900	29 January, 1900
17	1	30	1901	29 January, 1901
18	1	30	1962	29 January, 1962
19	1	30	2024	29 January, 2024
20	1	30	2025	29 January, 2025
21	1	31	1900	30 January, 1900
22	1	31	1901	30 January, 1901
23	1	31	1962	30 January, 1962
24	1	31	2024	30 January, 2024
25	1	31	2025	30 January, 2025
26	2	1	1900	31 January, 1900
27	2	1	1901	31 January, 1901
28	2	1	1962	31 January, 1962
29	2	1	2024	31 January, 2024
30	2	1	2025	31 January, 2025
31	2	2	1900	1 February, 1900

(Contd.)...

Software Testing

Test Case	Month	Day	Year	Expected output
32	2	2	1901	1 February, 1901
33	2	2	1962	1 February, 1962
34	2	2	2024	1 February, 2024
35	2	2	2025	1 February, 2025
36	2	15	1900	14 February, 1900
37	2	15	1901	14 February, 1901
38	2	15	1962	14 February, 1962
39	2	15	2024	14 February, 2024
40	2	15	2025	14 February, 2025
41	2	30	1900	Invalid date
42	2	30	1901	Invalid date
43	2	30	1962	Invalid date
44	2	30	2024	Invalid date
45	2	30	2025	Invalid date
46	2	31	1900	Invalid date
47	2	31	1901	Invalid date
48	2	31	1962	Invalid date

(Contd.)...

Software Testing

Test Case	*Month*	*Day*	*Year*	*Expected output*
49	2	31	2024	Invalid date
50	2	31	2025	Invalid date
51	6	1	1900	31 May, 1900
52	6	1	1901	31 May, 1901
53	6	1	1962	31 May, 1962
54	6	1	2024	31 May, 2024
55	6	1	2025	31 May, 2025
56	6	2	1900	1 June, 1900
57	6	2	1901	1 June, 1901
58	6	2	1962	1 June, 1962
59	6	2	2024	1 June, 2024
60	6	2	2025	1 June, 2025
61	6	15	1900	14 June, 1900
62	6	15	1901	14 June, 1901
63	6	15	1962	14 June, 1962
64	6	15	2024	14 June, 2024
65	6	15	2025	14 June, 2025

(Contd.)...

Software Testing

Test Case	Month	Day	Year	Expected output
66	6	30	1900	29 June, 1900
67	6	30	1901	29 June, 1901
68	6	30	1962	29 June, 1962
69	6	30	2024	29 June, 2024
70	6	30	2025	29 June, 2025
71	6	31	1900	Invalid date
72	6	31	1901	Invalid date
73	6	31	1962	Invalid date
74	6	31	2024	Invalid date
75	6	31	2025	Invalid date
76	11	1	1900	31 October, 1900
77	11	1	1901	31 October, 1901
78	11	1	1962	31 October, 1962
79	11	1	2024	31 October, 2024
80	11	1	2025	31 October, 2025
81	11	2	1900	1 November, 1900
82	11	2	1901	1 November, 1901

(Contd.)...

Software Testing

Test Case	Month	Day	Year	Expected output
83	11	2	1962	1 November, 1962
84	11	2	2024	1 November, 2024
85	11	2	2025	1 November, 2025
86	11	15	1900	14 November, 1900
87	11	15	1901	14 November, 1901
88	11	15	1962	14 November, 1962
89	11	15	2024	14 November, 2024
90	11	15	2025	14 November, 2025
91	11	30	1900	29 November, 1900
92	11	30	1901	29 November, 1901
93	11	30	1962	29 November, 1962
94	11	30	2024	29 November, 2024
95	11	30	2025	29 November, 2025
96	11	31	1900	Invalid date
97	11	31	1901	Invalid date
98	11	31	1962	Invalid date
99	11	31	2024	Invalid date
100	11	31	2025	Invalid date

(Contd.)...

Software Testing

Test Case	Month	Day	Year	Expected output
101	12	1	1900	30 November, 1900
102	12	1	1901	30 November, 1901
103	12	1	1962	30 November, 1962
104	12	1	2024	30 November, 2024
105	12	1	2025	30 November, 2025
106	12	2	1900	1 December, 1900
107	12	2	1901	1 December, 1901
108	12	2	1962	1 December, 1962
109	12	2	2024	1 December, 2024
110	12	2	2025	1 December, 2025
111	12	15	1900	14 December, 1900
112	12	15	1901	14 December, 1901
113	12	15	1962	14 December, 1962
114	12	15	2024	14 December, 2024
115	12	15	2025	14 December, 2025
116	12	30	1900	29 December, 1900
117	12	30	1901	29 December, 1901
118	12	30	1962	29 December, 1962

(Contd.)...

Software Testing

Test Case	***Month***	***Day***	***Year***	***Expected output***
119	12	30	2024	29 December, 2024
120	12	30	2025	29 December, 2025
121	12	31	1900	30 December, 1900
122	12	31	1901	30 December, 1901
123	12	31	1962	30 December, 1962
124	12	31	2024	30 December, 2024
125	12	31	2025	30 December, 2025

Software Testing

Example – 6

Consider the triangle problem. Generate robust and worst test cases for this problem.

Software Testing

Solution

Robust test cases are given on next slide.

Software Testing

`	x	y	z	*Expected Output*
1	50	50	0	Invalid input`
2	50	50	1	Isosceles
3	50	50	2	Isosceles
4	50	50	50	Equilateral
5	50	50	99	Isosceles
6	50	50	100	Not a triangle
7	50	50	101	Invalid input
8	50	0	50	Invalid input
9	50	1	50	Isosceles
10	50	2	50	Isosceles
11	50	99	50	Isosceles
12	50	100	50	Not a triangle
13	50	101	50	Invalid input
14	0	50	50	Invalid input
15	1	50	50	Isosceles
16	2	50	50	Isosceles
17	99	50	50	Isosceles
18	100	50	50	Not a triangle
19	100	50	50	Invalid input

Software Testing

Worst test cases are 125 and are given below:

Test Case	x	y	z	Expected output
1	1	1	1	Equilateral
2	1	1	2	Not a triangle
3	1	1	50	Not a triangle
4	1	1	99	Not a triangle
5	1	1	100	Not a triangle
6	1	2	1	Not a triangle
7	1	2	2	Isosceles
8	1	2	50	Not a triangle
9	1	2	99	Not a triangle
10	1	2	100	Not a triangle
11	1	50	1	Not a triangle
12	1	50	2	Not a triangle
13	1	50	50	Isosceles
14	1	50	99	Not a triangle

(Contd.)...

Test Case	***A***	***b***	***c***	***Expected output***
15	1	50	100	Not a triangle
16	1	99	1	Not a triangle
17	1	99	2	Not a triangle
18	1	99	50	Not a triangle
19	1	99	99	Isosceles
20	1	99	100	Not a triangle
21	1	100	1	Not a triangle
22	1	100	2	Not a triangle
23	1	100	50	Not a triangle
24	1	100	99	Not a triangle
25	1	100	100	Isosceles
26	2	1	1	Not a triangle
27	2	1	2	Isosceles
28	2	1	50	Not a triangle
29	2	1	99	Not a triangle
30	2	1	100	Not a triangle
31	2	2	1	Isosceles

(Contd.)...

Test Case	A	b	C	Expected output
32	2	2	2	Equilateral
33	2	2	50	Not a triangle
34	2	2	99	Not a triangle
35	2	2	100	Not a triangle
36	2	50	1	Not a triangle
37	2	50	2	Not a triangle
38	2	50	50	Isosceles
39	2	50	99	Not a triangle
40	2	50	100	Not a triangle
41	2	99	1	Not a triangle
42	2	99	2	Not a triangle
43	2	99	50	Not a triangle
44	2	99	99	Isosceles
45	2	99	100	Scalene
46	2	100	1	Not a triangle
47	2	100	2	Not a triangle
48	2	100	50	Not a triangle

(Contd.)...

Test Case	*A*	*b*	*C*	*Expected output*
49	2	100	50	Scalene
50	2	100	99	Isosceles
51	50	1	100	Not a triangle
52	50	1	1	Not a triangle
53	50	1	2	Isosceles
54	50	1	50	Not a triangle
55	50	1	99	Not a triangle
56	50	2	100	Not a triangle
57	50	2	1	Not a triangle
58	50	2	2	Isosceles
59	50	2	50	Not a triangle
60	50	2	99	Not a triangle
61	50	50	100	Isosceles
62	50	50	1	Isosceles
63	50	50	2	Equilateral
64	50	50	50	Isosceles
65	50	50	99	Not a triangle

(Contd.)...

Test Case	A	B	C	Expected output
66	50	99	1	Not a triangle
67	50	99	2	Not a triangle
68	50	99	50	Isosceles
69	50	99	99	Isosceles
70	50	99	100	Scalene
71	50	100	1	Not a triangle
72	50	100	2	Not a triangle
73	50	100	50	Not a triangle
74	50	100	99	Scalene
75	50	100	100	Isosceles
76	50	1	1	Not a triangle
77	99	1	2	Not a triangle
78	99	1	50	Not a triangle
79	99	1	99	Isosceles
80	99	1	100	Not a triangle
81	99	2	1	Not a triangle
82	99	2	2	Not a triangle

(Contd.)...

Test Case	***A***	***b***	***C***	***Expected output***
83	99	2	50	Not a triangle
84	99	2	99	Isosceles
85	99	2	100	Scalene
86	99	50	1	Not a triangle
87	99	50	2	Not a triangle
88	99	50	50	Isosceles
89	99	50	99	Isosceles
90	99	50	100	Scalene
91	99	99	1	Isosceles
92	99	99	2	Isosceles
93	99	99	50	Isosceles
94	99	99	99	Equilateral
95	99	99	100	Isosceles
96	99	100	1	Not a triangle
97	99	100	2	Scalene
98	99	100	50	Scalene
99	99	100	99	Isosceles
100	99	100	100	Isosceles

(Contd.)...

Test Case	***A***	***b***	***C***	***Expected output***
101	100	1	1	Not a triangle
102	100	1	2	Not a triangle
103	100	1	50	Not a triangle
104	100	1	99	Not a triangle
105	100	1	100	Isosceles
106	100	2	1	Not a triangle
107	100	2	2	Not a triangle
108	100	2	50	Not a triangle
109	100	2	99	Scalene
110	100	2	100	Isosceles
111	100	50	1	Not a triangle
112	100	50	2	Not a triangle
113	100	50	50	Not a triangle
114	100	50	99	Scalene
115	100	50	100	Isosceles
116	100	99	1	Not a triangle
117	100	99	2	Scalene
118	100	99	50	Scalene

(Contd.)...

Test Case	***A***	***b***	***C***	***Expected output***
119	100	99	99	Isosceles
120	100	99	100	Isosceles
121	100	100	1	Isosceles
122	100	100	2	Isosceles
123	100	100	50	Isosceles
124	100	100	99	Isosceles
125	100	100	100	Equilateral

Software Testing

Equivalence Class Testing

In this method, input domain of a program is partitioned into a finite number of equivalence classes such that one can reasonably assume, but not be absolutely sure, that the test of a representative value of each class is equivalent to a test of any other value.

Two steps are required to implementing this method:

1. The equivalence classes are identified by taking each input condition and partitioning it into valid and invalid classes. For example, if an input condition specifies a range of values from 1 to 999, we identify one valid equivalence class [1<item<999]; and two invalid equivalence classes [item<1] and [item>999].

2. Generate the test cases using the equivalence classes identified in the previous step. This is performed by writing test cases covering all the valid equivalence classes. Then a test case is written for each invalid equivalence class so that no test contains more than one invalid class. This is to ensure that no two invalid classes mask each other.

Software Testing

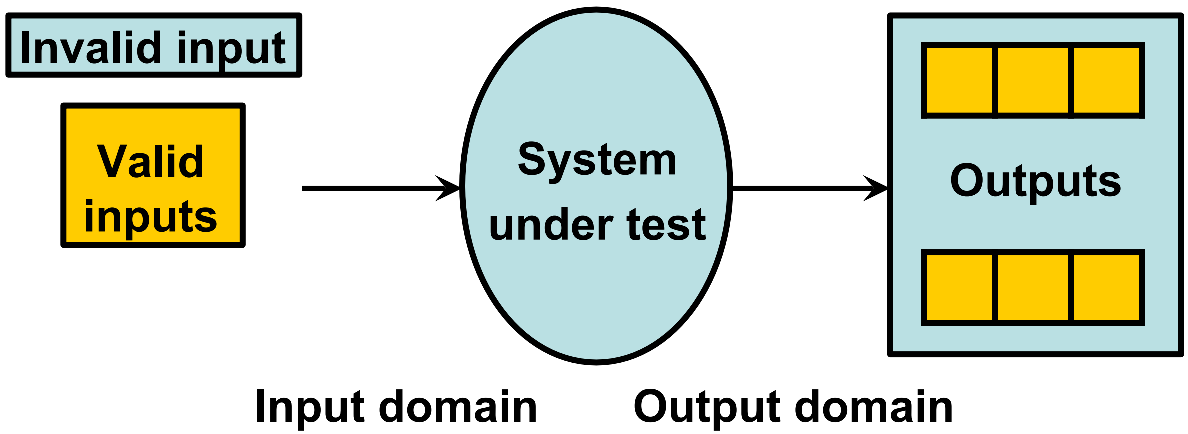


Fig. 7: Equivalence partitioning

Most of the time, equivalence class testing defines classes of the input domain. However, equivalence classes should also be defined for output domain. Hence, we should design equivalence classes based on input and output domain.

Software Testing

Example 7

Consider the program for the determination of nature of roots of a quadratic equation. Identify the equivalence class test cases for output and input domains.

Software Testing

Solution

Output domain equivalence class test cases can be identified as follows:

$O_1=\{<a,b,c>:\text{Not a quadratic equation if } a = 0\}$

$O_1=\{<a,b,c>:\text{Real roots if } (b^2-4ac)>0\}$

$O_1=\{<a,b,c>:\text{Imaginary roots if } (b^2-4ac)<0\}$

$O_1=\{<a,b,c>:\text{Equal roots if } (b^2-4ac)=0\}$`

The number of test cases can be derived form above relations and shown below:

Test case	***a***	***b***	***c***	***Expected output***
1	0	50	50	Not a quadratic equation
2	1	50	50	Real roots
3	50	50	50	Imaginary roots
4	50	100	50	Equal roots

Software Testing

We may have another set of test cases based on input domain.

$I_1 = \{a: a = 0\}$

$I_2 = \{a: a < 0\}$

$I_3 = \{a: 1 \leq a \leq 100\}$

$I_4 = \{a: a > 100\}$

$I_5 = \{b: 0 \leq b \leq 100\}$

$I_6 = \{b: b < 0\}$

$I_7 = \{b: b > 100\}$

$I_8 = \{c: 0 \leq c \leq 100\}$

$I_9 = \{c: c < 0\}$

$I_{10} = \{c: c > 100\}$

Software Testing

Test Case	a	b	c	Expected output
1	0	50	50	Not a quadratic equation
2	-1	50	50	Invalid input
3	50	50	50	Imaginary Roots
4	101	50	50	invalid input
5	50	50	50	Imaginary Roots
6	50	-1	50	invalid input
7	50	101	50	invalid input
8	50	50	50	Imaginary Roots
9	50	50	-1	invalid input
10	50	50	101	invalid input

Here test cases 5 and 8 are redundant test cases. If we choose any value other than nominal, we may not have redundant test cases. Hence total test cases are 10+4=14 for this problem.

Example 8

Consider the program for determining the previous date in a calendar as explained in example 3. Identify the equivalence class test cases for output & input domains.

Software Testing

Solution

Output domain equivalence class are:

O_1={<D,M,Y>: Previous date if all are valid inputs}

O_1={<D,M,Y>: Invalid date if any input makes the date invalid}

Test case	***M***	***D***	***Y***	***Expected output***
1	6	15	1962	14 June, 1962
2	6	31	1962	Invalid date

Software Testing

We may have another set of test cases which are based on input domain.

I_1={month: $1 \le m \le 12$}

I_2={month: $m < 1$}

I_3={month: $m > 12$}

I_4={day: $1 \le D \le 31$}

I_5={day: $D < 1$}

I_6={day: $D > 31$}

I_7={year: $1900 \le Y \le 2025$}

I_8={year: $Y < 1900$}

I_9={year: $Y > 2025$}

Software Testing

Inputs domain test cases are :

Test Case	*M*	*D*	*Y*	*Expected output*
1	6	15	1962	14 June, 1962
2	-1	15	1962	Invalid input
3	13	15	1962	invalid input
4	6	15	1962	14 June, 1962
5	6	-1	1962	invalid input
6	6	32	1962	invalid input
7	6	15	1962	14 June, 1962
8	6	15	1899	invalid input (Value out of range)
9	6	15	2026	invalid input (Value out of range)

Example – 9

Consider the triangle problem specified in a example. Identify the equivalence class test cases for output and input domain.

Software Testing

Solution

Output domain equivalence classes are:

O_1={<*x,y,z*>: Equilateral triangle with sides *x,y,z*}
O_1={<*x,y,z*>: Isosceles triangle with sides *x,y,z*}
O_1={<*x,y,z*>: Scalene triangle with sides *x,y,z*}
O_1={<*x,y,z*>: Not a triangle with sides *x,y,z*}

The test cases are:

Test case	*x*	*y*	*z*	*Expected Output*
1	50	50	50	Equilateral
2	50	50	99	Isosceles
3	100	99	50	Scalene
4	50	100	50	Not a triangle

Software Testing

Input domain based classes are:

$I_1=\{x: x < 1\}$

$I_2=\{x: x > 100\}$

$I_3=\{x: 1 \leq x \leq 100\}$

$I_4=\{y: y < 1\}$

$I_5=\{y: y > 100\}$

$I_6=\{y: 1 \leq y \leq 100\}$

$I_7=\{z: z < 1\}$

$I_8=\{z: z > 100\}$

$I_9=\{z: 1 \leq z \leq 100\}$

Software Testing

Some inputs domain test cases can be obtained using the relationship amongst x,y and z.

$I_{10}=\{< x,y,z >: x = y = z\}$

$I_{11}=\{< x,y,z >: x = y, x \neq z\}$

$I_{12}=\{< x,y,z >: x = z, x \neq y\}$

$I_{13}=\{< x,y,z >: y = z, x \neq y\}$

$I_{14}=\{< x,y,z >: x \neq y, x \neq z, y \neq z\}$

$I_{15}=\{< x,y,z >: x = y + z\}$

$I_{16}=\{< x,y,z >: x > y +z\}$

$I_{17}=\{< x,y,z >: y = x +z\}$

$I_{18}=\{< x,y,z >: y > x + z\}$

$I_{19}=\{< x,y,z >: z = x + y\}$

$I_{20}=\{< x,y,z >: z > x +y\}$

Software Testing

Test cases derived from input domain are:

Test case	*x*	*y*	*z*	*Expected Output*
1	0	50	50	Invalid input
2	101	50	50	Invalid input
3	50	50	50	Equilateral
4	50	0	50	Invalid input
5	50	101	50	Invalid input
6	50	50	50	Equilateral
7	50	50	0	Invalid input
8	50	50	101	Invalid input
9	50	50	50	Equilateral
10	60	60	60	Equilateral
11	50	50	60	Isosceles
12	50	60	50	Isosceles
13	60	50	50	Isosceles

(Contd.)...

Software Testing

Test case	*x*	*y*	*z*	*Expected Output*
14	100	99	50	Scalene
15	100	50	50	Not a triangle
16	100	50	25	Not a triangle
17	50	100	50	Not a triangle
18	50	100	25	Not a triangle
19	50	50	100	Not a triangle
20	25	50	100	Not a triangle

Software Testing

Decision Table Based Testing

Condition Stub		Entry						
	c_1	True				False		
	c_2	True		False		True		False
	c_3	True	False	True	False	True	False	---
Action Stub	a_1	X	X			X		
	a_2	X		X			X	
	a_3		X			X		
	a_4				X		X	X

Table 2: Decision table terminology

Software Testing

Test case design

C_1:x,y,z are sides of a triangle?	N	Y							
C_2:x = y?	--	Y				N			
C_3:x = z?	--	Y		N		Y		N	
C_4:y = z?	--	Y	N	Y	N	Y	N	Y	N
a_1: Not a triangle	X								
a_2: Scalene									X
a_3: Isosceles					X		X	X	
a_4: Equilateral		X							
a_5: Impossible			X	X		X			

Table 3: Decision table for triangle problem

Software Testing

Conditions **C_1 : x < y + z ?**	F	T	T	T	T	T	T	T	T	T	T
C_2 : y < x + z ?	--	F	T	T	T	T	T	T	T	T	T
C_3 : z < x + y ?	--	--	F	T	T	T	T	T	T	T	T
C_4 : x = y ?	--	--	--	T	T	T	T	F	F	F	F
C_5 : x = z ?	--	--	--	T	T	F	F	T	T	F	F
C_6 : y = z ?	--	--	--	T	F	T	F	T	F	T	F
a_1 : Not a triangle	X	X	X								
a_2 : Scalene											X
a_3 : Isosceles							X		X	X	
a_4 : Equilateral				X							
a_5 : Impossible					X	X		X			

Table 4: Modified decision table

Software Testing

Example 10

Consider the triangle program specified in example 3. Identify the test cases using the decision table of Table 4.

Solution

There are eleven functional test cases, three to fail triangle property, three impossible cases, one each to get equilateral, scalene triangle cases, and three to get on isosceles triangle. The test cases are given in Table 5.

Test case	***x***	***y***	***z***	***Expected Output***
1	4	1	2	Not a triangle
2	1	4	2	Not a triangle
3	1	2	4	Not a triangle
4	5	5	5	Equilateral
5	?	?	?	Impossible
6	?	?	?	Impossible
7	2	2	3	Isosceles
8	?	?	?	Impossible
9	2	3	2	Isosceles
10	3	2	2	Isosceles
11	3	4	5	Scalene

Test cases of triangle problem using decision table

Software Testing

Example 11

Consider a program for the determination of Previous date. Its input is a triple of day, month and year with the values in the range

$1 \leq \text{month} \leq 12$

$1 \leq \text{day} \leq 31$

$1900 \leq \text{year} \leq 2025$

The possible outputs are “Previous date” and “Invalid date”. Design the test cases using decision table based testing.

Software Testing

Solution

The input domain can be divided into following classes:

I_1= {M_1: month has 30 days}

I_2= {M_2: month has 31 days except March, August and January}

I_3= {M_3: month is March}

I_4= {M_4: month is August}

I_5= {M_5: month is January}

I_6= {M_6: month is February}

I_7= {D_1: day = 1}

I_8= {D_2: $2 \le$ day ≤ 28}

I_9= {D_3: day = 29}

I_{10}={D_4: day = 30}

I_{11}={D_5: day = 31}

I_{12}={Y_1: year is a leap year}

I_{13}={Y_2: year is a common year}

Software Testing

The decision table is given below:

Sr.No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
C_1: Months in	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_1	M_2	M_2	M_2	M_2	M_2
C_2: days in	D_1	D_1	D_2	D_2	D_3	D_3	D_4	D_4	D_5	D_5	D_1	D_1	D_2	D_2	D_3
C_3: year in	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1
a_1: Impossible									X	X					
a_2: Decrement day			X	X	X	X	X	X					X	X	X
a_3: Reset day to 31	X	X													
a_4: Reset day to 30											X	X			
a_5: Reset day to 29															
a_6: Reset day to 28															
a_7: decrement month	X	X									X	X			
a_8: Reset month to December															
a_9: Decrement year															

Software Testing

Sr.No.	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
C_1: Months in	M_2	M_2	M_2	M_2	M_2	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3	M_3
C_2: days in	D_3	D_4	D_4	D_5	D_5	D_1	D_1	D_2	D_2	D_3	D_3	D_4	D_4	D_5	D_5
C_3: year in	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2
a_1: Impossible															
a_2: Decrement day	X	X	X	X	X			X	X	X	X	X	X	X	X
a_3: Reset day to 31															
a_4: Reset day to 30															
a_5: Reset day to 29						X									
a_6: Reset day to 28							X								
a_7: decrement month						X	X								
a_8: Reset month to December															
a_9: Decrement year															

Software Testing

Sr.No.	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45
C_1: Months in	M_4	M_4	M_4	M_4	M_4	M_4	M_4	M_4	M_4	M_4	M_5	M_5	M_5	M_5	M_5
C_2: days in	D_1	D_1	D_2	D_2	D_3	D_3	D_4	D_4	D_5	D_5	D_1	D_1	D_2	D_2	D_3
C_3: year in	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1
a_1: Impossible															
a_2: Decrement day			X	X	X	X	X	X	X	X			X	X	X
a_3: Reset day to 31	X	X									X	X			
a_4: Reset day to 30															
a_5: Reset day to 29															
a_6: Reset day to 28															
a_7: decrement month	X	X													
a_8: Reset month to December											X	X			
a_9: Decrement year											X	X			

Software Testing

Sr.No.	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60
C_1: Months in	M_5	M_5	M_5	M_5	M_5	M_6	M_6	M_6	M_6	M_6	M_6	M_6	M_6	M_6	M_6
C_2: days in	D_3	D_4	D_4	D_5	D_5	D_1	D_1	D_2	D_2	D_3	D_3	D_4	D_4	D_5	D_5
C_3: year in	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Y_1	Y_2
a_1: Impossible											X	X	X	X	X
a_2: Decrement day	X	X	X	X	X			X	X	X					
a_3: Reset day to 31						X	X								
a_4: Reset day to 30															
a_5: Reset day to 29															
a_6: Reset day to 28															
a_7: decrement month						X	X								
a_8: Reset month to December															
a_9: Decrement year															

Software Testing

Test case	Month	Day	Year	Expected output
1	June	1	1964	31 May, 1964
2	June	1	1962	31 May, 1962
3	June	15	1964	14 June, 1964
4	June	15	1962	14 June, 1962
5	June	29	1964	28 June, 1964
6	June	29	1962	28 June, 1962
7	June	30	1964	29 June, 1964
8	June	30	1962	29 June, 1962
9	June	31	1964	Impossible
10	June	31	1962	Impossible
11	May	1	1964	30 April, 1964
12	May	1	1962	30 April, 1962
13	May	15	1964	14 May, 1964
14	May	15	1962	14 May, 1962
15	May	29	1964	28 May, 1964

Software Testing

Test case	Month	Day	Year	Expected output
16	May	29	1962	28 May, 1962
17	May	30	1964	29 May, 1964
18	May	30	1962	29 May, 1962
19	May	31	1964	30 May, 1964
20	May	31	1962	30 May, 1962
21	March	1	1964	29 February, 1964
22	March	1	1962	28 February, 1962
23	March	15	1964	14 March, 1964
24	March	15	1962	14 March, 1962
25	March	29	1964	28 March, 1964
26	March	29	1962	28 March, 1962
27	March	30	1964	29 March, 1964
28	March	30	1962	29 March, 1962
29	March	31	1964	30 March, 1964
30	March	31	1962	30 March, 1962

Software Testing

Test case	Month	Day	Year	Expected output
31	August	1	1964	31 July, 1962
32	August	1	1962	31 July, 1964
33	August	15	1964	14 August, 1964
34	August	15	1962	14 August, 1962
35	August	29	1964	28 August, 1964
36	August	29	1962	28 August, 1962
37	August	30	1964	29 August, 1964
38	August	30	1962	29 August, 1962
39	August	31	1964	30 August, 1964
40	August	31	1962	30 August, 1962
41	January	1	1964	31 December, 1964
42	January	1	1962	31 December, 1962
43	January	15	1964	14 January, 1964
44	January	15	1962	14 January, 1962
45	January	29	1964	28 January, 1964

Software Testing

Test case	Month	Day	Year	Expected output
46	January	29	1962	28 January, 1962
47	January	30	1964	29 January, 1964
48	January	30	1962	29 January, 1962
49	January	31	1964	30 January, 1964
50	January	31	1962	30 January, 1962
51	February	1	1964	31 January, 1964
52	February	1	1962	31 January, 1962
53	February	15	1964	14 February, 1964
54	February	15	1962	14 February, 1962
55	February	29	1964	28 February, 1964
56	February	29	1962	Impossible
57	February	30	1964	Impossible
58	February	30	1962	Impossible
59	February	31	1964	Impossible
60	February	31	1962	Impossible

Software Testing

Cause Effect Graphing Technique

- Consider single input conditions
- do not explore combinations of input circumstances

Steps

1. Causes & effects in the specifications are identified.

   A cause is a distinct input condition or an equivalence class of input conditions.

   An effect is an output condition or a system transformation.

2. The semantic content of the specification is analysed and transformed into a boolean graph linking the causes & effects.

3. Constraints are imposed

4. graph – limited entry decision table

   Each column in the table represent a test case.

5. The columns in the decision table are converted into test cases.

Software Testing

The basic notation for the graph is shown in fig. 8

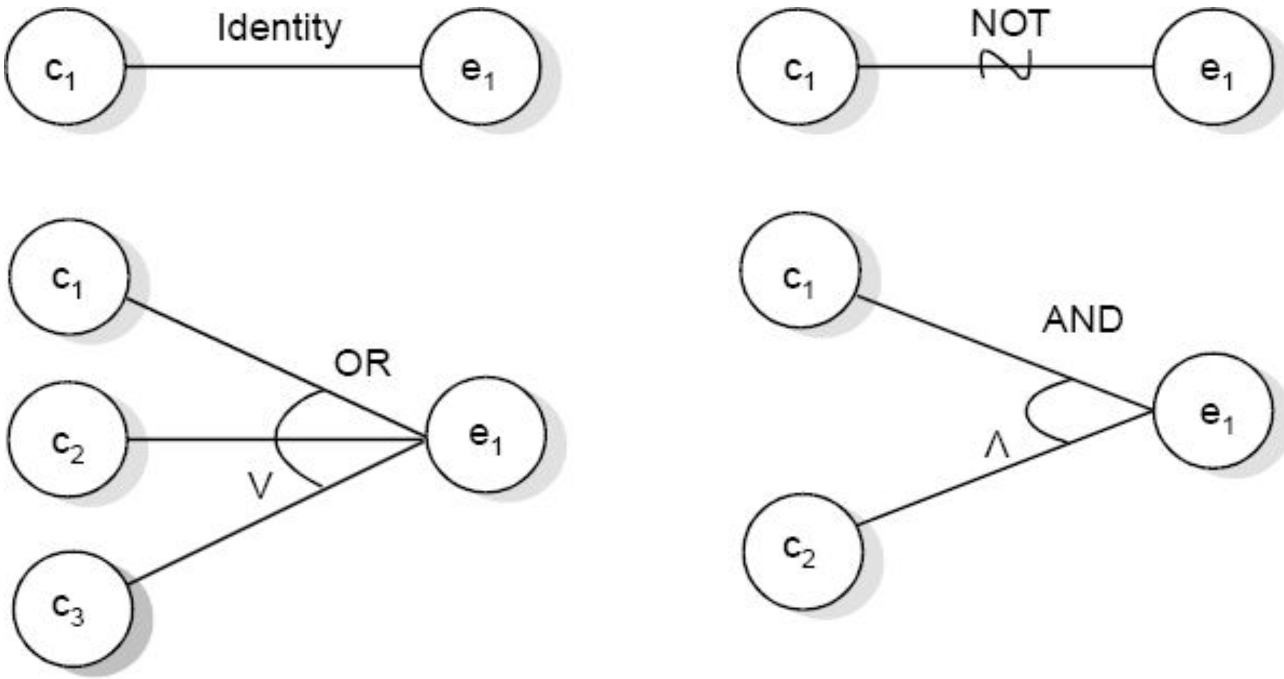


Fig.8 : Basic cause effect graph symbols

Software Testing

Myers explained this effectively with following example. “The characters in column 1 must be an A or B. The character in column 2 must be a digit. In this situation, the file update is made. If the character in column 1 is incorrect, message *x* is issued. If the character in column 2 is not a digit, message y is issued”.

The causes are

c_1: character in column 1 is A

c_2: character in column 1 is B

c_3: character in column 2 is a digit

and the effects are

e_1: update made

e_2: message *x* is issued

e_3: message *y* is issued

Software Testing

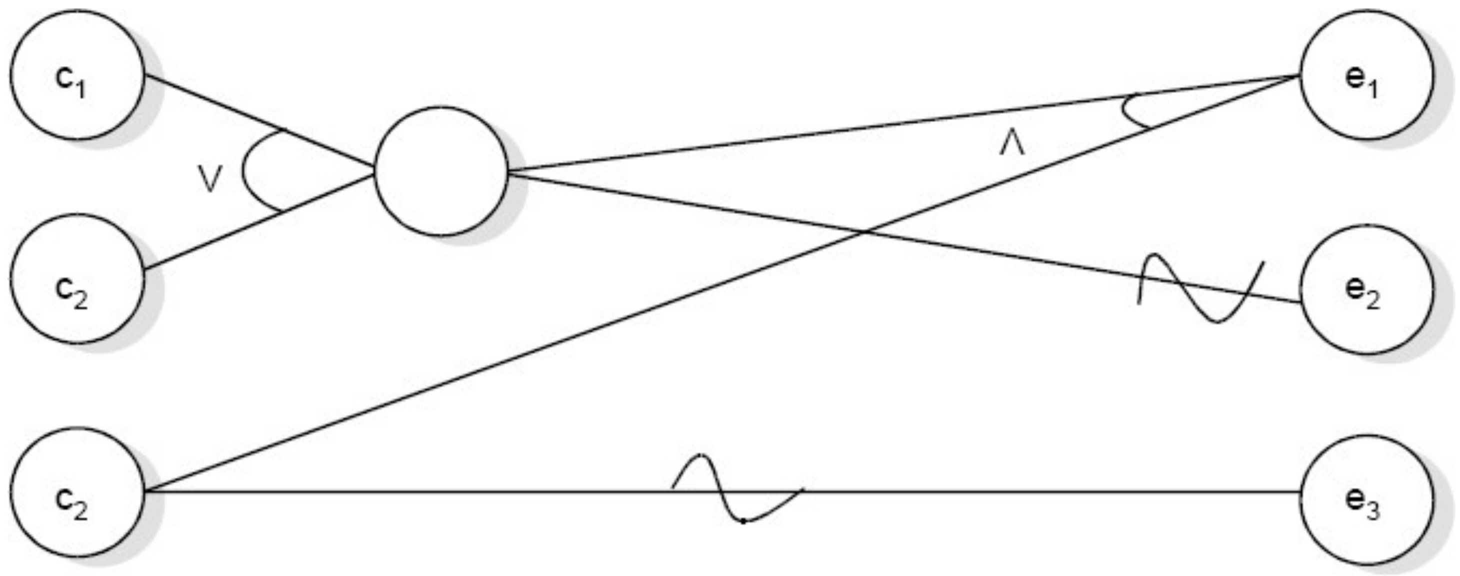


Fig. 9: Sample cause effect graph

Software Testing

The **E** constraint states that it must always be true that at most one of c_1 or c_2 can be 1 (c_1 or c_2 cannot be 1 simultaneously). The **I** constraint states that at least one of c_1, c_2 and c_3 must always be 1 (c_1, c_2 and c_3 cannot be 0 simultaneously). The **O** constraint states that one, and only one, of c_1 and c_2 must be 1. The constraint **R** states that, for c_1 to be 1, c_2 must be 1 (i.e. it is impossible for c_1 to be 1 and c_2 to be 0),

Software Testing

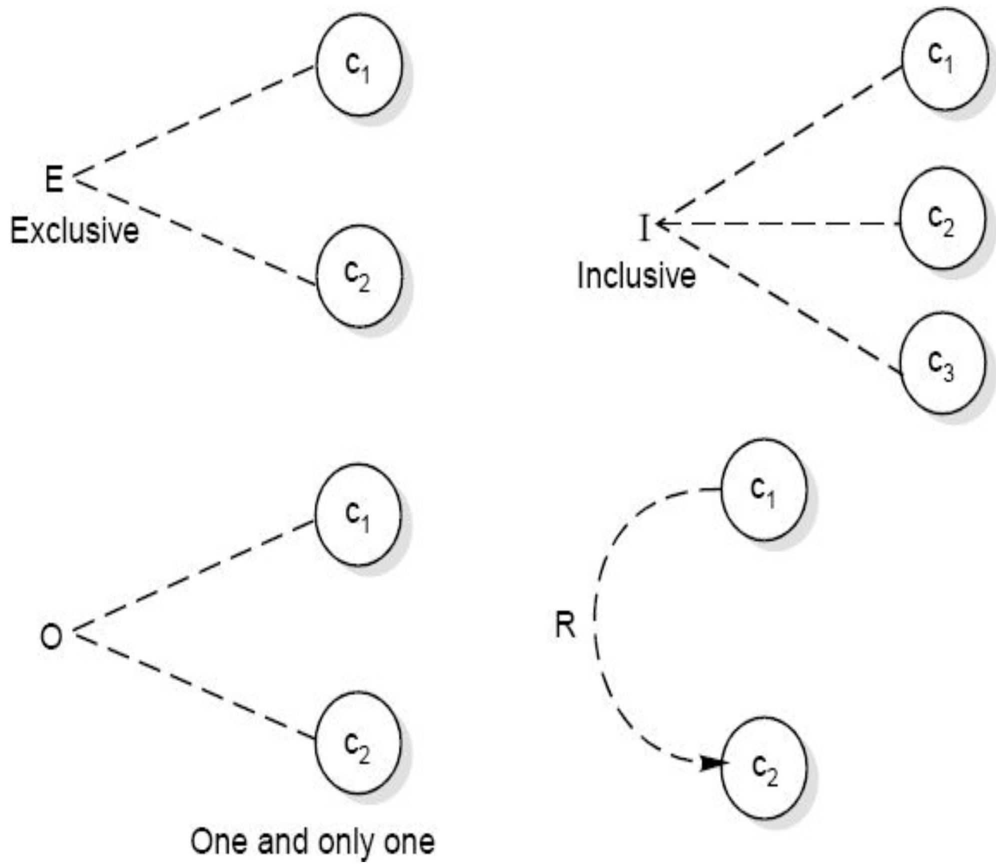


Fig. 10: Constraint symbols

Software Testing

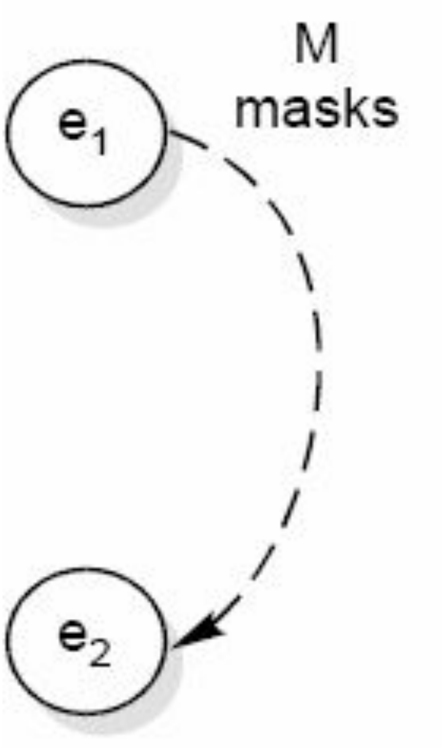


Fig. 11: Symbol for masks constraint

Software Testing

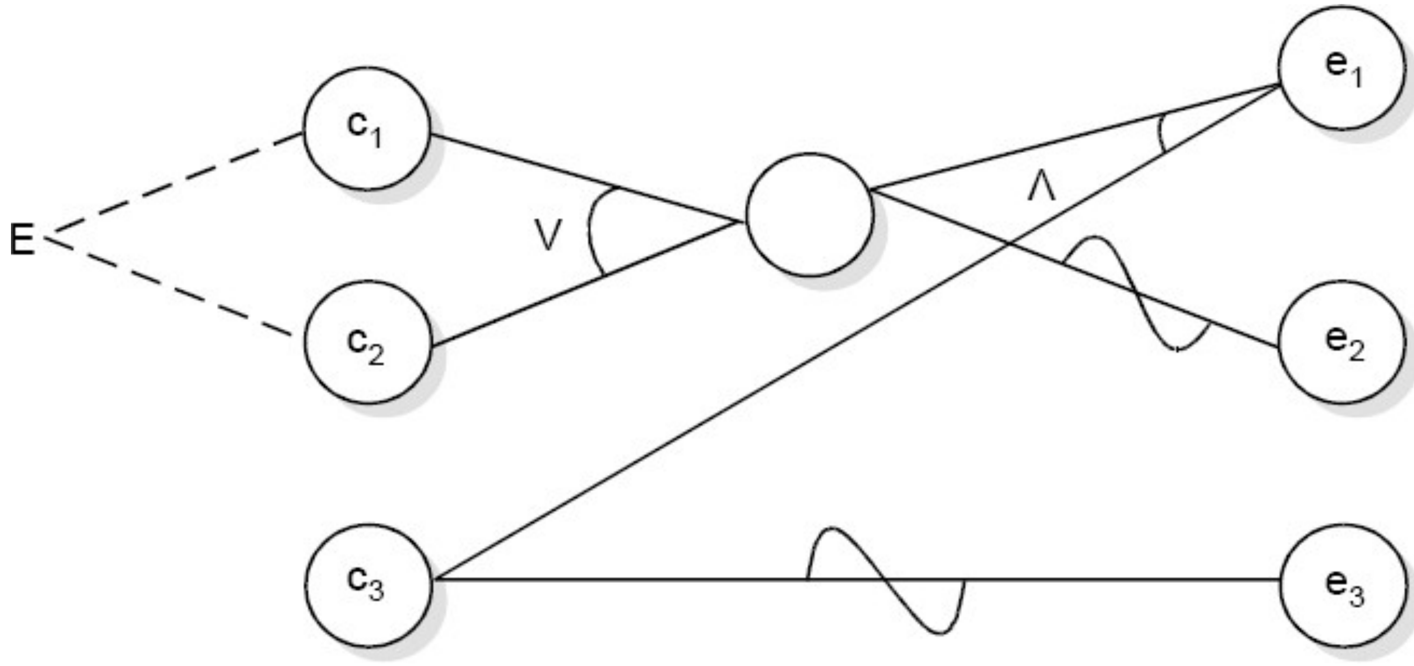


Fig. 12 : Sample cause effect graph with exclusive constraint

Software Testing

Example 12

Consider the triangle problem specified in the example 3. Draw the Cause effect graph and identify the test cases.

Software Testing

Solution

The causes are

c_1: side x is less than sum of sides y and z

c_2: side y is less than sum of sides x and y

c_3: side z is less than sum of sides x and y

c_4: side x is equal to side y

c_5: side x is equal to side z

c_6: side y is equal to side z

and effects are

e_1: Not a triangle

e_2: Scalene triangle

e_3: Isosceles triangle

e_4: Equilateral triangle

e_5: Impossible stage

Software Testing

The cause effect graph is shown in fig. 13 and decision table is shown in table 6. The test cases for this problem are available in Table 5.

Conditions											
C_1: x < y + z ?	0	1	1	1	1	1	1	1	1	1	1
C_2: y < x + z ?	X	0	1	1	1	1	1	1	1	1	1
C_3: z < x + y ?	X	X	0	1	1	1	1	1	1	1	1
C_4: x = y ?	X	X	X	1	1	1	1	0	0	0	0
C_5: x = z ?	X	X	X	1	1	0	0	1	1	0	0
C_6: y = z ?	X	X	X	1	0	1	0	1	0	1	0
e_1: Not a triangle	1	1	1								
e_2: Scalene											1
e_3: Isosceles							1		1	1	
e_4: Equilateral				1							
e_5: Impossible					1	1		1			

Table 6: Decision table

Software Testing

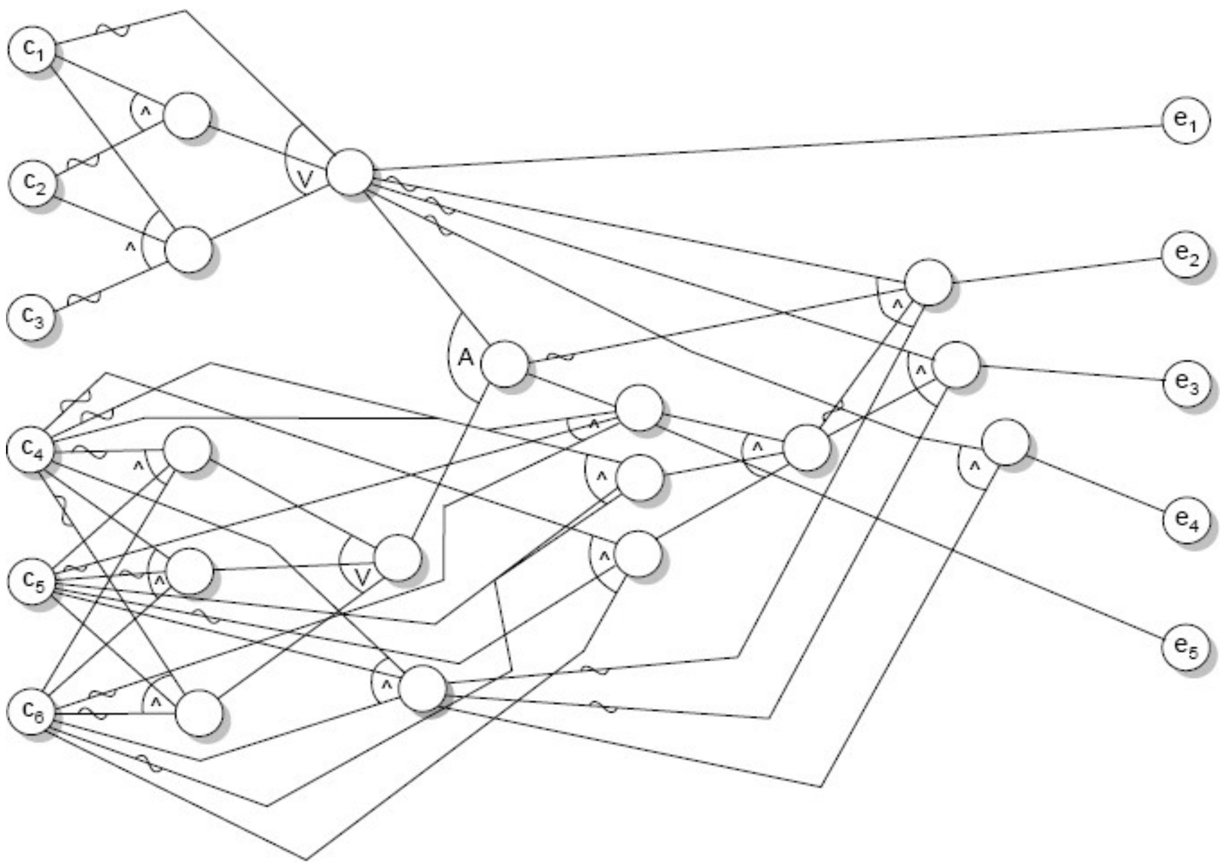


Fig. 13: Cause effect graph of triangle problem

Software Testing

Structural Testing

A complementary approach to functional testing is called structural / white box testing. It permits us to examine the internal structure of the program.

Path Testing

Path testing is the name given to a group of test techniques based on judiciously selecting a set of test paths through the program. If the set of paths is properly chosen, then it means that we have achieved some measure of test thoroughness.

This type of testing involves:

1. generating a set of paths that will cover every branch in the program.
2. finding a set of test cases that will execute every path in the set of program paths.

Software Testing

Flow Graph

The control flow of a program can be analysed using a graphical representation known as flow graph. The flow graph is a directed graph in which nodes are either entire statements or fragments of a statement, and edges represents flow of control.

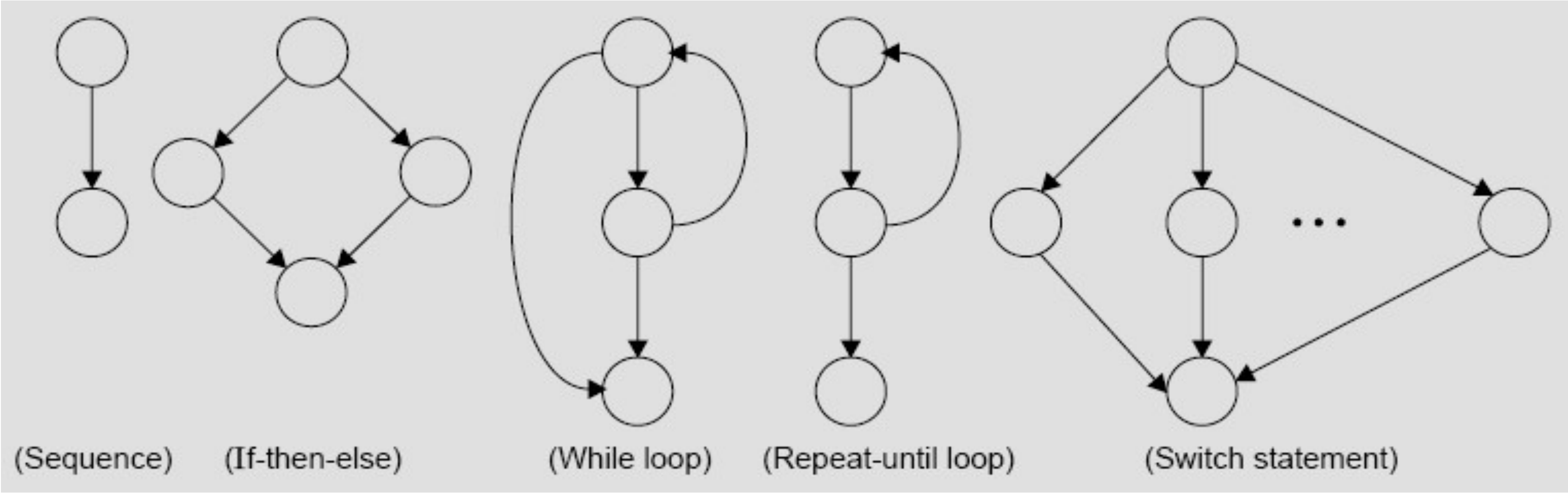
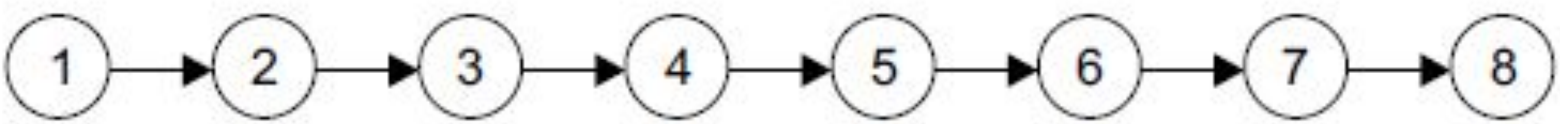
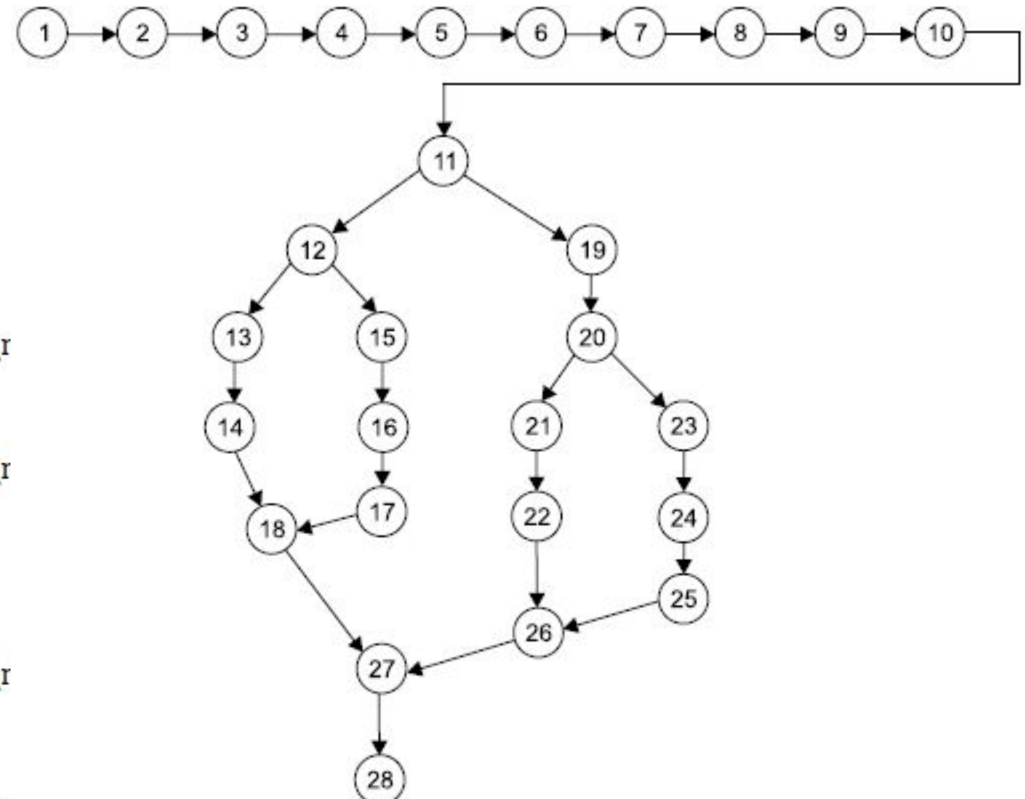


Fig. 14: The basic construct of the flow graph

```c
#include<stdio.h>
void main()
{
int num, result;
printf("Enter the number:");
scanf("%d", &num);
result=num*num;
printf("The result is: %d", result);
}
```



```c
#include<stdio.h>
#include<conio.h>
void main()
{
float A,B,C;
clrscr();
printf("Enter number 1:\n");
scanf("%f", &A);
printf("Enter number 2:\n");
scanf("%f", &B);
printf("Enter number 3:\n");
scanf("%f", &C);
/*Check for greatest of three numbers*/
if(A>B) {
if(A>C) {
        printf("The largest number is: %f\r
        }
else  {
        printf("The largest number is: %f\r
        }
}
else {
if(C>B) {
        printf("The largest number is: %f\r
        }
else  {
        printf("The largest number is: %f\1. ,-,,
        }
}
getch();
}
```



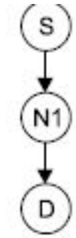
Software Testing

```
/* Program to generate the previous date given a date, assumes data
given as dd mm yyyy separated by space and performs error checks on the
validity of the current date entered. */

#include <stdio.h>
#include <conio.h>

int main()
{
  int day, month, year, validDate = 0;
/*Date Entry*/
  printf(“Enter the day value: ”);
  scanf(“%d”, &day);
  printf(“Enter the month value: ”);
  scanf(“%d”, &month);
  printf(“Enter the year value: ”);
  scanf(“%d”,&year);
/*Check Date Validity */
  if (year >= 1900 && year <= 2025) {
    if (month == 1 || month == 3 || month == 5 || month == 7 ||
      month == 8 || month == 10 || month == 12) {
```

(Contd.)...

Program graph nodes	DD path graph corresponding nodes	Remarks
1	S	Source node
2 – 7	N1	Sequential flow
8	D	Destination node

Mapping of program graph nodes and DD path graph nodes



DD Path graph

Table 3.3. Mapping of program graph nodes and DD graph nodes

Program graph nodes	DD path graph corresponding node	Remarks
1	S	Source node
2 to 10	N1	Sequential nodes, there is a sequential flow from node 2 to 10
11	N2	Decision node, if true goto 12, else goto 19
12	N3	Decision node, if true goto 13 else goto 15
13, 14	N4	Sequential nodes
15, 16, 17	N5	Sequential nodes
18	N6	Junction node, two edges 14 and 17 are terminated here
19	N7	Intermediate node terminated at node 20
20	N8	Decision node, if true goto 21 else goto 23
21, 22	N9	Sequential nodes
23, 24, 25	N10	Sequential nodes
26	N11	Junction node, two edges 22 and 25 are terminated here
27	N12	Junction node, two edges 18 and 26 are terminated here
28	D	Destination node

Software Testing

```
      if (day >= 1 && day <= 31) {
          validDate = 1;
      }
      else {
        validDate = 0;
      }
    }
    else if (month == 2) {
      int rVal=0;
      if (year%4 == 0) {
        rVal=1;
        if ((year%100)==0 && (year % 400) !=0) {
          rVal=0;
        }
      }
      if (rVal ==1 && (day >=1 && day <=29) ) {
        validDate = 1;
      }
      else if (day >=1 && day <= 28 ) {
        validDate = 1;
      }
```

(Contd.)...

Software Testing

```
      else {
        validDate = 0;
      }
    }
    else if ((month >= 1 && month <= 12) && (day >= 1 && day <= 30)) {
      validDate = 1;
    }
    else {
      validDate = 0;
    }
  }
  /*Prev Date Calculation*/
  if (validDate) {
    if (day == 1) {
      if (month == 1) {
        year--;
        day=31;
        month=12;
      }
      else if (month == 3) {
        int rVal=0;
```

(Contd.)...

Software Testing

```
        if (year%4 == 0) {
          rVal=1;
          if ((year%100)==0 && (year % 400) !=0) {
        rVal=0;
      }
        }
        if (rVal ==1) {
          day=29;
          month--;
        }
        else {
          day=28;
          month--;
        }
      }
      else if (month == 2 || month == 4 || month == 6 || month == 9 ||
      month == 11) {
        day = 31;
        month--;
```

(Contd.)...

```
          }
          else {
            day=30;
            month--;
          }
        }
        else {
          day--;
        }
        printf("The next date is: %d-%d-%d",day,month,year);
      }
      else {
        printf("The entered date ( %d-%d-%d ) is invalid",day,month, year);
      }
      getche ();
      return 1;
    }
```

Fig. 15: Program for previous date problem

Software Testing

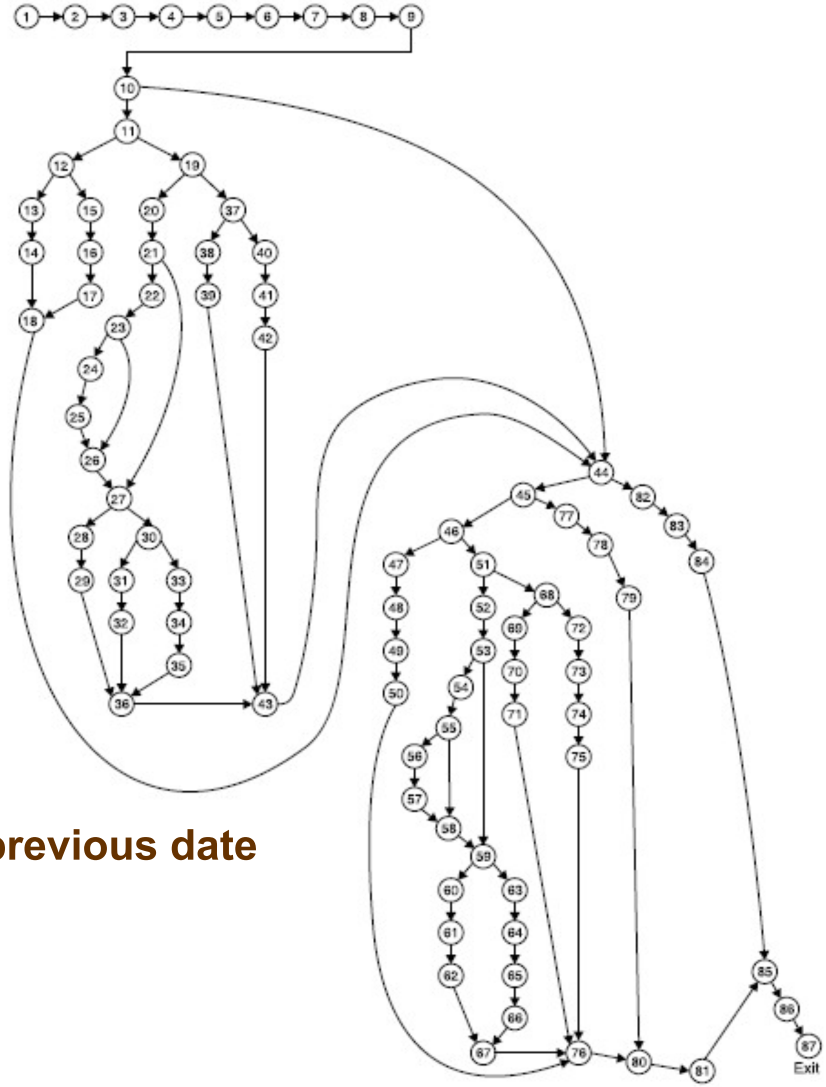


Fig. 16: Flow graph of previous date problem

Software Testing

Decision to Decision (DD) Path Graph

Table 7: Mapping of flow graph nodes and DD path nodes

Flow graph nodes	DD Path graph corresponding node	Remarks
1 to 9	n_1	There is a sequential flow from node 1 to 9
10	n_2	Decision node, if true go to 13 else go to 44
11	n_3	Decision node, if true go to 12 else go to 19
12	n_4	Decision node, if true go to 13 else go to 15
13,14	n_5	Sequential nodes and are combined to form new node n_5
15,16,17	n_6	Sequential nodes
18	n_7	Edges from node 14 to 17 are terminated here
19	n_8	Decision node, if true go to 20 else go to 37
20	n_9	Intermediate node with one input edge and one output edge
21	n_{10}	Decision node, if true go to 22 else go to 27
22	n_{11}	Intermediate node
23	n_{12}	Decision node, if true go to 24 else go to 26

Cont…

Software Testing

Flow graph nodes	DD Path graph corresponding node	Remarks
24,25	n_{13}	Sequential nodes
26	n_{14}	Two edges from node 25 & 23 are terminated here
27	n_{15}	Two edges from node 26 & 21 are terminated here. Also a decision node
28,29	n_{16}	Sequential nodes
30	n_{17}	Decision node, if true go to 31 else go to 33
31,32	n_{18}	Sequential nodes
33,34,35	n_{19}	Sequential nodes
36	n_{20}	Three edge from node 29,32 and 35 are terminated here
37	n_{21}	Decision node, if true go to 38 else go to 40
38,39	n_{22}	Sequential nodes
40,41,42	n_{23}	Sequential nodes
43	n_{24}	Three edge from node 36,39 and 42 are terminated here

Cont….

Software Testing

Flow graph nodes	DD Path graph corresponding node	Remarks
44	n_{25}	Decision node, if true go to 45 else go to 82. Three edges from 18,43 & 10 are also terminated here.
45	n_{26}	Decision node, if true go to 46 else go to 77
46	n_{27}	Decision node, if true go to 47 else go to 51
47,48,49,50	n_{28}	Sequential nodes
51	n_{29}	Decision node, if true go to 52 else go to 68
52	n_{30}	Intermediate node with one input edge & one output ege
53	n_{31}	Decision node, if true go to 54 else go to 59
54	n_{32}	Intermediate node
55	n_{33}	Decision node, if true go to 56 else go to 58
56,57	n_{34}	Sequential nodes
58	n_{35}	Two edge from node 57 and 55 are terminated here
59	n_{36}	Decision node, if true go to 60 else go to 63. Two edge from nodes 58 and 53 are terminated. **Cont….**

Software Testing

Flow graph nodes	DD Path graph corresponding node	Remarks
60,61,62	n_{37}	Sequential nodes
63,64,65,66	n_{38}	Sequential nodes
67	n_{39}	Two edge from node 62 and 66 are terminated here
68	n_{40}	Decision node, if true go to 69 else go to 72
69,70,71	n_{41}	Sequential nodes
72,73,74,75	n_{42}	Sequential nodes
76	n_{43}	Four edges from nodes 50, 67, 71 and 75 are terminated here.
77,78,79	n_{44}	Sequential nodes
80	n_{45}	Two edges from nodes 76 & 79 are terminated here
81	n_{46}	Intermediate node
82,83,84	n_{47}	Sequential nodes
85	n_{48}	Two edges from nodes 81 and 84 are terminated here
86,87	n_{49}	Sequential nodes with exit node

Software Testing

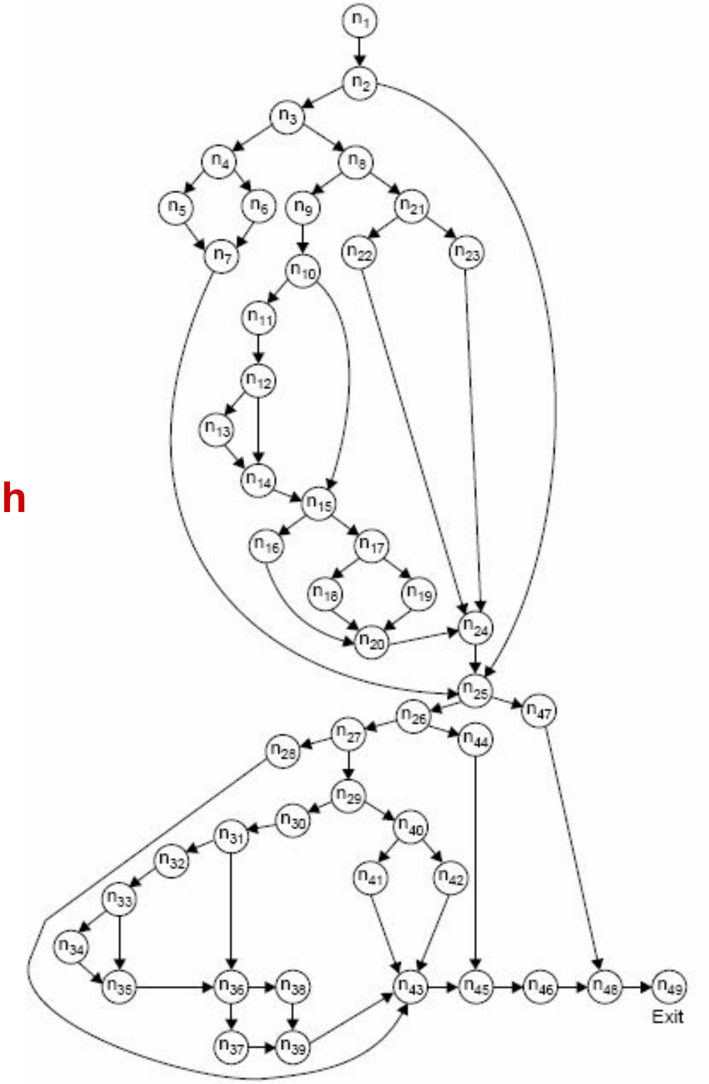


Fig. 17: DD path graph of previous date problem

	Independent paths of previous date problem
1	$n_1, n_2, n_{25}, n_{47}, n_{48}, n_{49}$
2	$n_1, n_2, n_3, n_4, n_5, n_7, n_{25}, n_{47}, n_{48}, n_{49}$
3	$n_1, n_2, n_3, n_4, n_6, n_7, n_{25}, n_{47}, n_{48}, n_{49}$
4	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
5	$n_1, n_2, n_3, n_8, n_{21}, n_{23}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
6	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{17}, n_{19}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
7	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{17}, n_{18}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
8	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{13}, n_{14}, n_{15}, n_{17}, n_{18}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
9	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{14}, n_{15}, n_{17}, n_{18}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
10	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{47}, n_{48}, n_{49}$
11	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{44}, n_{45}, n_{46}, n_{48}, n_{49}$
12	$n_1, n_2, n_3, n_8, n_9, n_{11}, n_{12}, n_{14}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{27}, n_{28}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
13	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{14}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{40}, n_{41}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
14	$n_1, n_2, n_3, n_8, n_9, n_{10}, n_{11}, n_{12}, n_{14}, n_{15}, n_{16}, n_{20}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{40}, n_{42}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
15	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{36}, n_{38}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
16	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{36}, n_{37}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
17	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{32}, n_{33}, n_{34}, n_{35}, n_{36}, n_{37}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$
18	$n_1, n_2, n_3, n_8, n_{21}, n_{22}, n_{24}, n_{25}, n_{26}, n_{27}, n_{29}, n_{30}, n_{31}, n_{32}, n_{33}, n_{35}, n_{36}, n_{37}, n_{39}, n_{43}, n_{45}, n_{46}, n_{48}, n_{49}$

Fig. 18: Independent paths of previous date problem

Software Testing

Example 8.13

Consider the problem for the determination of the nature of roots of a quadratic equation. Its input a triple of positive integers (say a,b,c) and value may be from interval [0,100].

The program is given in fig. 19. The output may have one of the following words:

[Not a quadratic equation; real roots; Imaginary roots; Equal roots]

Draw the flow graph and DD path graph. Also find independent paths from the DD Path graph.

Software Testing

```
#include <conio.h>
#include <math.h>
    int main()
  {
    int a,b,c,validInput=0,d;
    double D;
    printf("Enter the 'a' value: ");
    scanf("%d",&a);
    printf("Enter the 'b' value: ");
    scanf("%d",&b);
    printf("Enter the 'c' value: ");
    scanf("%d",&c);
    if ((a >= 0) && (a <= 100) && (b >= 0) && (b <= 100) && (c >= 0)
      && (c <= 100)) {
      validInput = 1;
      if (a == 0) {
        validInput = -1;
      }
    }
    if (validInput==1) {
      d = b*b - 4*a*c;
      if (d == 0) {
        printf("The roots are equal and are r1 = r2 = %f\n",
               -b/(2*(float) a));
```

Cont….

Software Testing

```
        }
        else if ( d > 0 ) {
          D=sqrt(d);
          printf("The roots are real and are r1 = %f and r2 = %f\n",
                  (-b-D)/(2* a), (-b+D)/(2* a));
        }
        else {
          D=sqrt(-d)/(2*a);
          printf("The roots are imaginary and are r1 = (%f,%f) and
                  r2 = (%f,%f)\n", -b/(2.0*a),D,-b/(2.0*a),-D);
        }
      }
      else if (validInput == -1) {
        printf("The vlaues do not constitute a Quadratic equation.");
      }
      else {
        printf("The inputs belong to invalid range.");
      }
      getche();
      return 1;
    }
```

Fig. 19: Code of quadratic equation problem

Solution

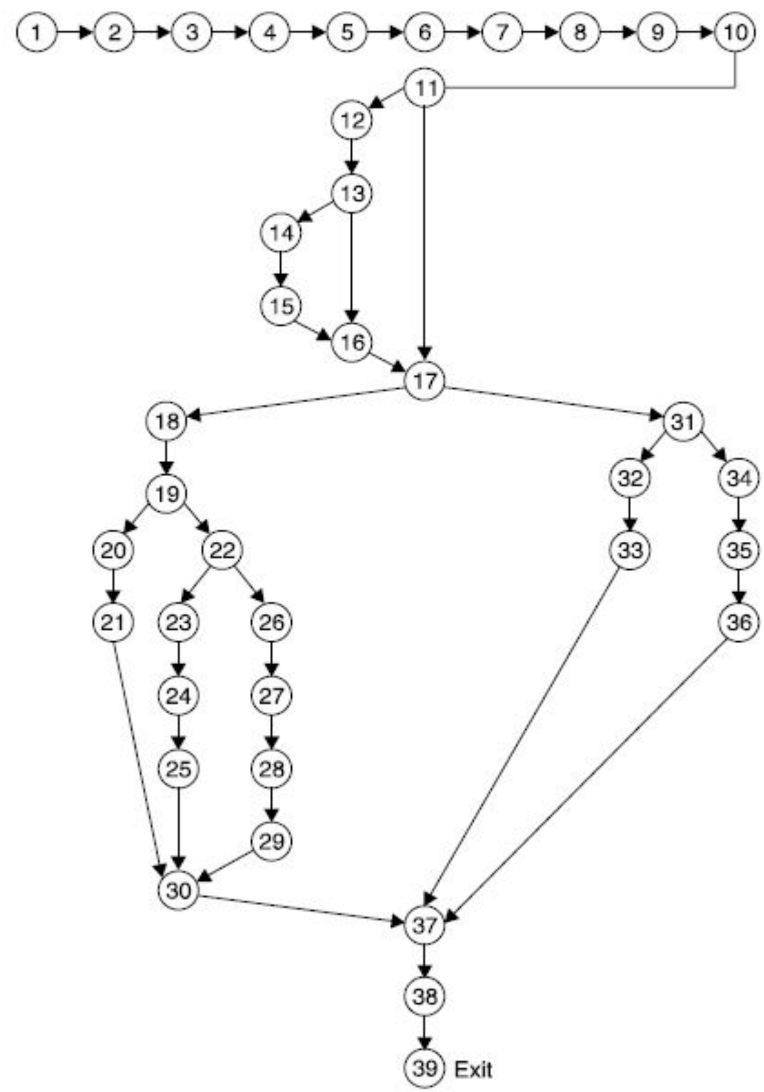


Fig. 19 (a) : Program flow graph

Software Testing

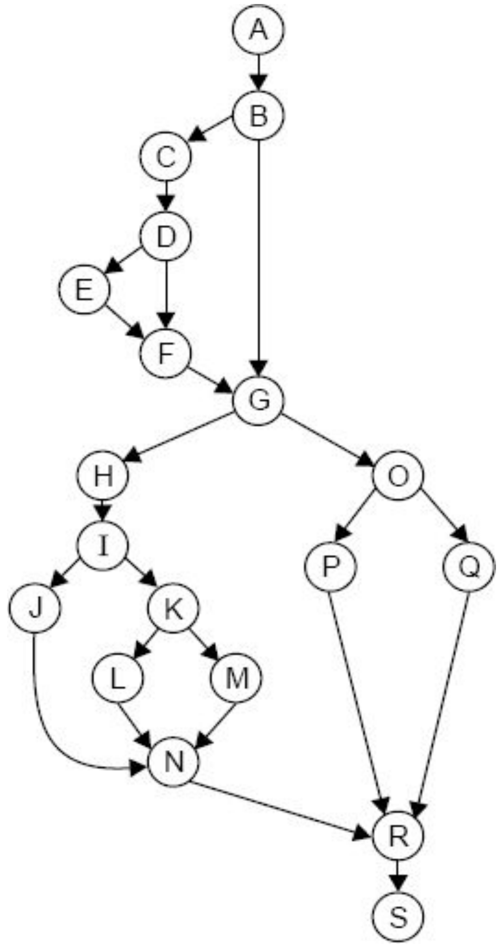


Fig. 19 (b) : DD Path graph

Software Testing

The mapping table for DD path graph is:

Flow graph nodes	DD Path graph corresponding node	Remarks
1 to 10	A	Sequential nodes
11	B	Decision node
12	C	Intermediate node
13	D	Decision node
14,15	E	Sequential node
16	F	Two edges are combined here
17	G	Two edges are combined and decision node
18	H	Intermediate node
19	I	Decision node
20,21	J	Sequential node
22	K	Decision node
23,24,25	L	Sequential node

Cont….

Software Testing

Flow graph nodes	DD Path graph corresponding node	Remarks
26,27,28,29	M	Sequential nodes
30	N	Three edges are combined
31	O	Decision node
32,33	P	Sequential node
34,35,36	Q	Sequential node
37	R	Three edges are combined here
38,39	S	Sequential nodes with exit node

Independent paths are:

(i) ABGOQRS
(ii) ABGOPRS
(iii) ABCDFGOQRS
(iv) ABCDEFGOPRS
(v) ABGHIJNRS
(vi) ABGHIKLNRS
(vi) ABGHIKMNRS

Software Testing

Example 8.14

Consider a program given in Fig.8.20 for the classification of a triangle. Its input is a triple of positive integers (say a,b,c) from the interval [1,100]. The output may be [Scalene, Isosceles, Equilateral, Not a triangle].

Draw the flow graph & DD Path graph. Also find the independent paths from the DD Path graph.

Software Testing

```
#include <stdio.h>
#include <conio.h>
int main()
{
  int a,b,c,validInput=0;
  printf(“Enter the side ‘a’ value: ”);
  scanf(“%d”,&a);
  printf(“Enter the side ‘b’ value: ”)
  scanf(“%d”,&b);
  printf(“Enter the side ‘c’ value:”);
  scanf(“%d”,&c);
  if ((a > 0) && (a <= 100) && (b > 0) && (b <= 100) && (c > 0)
      && (c <= 100)) {
    if ( (a + b) > c) && ((c + a) > b) && ((b + c) > a)) {
      validInput = 1;
    }
  }
  else {
    validInput = -1;
  }
  If (validInput==1) {
    If ((a==b) && (b==c)) {
      printf(“The trinagle is equilateral”);
    }
    else if ( (a == b) || (b == c) || (c == a) ) {
```

(Contd.)...

Software Testing

```
            printf(“The triangle is isosceles”);
          }
          else {
            printf(“The trinagle is scalene”);
          }
        }
        else if (validInput == 0) {
          printf(“The values do not constitute a Triangle”);
        }
        else {
          printf(“The inputs belong to invalid range”);
        }
        getche();
        return 1;
      }
```

Fig. 20 : Code of triangle classification problem

Software Testing

Solution :

Flow graph of triangle problem is:

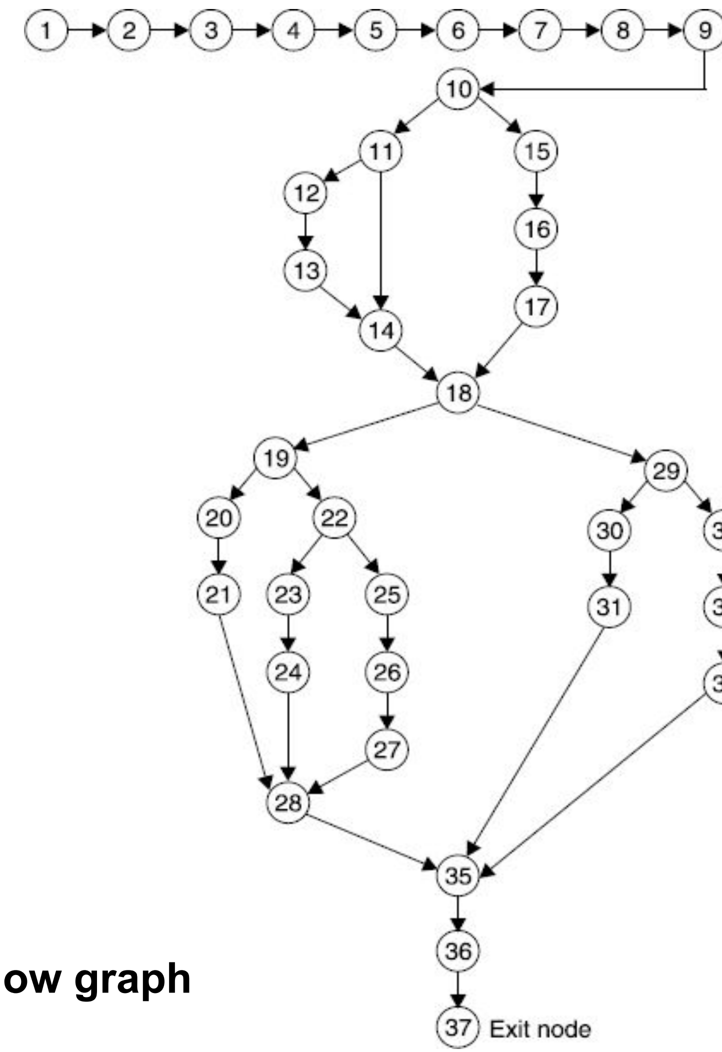


Fig.8. 20 (a): Program flow graph

Software Testing

The mapping table for DD path graph is:

Flow graph nodes	DD Path graph corresponding node	Remarks
1 TO 9	A	Sequential nodes
10	B	Decision node
11	C	Decision node
12, 13	D	Sequential nodes
14	E	Two edges are joined here
15, 16, 17	F	Sequential nodes
18	G	Decision nodes plus joining of two edges
19	H	Decision node
20, 21	I	Sequential nodes
22	J	Decision node
23, 24	K	Sequential nodes
25, 26, 27	L	Sequential nodes

Cont….

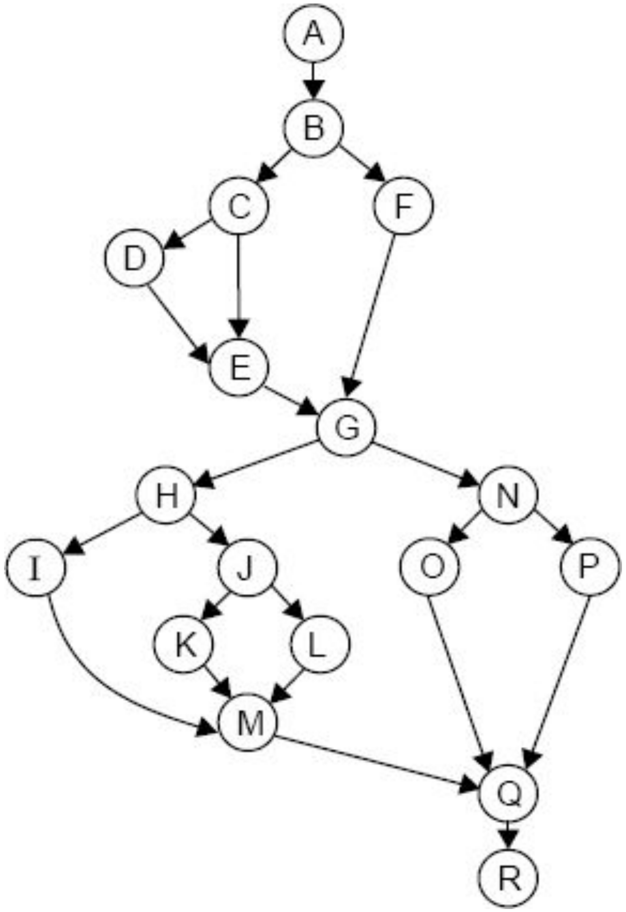
Software Testing

Flow graph nodes	DD Path graph corresponding node	Remarks
28	M	Three edges are combined here
29	N	Decision node
30, 31	O	Sequential nodes
32, 33, 34	P	Sequential nodes
35	Q	Three edges are combined here
36, 37	R	Sequential nodes with exit node

Fig. 20 (b): DD Path graph

Software Testing

DD Path graph is given in Fig. 20 (b)



Independent paths are:

(i) ABFGNPQR
(ii) ABFGNOQR
(iii) ABCEGNPQR
(iv) ABCDEGNOQR
(v) ABFGHIMQR
(vi)ABFGHJKMQR
(vii)ABFGHJMQR

Fig. 20 (b): DD Path graph

Software Testing

Cyclomatic Complexity

McCabe's cyclomatic metric $V(G) = e - n + 2P$.

For example, a flow graph shown in in Fig. 21 with entry node '*a*' and exit node '*f*'.

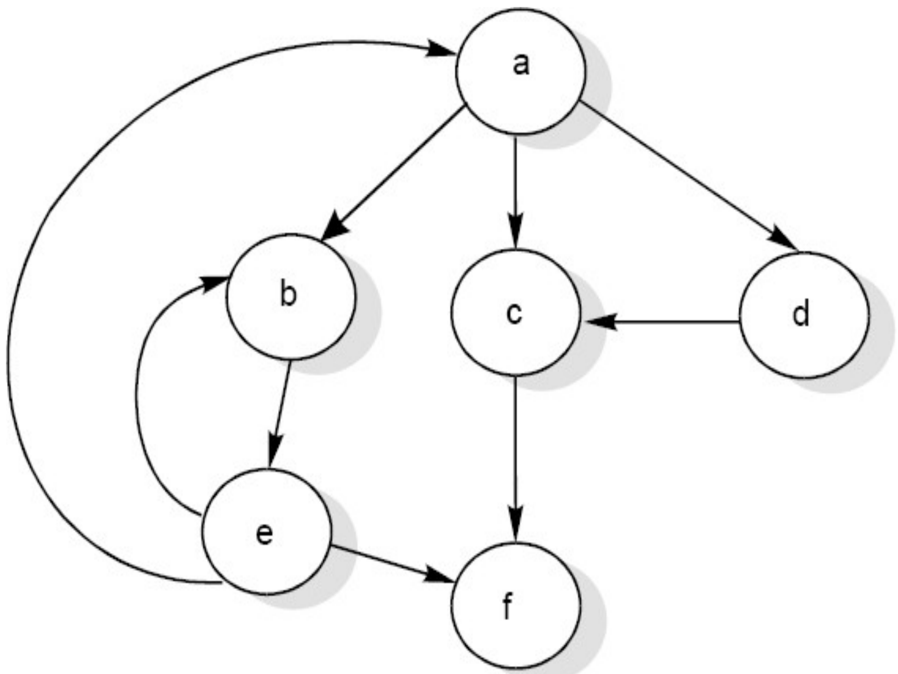


Fig. 21: Flow graph

Software Testing

The value of cyclomatic complexity can be calculated as :

$$V(G) = 9 - 6 + 2 = 5$$

Here $e = 9$, $n = 6$ and P =1

There will be five independent paths for the flow graph illustrated in Fig. 21.

Path 1 : *a c f*

Path 2 : *a b e f*

Path 3 : *a d c f*

Path 4 : *a b e a c f* **or** *a b e a b e f*

Path 5 : *a b e b e f*

Software Testing

Several properties of cyclomatic complexity are stated below:

1. $V(G) \geq 1$
2. V (G) is the maximum number of independent paths in graph G.
3. Inserting & deleting functional statements to G does not affect V(G).
4. G has only one path if and only if V(G)=1.
5. Inserting a new row in G increases V(G) by unity.
6. V(G) depends only on the decision structure of G.

Software Testing

The role of P in the complexity calculation $V(G)=e-n+2P$ is required to be understood correctly. We define a flow graph with unique entry and exit nodes, all nodes reachable from the entry, and exit reachable from all nodes. This definition would result in all flow graphs having only one connected component. One could, however, imagine a main program M and two called subroutines A and B having a flow graph shown in Fig. 22.

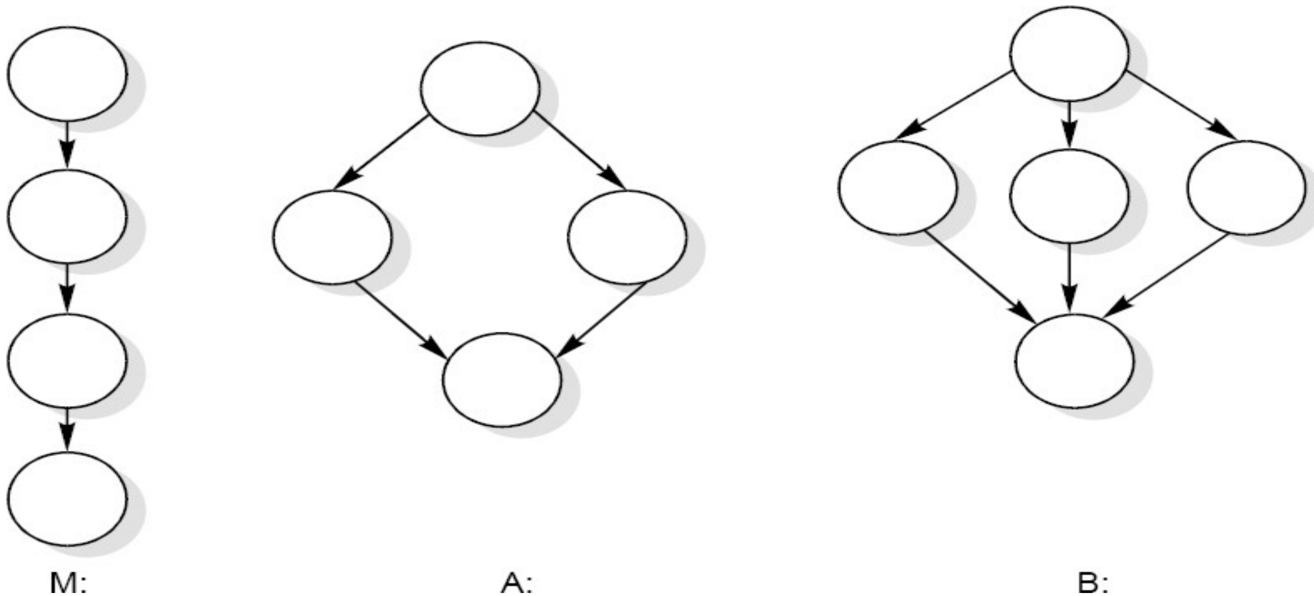


Fig. 22

Software Testing

Let us denote the total graph above with 3 connected components as

$$V(M \cup A \cup B) = e - n + 2P$$

$$= 13-13+2*3$$

$$= 6$$

This method with $P \neq 1$ can be used to calculate the complexity of a collection of programs, particularly a hierarchical nest of subroutines.

Software Testing

Notice that $V(M \cup A \cup B) = V(M) + V(A) + V(B) = 6$. In general, the complexity of a collection C of flow graphs with K connected components is equal to the summation of their complexities. To see this let C_i, $1 \leq I \leq K$ denote the k distinct connected component, and let e_i and n_i be the number of edges and nodes in the ith-connected component. Then

$$V(C) = e - n + 2p = \sum_{i=1}^{k} e_i - \sum_{i=1}^{k} n_i + 2K$$

$$= \sum_{i=1}^{k} (e_i - n_i + 2) = \sum_{i=1}^{k} V(C_i)$$

Software Testing

Two alternate methods are available for the complexity calculations.

1. Cyclomatic complexity V(G) of a flow graph G is equal to the number of predicate (decision) nodes plus one.

   $V(G) = \Pi + 1$

   Where Π is the number of predicate nodes contained in the flow graph G.

2. Cyclomatic complexity is equal to the number of regions of the flow graph.

Example 8.15

Consider a flow graph given in Fig. 23 and calculate the cyclomatic complexity by all three methods.

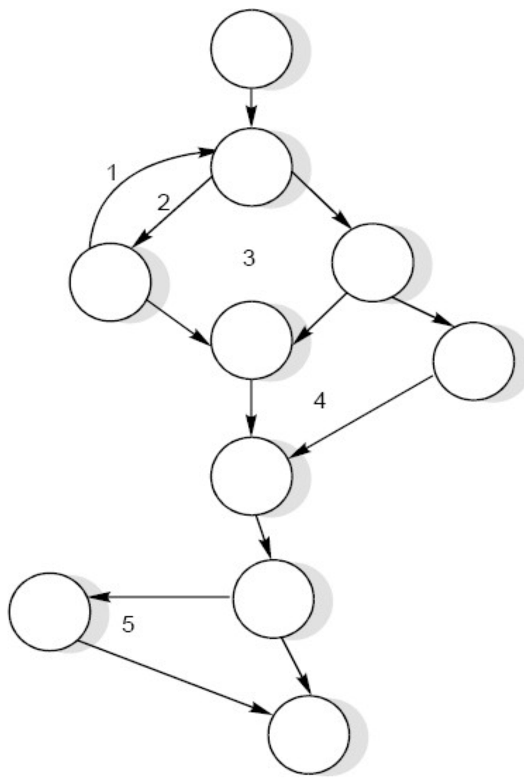


Fig. 23

Software Testing

Solution

Cyclomatic complexity can be calculated by any of the three methods.

1. V(G) $= e - n + 2P$

$= 13 - 10 + 2 = 5$

2. V(G) $= \pi + 1$

$= 4 + 1 = 5$

3. V(G) = number of regions

$= 5$

Therefore, complexity value of a flow graph in Fig. 23 is 5.

Software Testing

Example 8.16

Consider the previous date program with DD path graph given in Fig. 17. Find cyclomatic complexity.

Software Testing

Solution

Number of edges (e) = 65

Number of nodes (n) =49

(i) $V(G) = e - n + 2P = 65 - 49 + 2 = 18$

(ii) $V(G) = \pi + 1 = 17 + 1 = 18$

(iii) V(G) = Number of regions = 18

The cyclomatic complexity is 18.

Example 8.17

Consider the quadratic equation problem given in example 8.13 with its DD Path graph. Find the cyclomatic complexity:

Software Testing

Solution

Number of nodes (n) = 19

Number of edges (e) = 24

(i) V(G) = $e - n$ + 2P = 24 – 19 + 2 = 7

(ii) V(G) = π + 1 = 6 + 1 = 7

(iii) V(G) = Number of regions = 7

Hence cyclomatic complexity is meaning thereby, seven
7 independent paths in the DD Path
graph.

Example 8.18

Consider the classification of triangle problem given in example 8.14. Find the cyclomatic complexity.

Software Testing

Solution

Number of edges (e) = 23

Number of nodes (n) =18

(i) $V(G) = e - n + 2P = 23 - 18 + 2 = 7$

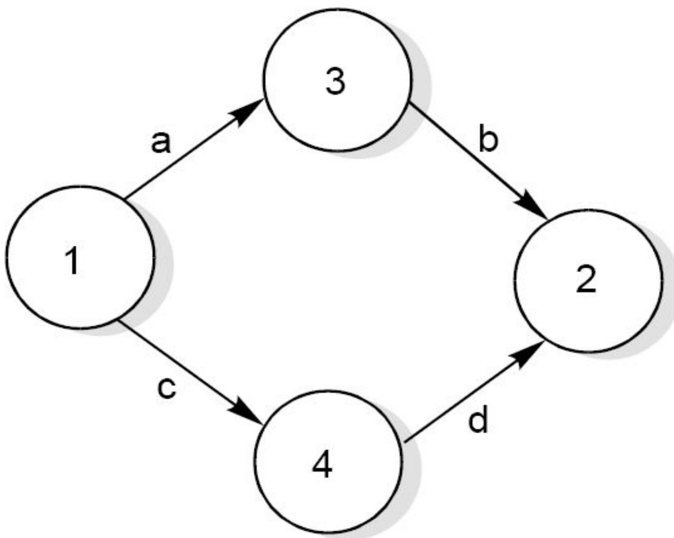
(ii) $V(G) = \pi + 1 = 6 + 1 = 7$

(iii) V(G) = Number of regions = 7

The cyclomatic complexity is 7. Hence, there are seven independent paths as given in example 8.14.

Graph Matrices

A graph matrix is a square matrix with one row and one column for every node in the graph. The size of the matrix (i.e., the number of rows and columns) is equal to the number of nodes in the flow graph. Some examples of graphs and associated matrices are shown in fig. 24.



Flow graph

	1	2	3	4
1			a	c
2				
3		b		
4		d		

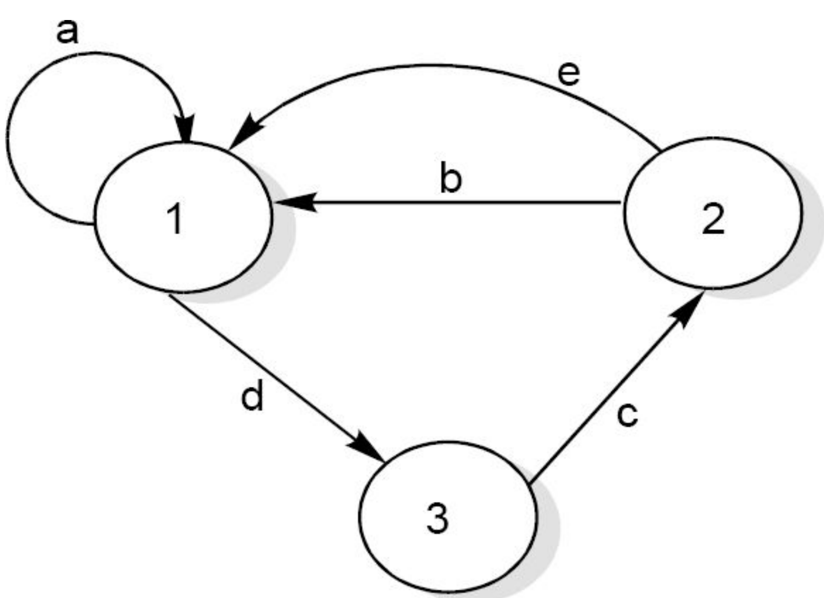
Graph Matrix

(a)

Fig. 24 (a): Flow graph and graph matrices

(Contd.)...

Software Testing



Flow graph

(b)

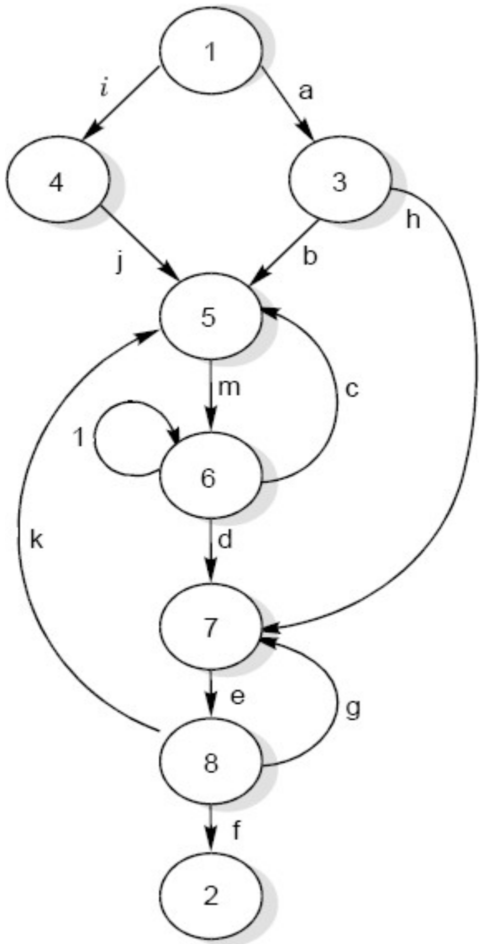
	1	2	3
1	a		d
2	b + e		
3		c	

Graph Matrix

Fig. 24 (b): Flow graph and graph matrices

(Contd.)...

Software Testing



Flow graph

	1	2	3	4	5	6	7	8
1			a	i				
2								
3					b		h	
4					j			
5						m		
6					c	l	d	
7								e
8		f			k		g	

Graph Matrix

(c)

Fig. 24 (c): Flow graph and graph matrices

Software Testing

	1	2	3	4	5	6	7	8	Connections
1			1	1					2 – 1 = 1
2									
3					1		1		2 – 1 = 1
4					1				1 – 1 = 0
5						1			1 – 1 = 0
6					1	1	1		3 – 1 = 2
7								1	1 – 1 = 0
8		1			1		1		3 – 1 = 2
									6 + 1 = 7

Fig. 25 : Connection matrix of flow graph shown in Fig. 24 (c)

Software Testing

	1	2	3	4
1			a	c
2				
3		b		
4		d		

[A]

	1	2	3	4
1		ab + cd		
2				
3				
4				

$[A]^2$

The square matrix represent that there are two path *ab* and *cd* from node 1 to node 2.

Example 8.19

Consider the flow graph shown in the Fig. 26 and draw the graph & connection matrices. Find out cyclomatic complexity and two / three link paths from a node to any other node.

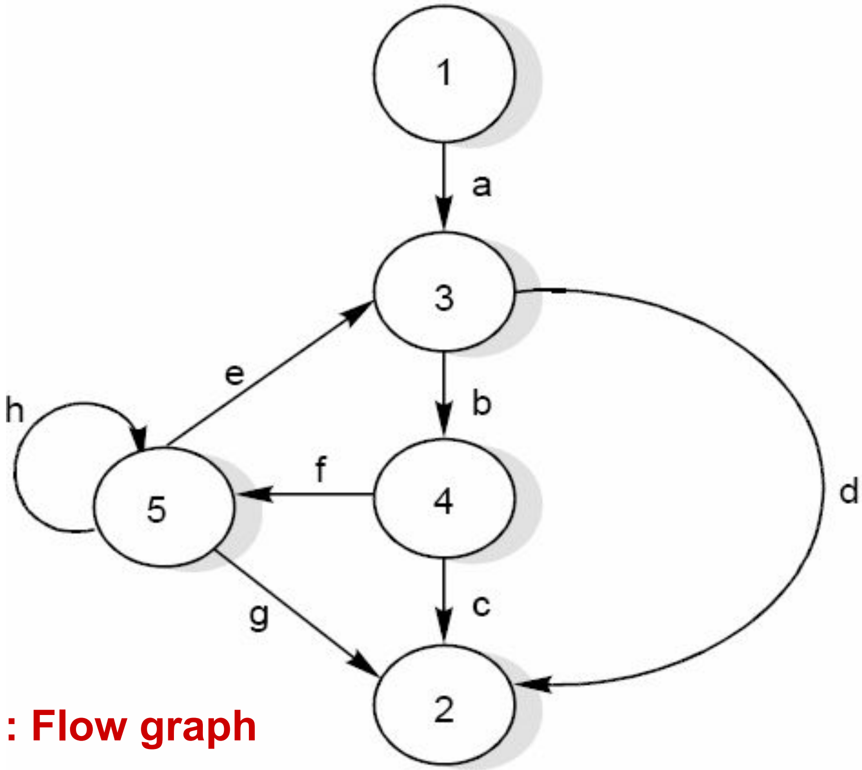


Fig. 26 : Flow graph

Software Testing

Solution

The graph & connection matrices are given below :

	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		h

Graph Matrix (A)

	1	2	3	4	5	Connections
1			1			1 – 1 = 0
2						
3		1		1		2 – 1 = 1
4		1			1	2 – 1 = 1
5		1	1		1	3 – 1 = 2
						4 + 1 = 5

Connection Matrix

To find two link paths, we have to generate a square of graph matrix [A] and for three link paths, a cube of matrix [A] is required.

Software Testing

	1	2	3	4	5
1		ad		ab	
2					
3		bc			bf
4		fg	fe		fh
5		ed + hg	he	eb	h^2

$[A^2]$

	1	2	3	4	5
1		abc			afb
2					
3		bfg	bfe		bfh
4		fed + fhg	fhe	feb	fh^2
5		$ebc + hed + h^2g$	h^2e	heb	$ebf + h^3$

$[A^3]$

Software Testing

Control Flow Testing

We identify paths of the program and write test cases to execute those paths.

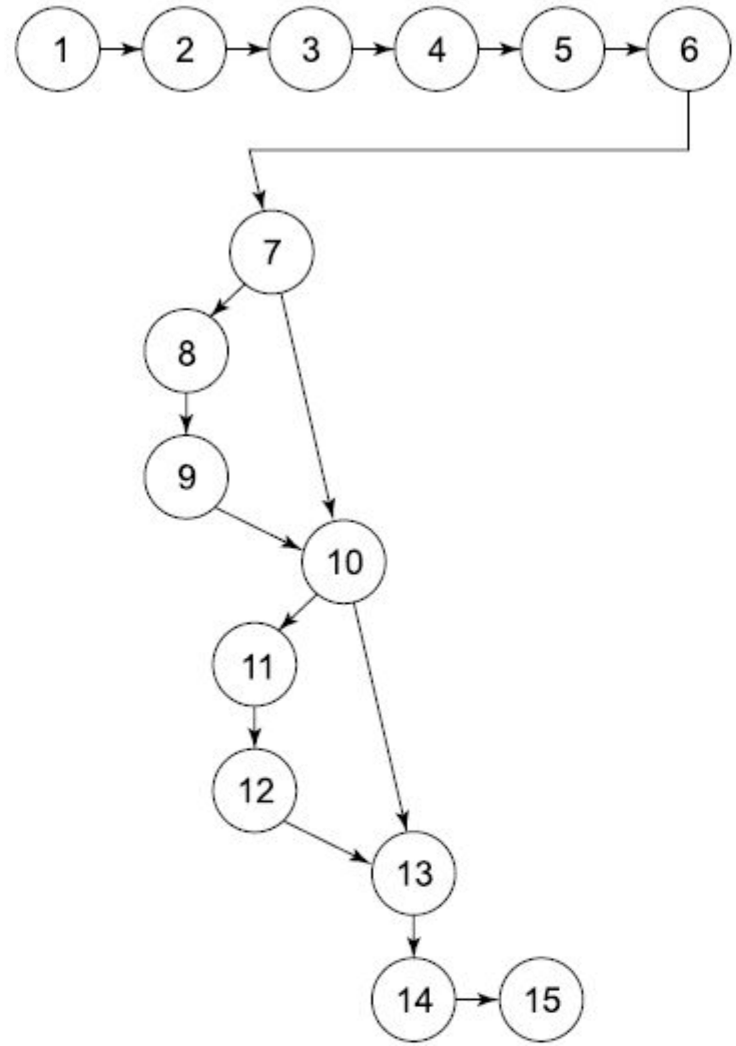
We define ‘coverage’ as a ‘percentage of source code that has been tested with respect to the total source code available for testing’.

The most reasonable level may be to test every statement of a program at least once before the completion of testing.

We may write test cases that ensure the execution of every statement.

Software Testing

```c
#include<stdio.h>
#include<conio.h>

void main()
{
int a,b,c,x=0,y=0;
clrscr();
printf("Enter three numbers:");
scanf("%d %d %d",&a,&b,&c);
if((a>b)&&(a>c)){
        x=a*a+b*b;
}
if(b>c){
        y=a*a-b*b;
}
printf("x= %d y= %d",x,y);
getch();
}
```



Software Testing

The cyclomatic complexity of this graph is:

$$V(G) = e - n + 2P = 16 - 15 + 2 = 3$$

$$V(G) = \text{no. of regions} = 3$$

$$V(G) = \Pi + 1 = 2 + 1 = 3$$

Hence, independent paths are three and are given as:

(i) 1–7, 10, 13–15
(ii) 1–7, 10–15
(iii) 1–10, 13–15

Software Testing

Branch Coverage :

We want to test every branch of the program. Hence, we wish to test every 'True' and 'False' condition of the program.

a = 9, b = 8, c = 7

a = 7, b = 8, c = 9

Condition Coverage:

Condition coverage is better than branch coverage because we want to test every condition at least once.

(i) a = 9, b = 8, c = 7 (first possibility when both are true)
(ii) a = 9, b = 8, c = 10 (second possibility – first is true, second is false)
(iii) a = 7, b = 8, c = 9 (third and fourth possibilities- first is false, statement number 7 is false)

Software Testing

Path Coverage:

In this coverage criteria, we want to test every path of the program.

S. No.	Paths Id.	Paths	Inputs a	b	c	Expected Output
1.	Path-1	1–7,10, 13–15	7	8	9	x=0 y=0
2.	Path-2	1–7, 10–15	7	8	6	x=0 y=–15
3.	Path-3	1–10, 13–15	9	7	8	x=130 y=0
4.	Path-4	1–15	9	8	7	x=145 y=17

Software Testing

Data Flow Testing

Data flow testing is another from of structural testing. It has nothing to do with data flow diagrams.

i. Statements where variables receive values.

ii. Statements where these values are used or referenced.

As we know, variables are defined and referenced throughout the program. We may have few define/ reference anomalies:

i. A variable is defined but not used/ referenced.

ii. A variable is used but never defined.

iii. A variable is defined twice before it is used.

Software Testing

Definitions

The definitions refer to a program P that has a program graph G(P) and a set of program variables V. The G(P) has a single entry node and a single exit node. The set of all paths in P is PATHS(P)

(i) **Defining Node:** Node n z G(P) is a defining node of the variable *v* z V, written as DEF (*v*, *n*), if the value of the variable *v* is defined at the statement fragment corresponding to node *n*.

(ii) **Usage Node:** Node n z G(P) is a usage node of the variable *v* z V, written as USE (*v*, *n*), if the value of the variable *v* is used at statement fragment corresponding to node *n*. A usage node USE (*v*, n) is a predicate use (denote as p) if statement n is a predicate statement otherwise USE (*v*, n) is a computation use (denoted as *c*).

Software Testing

(iii) **Definition use:** A definition use path with respect to a variable v (denoted du-path) is a path in PATHS(P) such that, for some *v* z V, there are define and usage nodes DEF(v, m) and USE(v, n) such that m and n are initial and final nodes of the path.

(iv) **Definition clear :** A definition clear path with respect to a variable v (denoted dc-path) is a definition use path in PATHS(P) with initial and final nodes DEF (*v*, m) and USE (*v*, n), such that no other node in the path is a defining node of *v*.

The du-paths and dc-paths describe the flow of data across source statements from points at which the values are defined to points at which the values are used. The du-paths that are not definition clear are potential trouble spots.

Software Testing

4.2.3 Identification of du and dc Paths

The various steps for the identification of du and dc paths are given as:

(i) Draw the program graph of the program.

(ii) Find all variables of the program and prepare a table for define / use status of all variables using the following format:

S. No.	Variable(s)	Defined at node	Used at node

(iii) Generate all du-paths from define/use variable table of step (iii) using the following format:

S. No.	Variable	du-path(begin, end)

(iv) Identify those du-paths which are not dc-paths.

Software Testing

Hence, our objective is to find all du-paths and then identity those du-paths which are not dc-paths. The steps are given in Fig. 27. We may like to generate specific test cases for du-paths that are not dc-paths.

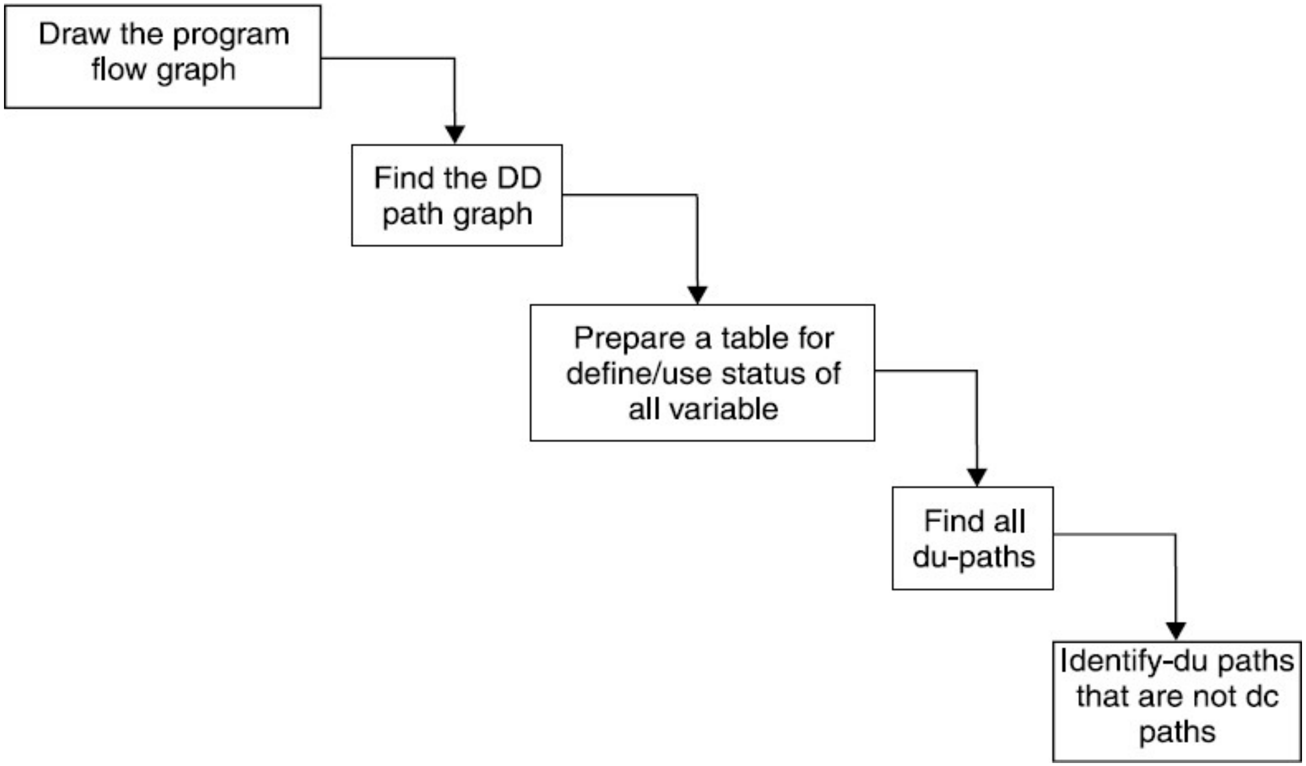


Fig. 27 : Steps for data flow testing

S. No.	Variable	Defined at node	Used at node
1.	A	6	11, 12, 13
2.	B	8	11, 20, 24
3.	C	10	12, 16, 20, 21

Variable	du-path (Begin, end)
A	6, 11
	6, 12
	6, 13
B	8, 11
	8, 20
	8, 24
C	10, 12
	10, 16
	10, 20
	10, 21

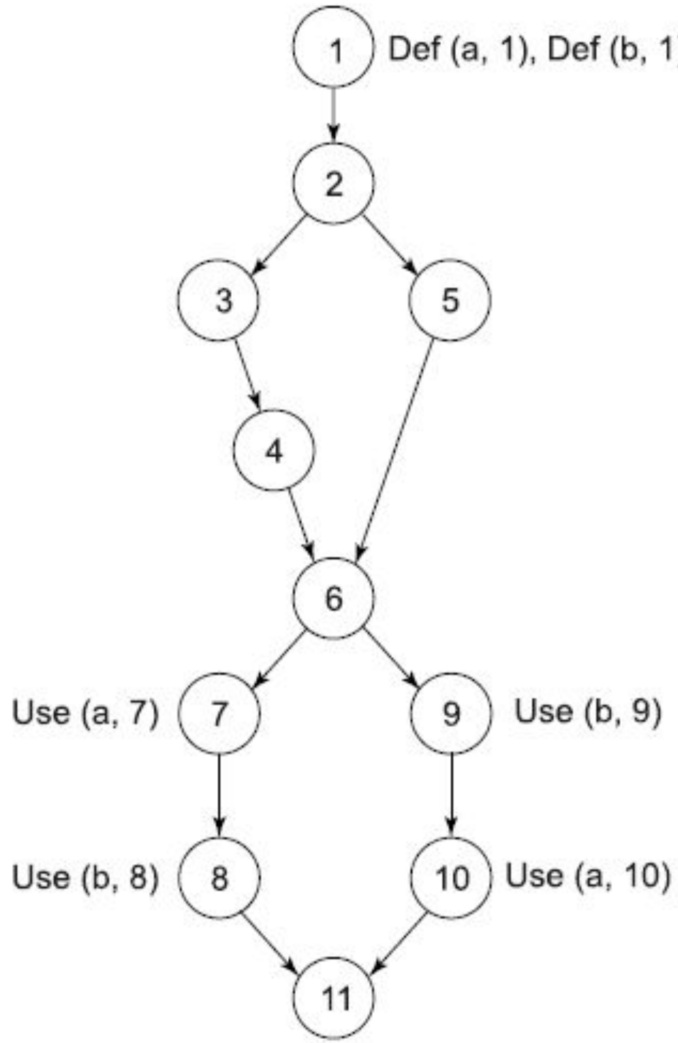
	Paths	Definition clear?
All du paths and all uses (Both are same in this example)	6–11	Yes
	6–12	Yes
	6–13	Yes
	8–11	Yes
	8–11, 19, 20	Yes
	8–11, 19, 20, 23, 24	Yes
	10–12	Yes
	10–12, 15, 16	Yes
	10, 11, 19, 20	Yes
	10, 11, 19–21	Yes
All definitions	6–11	Yes
	8–11	Yes
	10–12	Yes

Test all du-paths

S. No.	Inputs A	B	C	Expected Output	Remarks
1.	9	8	7	9	6–11
2.	9	8	7	9	6–12
3.	9	8	7	9	6–13
4.	7	9	8	9	8–11
5.	7	9	8	9	8–11, 19, 20
6.	7	9	8	9	8–11, 19, 20, 23, 24
7.	8	7	9	9	10–12
8.	8	7	9	9	10–12, ,15, 16
9.	7	8	9	9	10, 11, 19, 20
10.	7	8	9	9	10, 11, 19–21

Test All definitions

S. No.	Inputs A	B	C	Expected Output	Remarks
1.	9	8	7	9	6–11
2.	7	9	8	9	8–11
3.	8	7	9	9	10–12


1
Def (a, 1), Def (b, 1)
2
3
5
4
6
Use (a, 7)
7
9
Use (b, 9)
Use (b, 8)
8
10
Use (a, 10)
11

S. No.	Variables	Defined at node	Used at node
1.	a	1	7, 10
2.	b	1	8, 9

S. No.	Variables	du-paths (Begin, end)
1.	a	1, 7
		1, 10
2.	b	1, 8
		1, 9

	Paths	Definition clear?
All du paths	1–4, 6, 7	Yes
(8 paths)	1, 2, 5–7	Yes
	1–4, 6, 9, 10	Yes
	1, 2, 5, 6, 9, 10	Yes
	1–4, 6, 7, 8	Yes
	1, 2, 5–8	Yes
	1–4, 6, 9	Yes
	1, 2, 5, 6, 9	Yes

	Paths	Definition clear?
All uses	1–4, 6, 7	Yes
(4 paths)	1–4, 6, 9, 10	Yes
	1–4, 6-8	Yes
	1–4, 6, 9	Yes
All definitions	1–4, 6, 7	Yes
(2 paths)	1–4, 6–8	Yes

Example 8.20

Consider the program of the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values for each of these may be from interval [0,100]. The program is given in Fig. 19. The output may have one of the option given below:

(i) Not a quadratic program

(ii) real roots

(iii) imaginary roots

(iv) equal roots

(v) invalid inputs

Find all du-paths and identify those du-paths that are definition clear.

Solution

Step I: The program flow graph is given in Fig. 19 (a). The variables used in the program are a,b,c,d, validinput, D.

Step II: DD Path graph is given in Fig. 19(b). The cyclomatic complexity of this graph is 7 indicating there are seven independent paths.

Step III: Define/use nodes for all variables are given below:

Variable	Defined at node	Used at node
a	6	11,13,18,20,24,27,28
b	8	11,18,20,24,28
c	10	11,18
d	18	19,22,23,27
D	23, 27	24,28
Validinput	3, 12, 14	17,31

Software Testing

Variable	Path (beginning, end) nodes	Definition clear ?
c	10, 11 10, 18	Yes Yes
d	18, 19 18, 22 18, 23 18, 27	Yes Yes Yes Yes
D	23, 24 23, 28 27, 24 27, 28	Yes Path not possible Path not possible Yes
validinput	3, 17 3, 31 12, 17 12, 31 14, 17 14, 31	no no no no yes yes

Software Testing

Example 8.21

Consider the program given in Fig. 20 for the classification of a triangle. Its input is a triple of positive integers (say a,b,c) from the interval [1,100]. The output may be:

[Scalene, Isosceles, Equilateral, Not a triangle, Invalid inputs].

Find all du-paths and identify those du-paths that are definition clear.

Software Testing

Solution

Step I: The program flow graph is given in Fig. 20 (a). The variables used in the program are a,b,c, valid input.

Step II: DD Path graph is given in Fig. 20(b). The cyclomatic complexity of this graph is 7 and thus, there are 7 independent paths.

Step III: Define/use nodes for all variables are given below:

Variable	Defined at node	Used at node
a	6	10, 11, 19, 22
b	7	10, 11, 19, 22
c	9	10, 11, 19, 22
valid input	3, 13, 16	18, 29

Software Testing

Variable	Path (beginning, end) nodes	Definition clear ?
c	9, 10 9, 11 9, 19 9, 22	Yes Yes Yes Yes
valid input	3, 18 3, 29 12, 18 12, 29 16, 18 16, 29	no no no no Yes Yes

Hence total du-paths are 18 out of which four paths are not definition clear