Criticality Factor of Modules with the Application of Modified Neighborhood Integration Testing and Initiating Build Testing

Dr. Namita Gupta[1], Nitesh Goyal[2]

1 Head of Department, Computer Science Engineering, Maharaja Agrasen Institute of Technology, Rohini, Delhi - 110086, India

2 Student, Computer Science Engineering, Maharaja Agrasen Institute of Technology, Rohini, Delhi - 110086, India

(me.niteshgoyal@gmail.com, +919310345131, +919599944144)

Abstract— Software testing is the process used to measure the quality of developed computer software. It is done to find all the bugs, flaws, broken paths, stray components etc. in a given program or software. The initial step of doing software testing includes testing the most critical module of all, so that main body of software is tested first. In this paper the technique of finding the most critical module of all depending on functional dependency is being discussed with the initiation of build testing.

Keywords— Software Testing, Software Module, Module Criticality, Structural Testing, Integration Testing, Build Testing and Call Graph.

INTRODUCTION

Software testing is an art of finding flaws, bugs and causes of errors in a given software.
The software testing is must as most of the machines work on a computer algorithm and if any bug or flaw occurs in this algorithm the results may be catastrophic. Testing enhances the quality of a given software. Before initiating testing process, several test cases or constraints are defined on the given software, on the basis of which best testing technique is chosen. Without declaring these constraints the tester cannot decide what kind of testing is being needed in a specific software.
The test cases must include the following information:

The Input Information: a) Preconditions, it includes the information of circumstances hold prior to test case execution.
b) The actual inputs identified by some *Testing Method*.

The Output Information: a) Postconditions, it includes information of circumstances that need to be satisfied after the execution, like, number of outputs should be finite.
b) Actual outputs identified by some *Testing Method*.

Different testing techniques are available to generate the test cases based on software code complexity to be tested. Two fundamental approaches used to determine the test cases are – *Functional Testing and Structural Testing.*
Functional Testing consider software as function which will give an output for some given input or as a software which is generating a set of outputs (range) for set of inputs (domain). In this approach, the code of software is considered as *Black Box* which is generating a set of outputs for the entered input. If the output is not as per the requirements (constraints on software) than it is considered to contain bug(s), which needs to be rectified.
Structural Testing considers the structure/ internal logic of the software to find the bugs. That is why it is also termed as *White Box Testing*. The approach helps to detect defects like linking bugs, missing statements, unused code fragments and many more.

Process of testing starts in parallel to first phase of Software development life cycle (SDLC). Test cases are generated considering the user's requirements collected during requirement gathering phase of SDLC. This is called static testing. Similarly, different testing techniques are available to test the activities performed at design phase. Various black-box and white-box testing techniques are used to test each and every module implemented during the coding phase of SDLC. Then Integration testing is performed to test the interface between the individual modules. *Call Graph based Neighbourhood Integration Testing* is one of the integration testing technique used to detect the interface errors between the modules. In this technique, call graph showing all the modules of software is drawn.
Call Graph is basically a directed graph, which represents the links between modules, that is, it represents the directed link(s) between software modules telling the tester that which module(s) is/are being called by a particular module. The internal structure of the software is clearly represented by a call graph.

For example, here the modules are represented as nodes (numbered from 1 to 11) and the arrows represent the links between the different software modules as per the module calls made by them. The obtained structure forms a *call graph.*

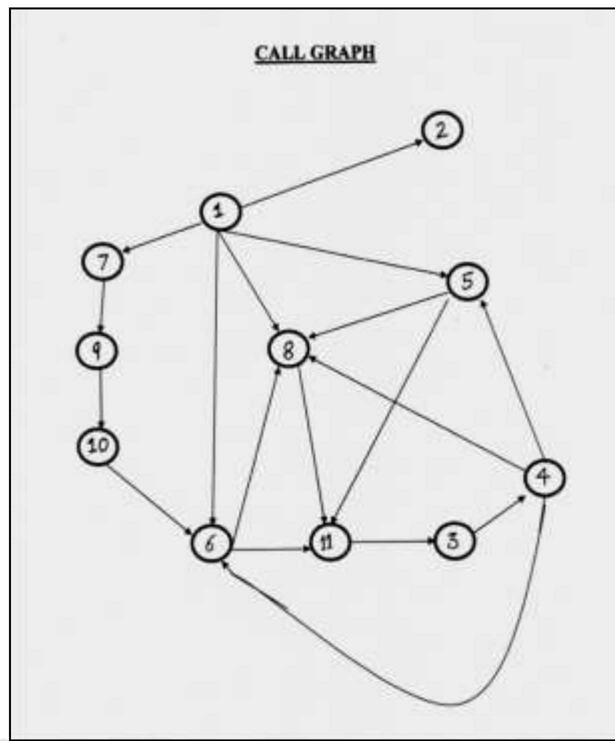

Fig. 1
The call graph of a BANK system.

Neighbourhood Integration Testing considers the neighbour modules of each and every software module represented as nodes in call graph. Topmost module in call graph with zero indegree is termed as root node, all modules having outdegree zero are termed l*eaf nodes*, and the modules being called are termed as *Successors, and* modules calling some other modules are termed *Predecessors.*
The biggest *advantage* of using this technique is that it reduces the number of test cases thus saving testing time.
But, the technique also has some disadvantages mention below:

1. If a node (module) that has been repeated most of the numbers of times and it is found that it has some error in that case the whole process need to be repeated again from the very starting for all the neighbourhoods related to that faulty node (module).
2. The modules are being checked only from its outer cover, but a lot depends on the internal threads also of those individual modules.

To overcome the above two disadvantages of call graph based neighbourhood integration testing, in this paper a new integration testing technique called "*Friend Integration testing*" is proposed. This new technique identifies the friends or (neighbours) of each node of call graph, including the root and the leave nodes and then identifies the most critical nodes to be tested based on the criticality which is discussed in next section.

PROPOSED FRIEND INTEGRATION TESTING TECHNIQUE

Software containing multiple integrated modules interface with each other and exchange necessary information among themselves to accomplish the required functionality as documented in user's specification document.
Due to time constraint, testing each and every module is not possible. Hence build testing is used to test the modules in different builds, each build containing modules of same criticality. Criticality of a module is determined based on its interaction with other modules of the same software. Module with highest indegree and outdegree is the most critical node as any defect in that module has direct impact on its calling and called modules.

Proposed *Friend* technique determines the criticality of each module using the formula given below:

$$C = \frac{Module\ (Successores)}{Graph\ (Successors)} \quad ; such\ that\ 0\ \leq\ C \leq 1$$

Module (Predecessors) = Number of modules calling the ith module
Module (Successors) = Number of modules being called by i^{th} module
Graph (Successors) = ΣModule (Successors), where indegree ≠0

Builds are created based on the following observations:

1. If Criticality Factor (C) of two modules M1 and M2 shows same value, then
   If Module M1 (Predecessor) > Module M2 (Predecessor) then
   Module M1 is considered more critical than module M2.
   Reason - Higher the number of predecessors more the module is being called by other modules. Thus higher the *Criticality Factor.*

2. If M1(Successors) == M2(Successors) and
   M1 (Predecessor) == Module M2 (Predecessor) then
   Compute the path length from root node to M1 and M2 using *Breath First Search (BFS) method.*
   Module with smaller path length is considered more critical.

   Reason *–main()* module is the most critical module of any software and Module near to Main module affects it more as compared to module far from Main module.

3. Modules showing same value of *Criticality Factor (*C*)* are kept in same build.

4. Module(s) are tested in decreasing order of their criticality factor values.

CASE STUDIES

CASE STUDY 1

Taking in account the example of BANK System.

The *BANK System* represents the minimal features that a bank constitutes. A user can see the bank branch, set up a new account, account number will be generated automatically on which user sets a password, and the deposit or withdrawal of money can be done with user login only. The balance of the account can be seen, every time, after any action on the user account take place.

Modules are:

MODULE NUMBERS	MODULES
1	main() *//Main function*
2	branch() *//Tell branch of bank*
3	get_password() *//Password of specific account.*

4	check_password () *//Password verification.*
5	withdraw() *//Withdraw money from bank*
6	deposit() *//Deposit money in bank*
7	new_account () *//Creating new account.*
8	balance() *//Display balance of the account*
9	generate_ac_no () *//Generating account number for new account.*
10	set_password() *//Creating new password*
11	ac_no() *//Taking account number of existing account.*

Table 1
Layout of BANK system.

(The call graph for BANK System is given in Fig 1)

MODULE NUMBERS	MODULE PREDECESSORS	MODULE SUCCESSORS
1	-	2, 5, 6, 7, 8
2	1	-
3	11	4
4	3	5, 6, 8
5	1, 4	8, 11
6	1, 10	8, 11
7	1	9
8	1, 4, 5, 6	11
9	7	10
10	9	6
11	5, 6, 8	3

TABLE 2
Friends of nodes in Bank System

Criticality Factor of modules using the formula —

$$C = \frac{Module\ (Successores)}{Graph\ (Successors)} \quad ; such\ that\ 0 \leq C \leq 1$$

Module (Predecessors) = Number of modules calling the ith module
Module (Successors) = Number of modules being called by i^{th} module
Graph (Successors) = ΣModule (Successors), where indegree ≠0

Using Table 2 Graph (Successors) = **18**.

1. For Module 1(From Table 2)

Module (Successors) = **5** ≡ (2, 5, 6, 7, 8 are the Module Successors)
The Criticality Factor for Module 1 using the above formula is calculated as follows:

$\mathbf{C_1 = \frac{5}{18} = 0.28}$

2. For Module 2(From Table 2)

   Module (Successors) = **0** ≡ (No Module Successors)
   The Criticality Factor for Module 2 using the above formula is calculated as follows:

   $\mathbf{C_2 = \frac{0}{18} = 0}$

3. For Module 3(From Table 2)

   Module (Successors) = **1** ≡ (4 is the Module Successor)
   The Criticality Factor for Module 3 using the above formula is calculated as follows:

   $\mathbf{C_3 = \frac{1}{18} = 0.056}$

4. For Module 4(From Table 2)

   Module (Successors) = **3** ≡ (5, 6, 8 are the Module Successors)
   The Criticality Factor for Module 4 using the above formula is calculated as follows:

   $\mathbf{C_4 = \frac{3}{18} = 0.167}$

5. For Module 5(From Table 2)

   Module (Successors) = **2** ≡ (8, 11 are the Module Successors)
   The Criticality Factor for Module 5 using the above formula is calculated as follows:

   $\mathbf{C_5 = \frac{2}{18} = 0.111}$

6. For Module 6(From Table 2)

   Module (Successors) = **2** ≡ (8, 11 are the Module Successors)
   The Criticality Factor for Module 6 using the above formula is calculated as follows:

   $\mathbf{C_6 = \frac{2}{18} = 0.111}$

7. For Module 7(From Table 2)

   Module (Successors) = **1** ≡ (9 is the Module Successor)
   The Criticality Factor for Module 7 using the above formula is calculated as follows:

   $\mathbf{C_7 = \frac{1}{18} = 0.056}$

8. For Module 8(From Table 2)

   Module (Successors) = **1** ≡ (11 is the Module Successor)
   The Criticality Factor for Module 8 using the above formula is calculated as follows:

   $\mathbf{C_8 = \frac{1}{18} = 0.056}$

9. For Module 9(From Table 2)

   Module (Successors) = **1** ≡ (10 is the Module Successor)

The Criticality Factor for Module 9 using the above formula is calculated as follows:

$\mathbf{C_9 = \frac{1}{18} = 0.056}$

10. For Module 10(From Table 2)

Module (Successors) = **1** ≡ (6 is the Module Successor)
The Criticality Factor for Module 10 using the above formula is calculated as follows:

$\mathbf{C_{10} = \frac{1}{18} = 0.056}$

11. For Module 11(From Table 2)

Module (Successors) = **1** ≡ (3 is the Module Successor)
The Criticality Factor for Module 11 using the above formula is calculated as follows:

$\mathbf{C_{11} = \frac{1}{18} = 0.056}$

The table of Criticality Factor of Modules for BANK System is:

MODULE NUMBERS	CRITICALITY FACTOR
1	0.28
2	0
3	0.056
4	0.167
5	0.111
6	0.111
7	0.056
8	0.056
9	0.056
10	0.056
11	0.056

Table 3
Criticality factor of modules of BANK system using Formula 1

Step 1:

If Criticality Factor (C) of two modules M1 and M2 shows same value, then
If Module M1 (Predecessor) > Module M2 (Predecessor) then
Module M1 is considered more critical than module M2.

The modules (5, 6) and (3, 7, 8, 9, 10, 11) have the same Criticality Factor, based on given rule the criticality of modules is:
N5 = N6
N8 > N11 > (N3 =N7 = N9 = N10)

Step 2:

If M1 (Successors) == M2 (Successors) and
M1 (Predecessor) == Module M2 (Predecessor) then
Compute the path length from root node to M1 and M2 using *Breath First Search (BFS) method.*
Module with smaller path length is considered more critical.

There is ambiguity in modules (5, 6) and (3, 7, 9, 10), following above rule:

By applying BFS the results is:
Path for N5 = **N1 -> N5**
Path for N6 = **N1 -> N6**
Path for N3 = **N1 -> N8 -> N11 ->N3**
Path for N7 = **N1 -> N7**
Path for N9 = **N1 -> N7 -> N9**
Path for N10 = **N1 -> N7 -> N9 -> N10**

From above observation the criticality of modules is:
N5 = N6
N7 > N9 > N10 = N3

Step 3:

Modules showing same value of C are kept in same build.

Distinction in the criticality of modules N3, N10 **and N5, N6** cannot be made, following above rule the criticality of all the modules is:

N1 > N4 > N5 = N6 > N8 > N11 > N7 > N9 > N10 = N3 > N2

Step 4:

Module(s) are tested in decreasing order of their criticality factor values.

Using above rule, the builds are:
Build 1 = N1
Build 2 = N4
Build 3 = N5, N6
Build 4 = N8
Build 5 = N11
Build 6 = N7
Build 7 = N9
Build 8 = N10, N3
Build 9 = N2

Each build will contain more than one modules (in most of the cases) if a more complex software system (containing many modules) is taken into account.
The build testing can now be applied to these builds which tell the tester that the given system can be tested further or not, that is, whether the given system is of any use or not.

CASE STUDY 2

Taking in account the example of User Data Storage Device (UDSD) System

The *User Data Storage Device (UDSD) System* represents the minimal features of user data storage device. User can load new data on storage device, update the existing data (data on storage device), and can directly load the network data (user data taken directly from internet). Using the network data user can access the E-Mails or other relevant data. The user data is automatically processed after every action on the storage media take place. At the end user can exit from the system.

Modules are:

MODULE NUMBERS	MODULES	
1	Main()	//main function
2	Load_user_data()	//loads user data
3	Process()	//processes user data
4	Done()	//display message processing is done
5	Mark_read()	//marks read when data is read by device
6	Get_sms_email()	//takes sms/email from internet()
7	Update_data()	//updates user data
8	Load_network_data()	//load data from the internet
9	Exit()	//exits from program

TABLE 4
Layout of User Data Storage Device (UDSD) System

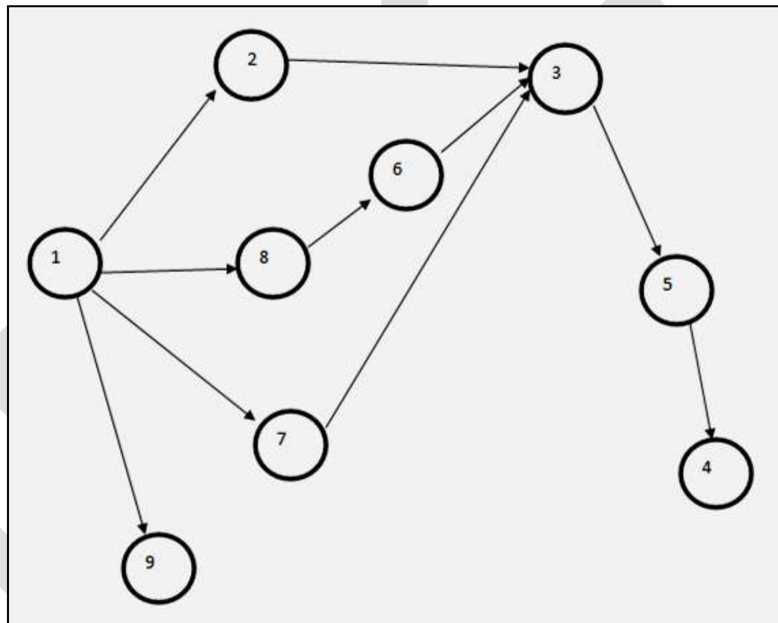

FIG 2
Call graph of User Data Storage Device (UDSD) System

MODULE NUMBERS	MODULE PREDECESSORS	MODULE SUCCESSORS
1	-	2, 7, 8, 9
2	1	3
3	2, 6, 7	5
4	5	-
5	3	4
6	8	3
7	1	3
8	1	6
9	1	-

TABLE 5
Friends of nodes in UDSD System

Criticality Factor of modules using the formula —

$$C = \frac{Module\ (Successores)}{Graph\ (Successors)} \quad ; such\ that\ 0 \leq C \leq 1$$

Module (Predecessors) = Number of modules calling the ith module
Module (Successors) = Number of modules being called by i^{th} module
Graph (Successors) = ΣModule (Successors), where indegree ≠0

Using Table 5, *Graph (Successors)* = **10**
The table of Criticality Factor of Modules for UDSD System is:

MODULE NUMBERS	CRITICALITY FACTOR
1	0.4
2	0.1
3	0.1
4	0
5	0.1
6	0.1
7	0.1
8	0.1
9	0

TABLE 6
Criticality factor of modules of UDSD system using Formula 1

Step 1:

If Criticality Factor (C) of two modules M1 and M2 shows same value, then
If Module M1 (Predecessor) > Module M2 (Predecessor) then
Module M1 is considered more critical than module M2.

The modules (4, 9) and (2, 3, 5, 6, 7, 8) have the same Criticality Factor, based on given rule the criticality of modules is:
N4 = N9
N3 > N5 = N6 = N7 = N8

Step 2:

If M1 (Successors) == M2 (Successors) and
M1 (Predecessor) == Module M2 (Predecessor) then
Compute the path length from root node to M1 and M2 using *Breath First Search (BFS) method.*
Module with smaller path length is considered more critical.

There is ambiguity in modules (4, 9) and (2, 5, 6, 7, 8), following the above rule:
By applying BFS the result is:
Path for N4 = **N1 -> N2 -> N3 -> N5 -> N4**
Path for N9 = **N1 -> N9**
Path for N2 = **N1 -> N2**

Path for N5= **N1 -> N2 -> N3 -> N5**
Path for N6 = **N1 ->N8 -> N6**
Path for N7 = **N1 -> N7**
Path for N8 = **N1 -> N8**

From above observation the criticality of modules is:
N9 > N4
N2 = N7 = N8 > N6 > N5

Step 3:

Modules showing same value of C are kept in same build.

Distinction in the criticality of modules N2**, N7, N8** cannot be made, following above rule the criticality of all the modules is:

N1 > N3 > N2 = N7 =N8 > N6 > N5 > N9 > N4

Step 4:

Module(s) are tested in decreasing order of their criticality factor values.

Using above rule, the builds are:
Build 1 = N1
Build 2 = N3
Build 3 = N2, N7, N8
Build 4 = N6
Build 5 = N5
Build 6 = N9
Build 7 = N4

The build testing can now be applied to these builds which tell the tester that the given system can be tested further or not, that is, whether the given system is of any use or not.

CASE STUDY 3

Taking in account the example of Movie Theatre System

The *Movie Theatre System* represents the minimal features of a movie theatre. User can choose the movie of the choice, can book required number of seats as per the luxury (only if seats are available in asked slot), and can buy food items. The tickets will be printed automatically with a bill signifying total payable amount to user.

Modules are:

MODULE NUMBERS	MODULES	
1	Main()	*//Main function*
2	Movie1()	*//Movie name*
3	Movie2()	*//Movie name*
4	Movie3()	*//Movie name*
5	Movie4()	*//Movie name*
6	Number_of_seats()	*//Number seats chosen*

7	Type_of_seats()	*//Type of seats (diamond, gold, silver, etc.)*
8	Payable_amount()	*//Total payable amount*
9	Food()	*//Food bought*
10	Print_bill()	*//Print bill*
11	Print_ticket()	*//Print movie tickets*

TABLE 7
Layout of Movie Theatre System

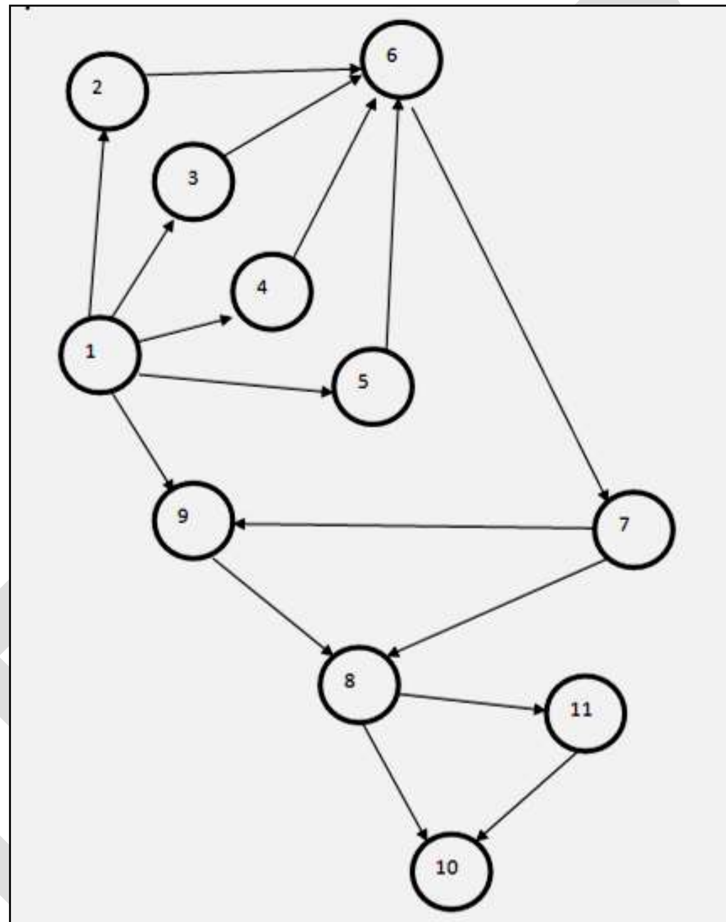

FIG 3
Call graph of User Data Storage Device (UDSD) System

MODULE NUMBERS	MODULE PREDECESSORS	MODULE SUCCESSORS
1	-	2, 3, 4, 5, 9
2	1	6
3	1	6
4	1	6
5	1	6
6	2, 3, 4, 5	7
7	6	8, 9
8	7, 9	10, 11
9	1, 7	8
10	10, 11	-

11	8	10

TABLE 8
Friends of nodes in Movie Theatre System

Criticality Factor of modules using the formula—

$$C = \frac{Module\ (Successores)}{Graph\ (Successors)} \quad ; such\ that\ 0 \le C \le 1$$

Module (Predecessors) = Number of modules calling the ith module
Module (Successors) = Number of modules being called by i^{th} module
Graph (Successors) = ΣModule (Successors), where indegree ≠0

Using Table 8, *Graph (Successors)* = **16**
The table of Criticality Factor of Modules for Movie Theatre System is:

MODULE NUMBERS	CRITICALITY FACTOR
1	0.3125
2	0.0625
3	0.0625
4	0.0625
5	0.0625
6	0.0625
7	0.125
8	0.125
9	0.0625
10	0
11	0.0625

TABLE 9
Criticality factor of modules of MOVIE THEATRE system using Formula 1

Step 1:

If Criticality Factor (C) of two modules M1 and M2 shows same value, then
If Module M1 (Predecessor) > Module M2 (Predecessor) then
Module M1 is considered more critical than module M2.

The modules (7, 8) and (2, 3, 4, 5, 6, 9, 11) have the same Criticality Factor, based on given rule the criticality of modules is:
N8 > N7
N6 > N9 > N2 = N3 = N4 = N5 = N11

Step 2:

If M1 (Successors) == M2 (Successors) and
M1 (Predecessor) == Module M2 (Predecessor) then
Compute the path length from root node to M1 and M2 using *Breath First Search (BFS) method.*

Module with smaller path length is considered more critical.

There is ambiguity in modules (2, 3, 4, 5, 11), following the above rule:
By applying BFS the result is:
Path for N2 = **N1 -> N2**
Path for N3 = **N1 -> N3**
Path for N4 = **N1 -> N4**
Path for N5 = **N1 -> N5**
Path for N6 = **N1 -> N9 -> N8 -> N11**

From above observation the criticality of modules is:
N2 = N3 = N4 = N5 > N11

Step 3:

Modules showing same value of C are kept in same build.

Distinction in the criticality of modules N2**, N3, N4, N5**cannot be made, following above rule the criticality of all the modules is:

N1 > N8 > N7 > N6 > N9 > N2 = N3 = N4 = N5 > N11 > N10

Step 4:

Module(s) are tested in decreasing order of their criticality factor values.

Using above rule, the builds are:
Build 1 = N1
Build 2 = N8
Build 3 = N7
Build 4 = N6
Build 5 = N9
Build 6 = N2, N3, N4, N5
Build 7 = N11
Build 8 = N10

The build testing can now be applied to these builds which tell the tester that the given system can be tested further or not, that is, whether the given system is of any use or not.

CASE STUDY 4

Taking in account the example of Simple Automatic Teller Machine (SATM) System, (taken from "Software Testing, A Craftsman's Approach ", Third Edition, by Paul C. Jorgensen ")

The *Simple Automatic Teller Machine (SATM) System* consists most of the features of an ATM. The user can withdraw or deposit money from it only after the PIN verification, can modify the account password, can see the present balance and print a receipt for it. The machine automatically dispenses the money entered by user (only if amount of money in user account is greater than or equal to the entered amount), and the balance is updated automatically depending on the transaction. At the end user can close the present session.

Modules are:

MODULE NUMBER	MODULE SEQUENCE	MODULES
1	1	SATM System
2	1.1	Device Sense and Control
3	1.1.1	Door Sense and Control
4	1.1.1.1	Get Door Status
5	1.1.1.2	Control Door
6	1.1.1.3	Dispense Cash
7	1.1.2	Slot Sense and Control
8	1.1.2.1	Watch Card Slot
9	1.1.2.2	Get Deposit Slot Status
10	1.1.2.3	Control Card Roller
11	1.1.2.4	Control Envelope Roller
12	1.1.2.5	Read Card Strip
13	1.2	Central Bank Communication
14	1.2.1	Get PIN for PAN
15	1.2.2	Get Account Status
16	1.2.3	Post Daily Transactions
17	1.3	Terminal Sense and Control
18	1.3.1	Screen Driver
19	1.3.2	Key Sensor
20	1.4	Manage Session
21	1.4.1	Validate Card
22	1.4.2	Validate Pin
23	1.4.2.1	Get PIN
24	1.4.3	Close Session
25	1.4.3.1	New Transaction Request

26	1.4.3.2	Print Receipt
27	1.4.3.3	Post Transaction Local
28	1.4.4	Manage Transaction
29	1.4.4.1	Get Transaction Type
30	1.4.4.2	Get Account Type
31	1.4.4.3	Report Balance
32	1.4.4.4	Process Deposit
33	1.4.4.5	Process Withdrawal

TABLE 10
Layout of SATM System

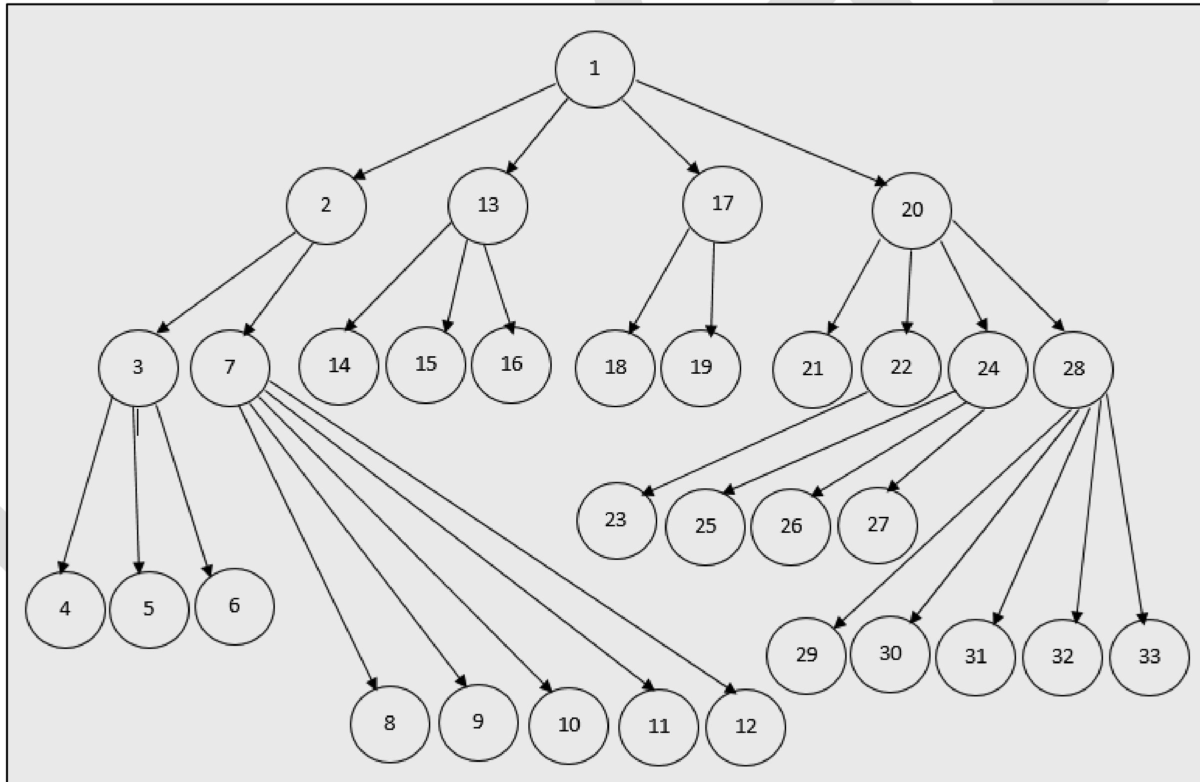

FIG 4
Call graph of User Data Storage Device (UDSD) System

MODULE NUMBERS	MODULE PREDECESSORS	MODULE SUCCESSORS
1	-	2, 13, 17 ,20
2	1	3, 7

3	2	4, 5, 6
4	3	-
5	3	-
6	3	-
7	2	8, 9, 10, 11, 12
8	7	-
9	7	-
10	7	-
11	7	-
12	7	-
13	1	14, 15, 16
14	13	-
15	13	-
16	13	-
17	1	18, 19
18	17	-
19	17	-
20	1	21, 22, 24, 28
21	20	-
22	20	23
23	22	-
24	20	25, 26, 27
25	24	-
26	24	-
27	24	-
28	20	29, 30, 31, 32, 33
29	28	-
30	28	-

31	28	-
32	28	-
33	29	-

TABLE 11
Friends of nodes in SATM System

Criticality Factor of modules using the formula—

$$C = \frac{Module\ (Successores)}{Graph\ (Successors)} \quad ; such\ that\ 0 \leq C \leq 1$$

Module (Predecessors) = Number of modules calling the ith module
Module (Successors) = Number of modules being called by i^{th} module
Graph (Successors) = ΣModule (Successors), where indegree ≠0

Using Table 11, *Graph (Successors)* = **32**
The table of Criticality Factor of Modules for SATM System is:

MODULE NUMBERS	CRITICALITY FACTOR
1	0.125
2	0.062
3	0.093
4	0
5	0
6	0
7	0.156
8	0
9	0
10	0
11	0
12	0
13	0.093
14	0

15	0
16	0
17	0.062
18	0
19	0
20	0.125
21	0
22	0.031
23	0
24	0.093
25	0
26	0
27	0
28	0.156
29	0
30	0
31	0
32	0
33	0

TABLE 12
Criticality factor of modules of SATM System using Formula 1

Step 1:

If Criticality Factor (C) of two modules M1 and M2 shows same value, then
If Module M1 (Predecessor) > Module M2 (Predecessor) then
Module M1 is considered more critical than module M2.

The modules (1, 20), (2, 17), (3, 13, 24), (7, 28) and (4, 5, 6, 8, 9, 10, 11, 12, 14, 15, 16, 18, 19, 21, 23, 25, 26, 27, 29, 30, 31, 32, 33) have the same Criticality Factor, based on given rule the criticality of modules is:
N20 > N1
N2 = N17
N3 = N13 = N24
N7 = N28
N4 = N5 = N6 = N8 = N9 = N10 = N11 = N12 = N14 = N15 = N16 = N18 = N19 = N21 = N23 = N25 = N26 = N27 = N29 = N30 = N31 = N32 = N33

Step 2:

If M1 (Successors) == M2 (Successors) and
M1 (Predecessor) == Module M2 (Predecessor) then
Compute the path length from root node to M1 and M2 using *Breath First Search (BFS) method.*
Module with smaller path length is considered more critical.

There is ambiguity in modules (2, 17), (3, 13, 24), (7, 28) and (4, 5, 6, 8, 9, 10, 11, 12, 14, 15, 16, 18, 19, 21, 23, 25, 26, 27, 29, 30, 31, 32, 33), following the above rule :
By applying BFS the result is:
Path for N2 = **N1 -> N2**
Path for N17 = **N1 -> N17**
Path for N3 = **N1 -> N2 -> N3**
Path for N13 = **N1 -> N13**
Path for N24 = **N1 -> N20 -> N24**
Path for N7 = **N1 -> N2 -> N7**
Path for N28 = **N1 -> N20 -> N28**
Path for N4 = **N1 -> N2 -> N3 -> N4**
Path for N5 = **N1 -> N2 -> N3 -> N5**
Path for N6 = **N1 -> N2 -> N3 -> N6**
Path for N8 = **N1 -> N2 -> N7 -> N8**
Path for N9 = **N1 -> N2 -> N7 -> N9**
Path for N10 = **N1 -> N2 -> N7 -> N10**
Path for N11 = **N1 -> N2 -> N7 -> N11**
Path for N12 = **N1 -> N2 -> N7 -> N12**
Path for N14 = **N1 -> N13 -> N14**
Path for N15 = **N1 -> N13 -> N15**
Path for N16 = **N1 -> N13 -> N16**
Path for N18 = **N1 -> N17 -> N18**
Path for N19 = **N1 -> N17 -> N19**
Path for N14 = **N1 -> N13 -> N14**
Path for N21 = **N1 -> N20 -> N21**
Path for N23 = **N1 -> N20 -> N22 -> N23**
Path for N25 = **N1 -> N20 -> N24 -> N25**
Path for N26 = **N1 -> N20 -> N24 -> N26**
Path for N27 = **N1 -> N20 -> N24 -> N27**
Path for N29 = **N1 -> N20 -> N28 -> N29**
Path for N30 = **N1 -> N20 -> N28 -> N30**
Path for N31 = **N1 -> N20 -> N28 -> N31**
Path for N32 = **N1 -> N20 -> N28 -> N32**
Path for N33 = **N1 -> N20 -> N28 -> N33**

From above observation the criticality of modules is:
N2 = N17
N13 > N3 = N24
N7 = N28
N14 = N15 = N16 = N18 = N19 = N21 > N4 = N5 = N6 = N8 = N9 = N10 = N11 = N12 = N23 = N25 = N26 = N27 = N29 = N30 = N31 = N32 = N33

Step 3:

Modules showing same value of C are kept in same build.

Distinction in the criticality of modules (**N2, N17), (N3, N24), (N7, N28), (N14, N15, N16, N18, N19, N21**) and **(N4, N5, N6, N8, N9, N10, N11, N12, N23, N25, N26, N27, N29, N30, N31, N32, N33)** cannot be made, following above rule the criticality of all the modules is:

N7 = N28 > N20 > N1 > N13 > N3 = N24 > N2 = N17 > N22 > N14 = N15 = N16 = N18 = N19 = N21 > N4 = N5 = N6 = N8 = N9 = N10 = N11 = N12 = N23 = N25 = N26 = N27 = N29 = N30 = N31 = N32 = N33

Step 4:

Module(s) are tested in decreasing order of their criticality factor values.

Using above rule, the builds are:
Build 1 = N7, N28
Build 2 = N20
Build 3 = N1
Build 4 = N13
Build 5 = N3, N24
Build 6 = N2, N17
Build 7 = N22
Build 8 = N14, N15, N16, N18, N19, N21
Build 9 = N4, N5, N6, N8, N9, N10, N11, N12, N23, N25, N26, N27, N29, N30, N31, N32, N33

The build testing can now be applied to these builds which tell the tester that the given system can be tested further or not, that is, whether the given system is of any use or not.

CONCLUSION

The above mentioned method for finding criticality factor of nodes (Software Modules) is very useful whenever it is difficult for the software tester to decide which modules are more critical on the basis of functional dependency of the modules. Using this methodology, tester can even find out the most critical and least critical modules as well.
This method can be vastly used by tester before beginning any test. For initialising the *Build Testing*, which depends on the criticality of different modules of the given system, this technique is very useful as shown in above examples.
It is found that testing the most critical node is always the first priority of every tester and hence this technique is very useful in every aspect.

REFERENCES:

[1] Paul C. Jorgensen, "Software Testing, A Craftsman's Approach ", Third Edition

[2] K. K. Aggarwal, Yogesh Singh, "Software Engineering", Third Edition

[3] Kan S H, Basili V R, Shapiro L N 1994 Software Quality : An overview from the perspective of total quality management

[4] Roge´rio Paulo, "Integration Testing", January 12, 2007

[5] V V S Sarma, D Vijay Rao, "A re-entrant line model for software product testing", 1997

[6] Praveen Ranjan Srivastava, Subrahmanyan Sankaran, Pushkar Pandey, Optimal Software Release Policy Approach Using Test Point Analysis and Module Prioritization, 2013

[7] http://www.cognizant.com/InsightsWhitepapers/Risk%20Based%20Testing.pdf

[8] https://en.wikipedia.org/wiki/Integration_testing

[9] https://en.wikipedia.org/wiki/Build_verification_test

[10] http://www.softwaretestingstuff.com/2008/06/build-verification-testing.html

[11] http://searchsoftwarequality.techtarget.com/definition/build

[12] https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0CBwQFjAAahUKEwj8w82KqrPIAhULBo4KHba5A98&url=http%3A%2F%2Fwww.istqb.org%2Fdownloads%2Ffinish%2F20%2F212.html&usg=AFQjCNEYHr045l6c8qE44Q66LFmQfhMFiw

IJERGS