Test Case Prioritization Using Aggregate Weight of the Independent Path

Harish Kumar, Naresh Chauhan

Department of Computer Engineering, YMCA University of Science & Technology, India
htanwar@gmail.com, Nareshchauhan19@gmail.com

Abstract
Software testing is considered an important phase for developing and maintaining any software. It controls the quality and reliability of the software being tested. The main objective of testing is to identify and eliminated bugs. Although it is a time consuming activity, the time spent is justified because it is used for generation of test cases and their testing. So, in order to avoid exhaustive testing, test cases are prioritized. In the past, many techniques of prioritization have been used to prioritize the test cases of white box testing, which fully covers the whole structural and functional part of software, as it covers the independent paths of all modules. But none of the techniques focused on the complexity of the statements covered by an independent path. So in this paper, the prioritization of test cases using basis path testing is presented, in which the most important independent path has been considered for testing. The importance of the independent path is calculated on the basis of the Complexity of the statements covered by that independent path.

Keywords: *Software, complexity, program, white box testing, language code*

INTRODUCTION

Software testing is a process which is basically carried out with the intent of finding errors [1]. It is done in a systematic manner in order to achieve the fullest potential of the software. Testing must be done by keeping in mind all the essential factors such as quality, reliability, integrity and efficiency. Designing of suitable and efficient test cases is a challenging task. The testing process has to be planned, scheduled, designed and prioritized. The need of prioritization is to meet the cost constraints, to minimize the test suites, and early detection of faults in order to maximize the objective function.

There are two main techniques of testing: white box testing and black box testing [2–4]. White box testing or structural testing is typically focused on the internal structure of the program [2–4]. In white box testing, structure means the logic of the program which has been implemented in the language code. The base of the path coverage is determined by basis path testing. This type of testing is the oldest structural testing technique, which is based on the control structure of the program. This control structure further uses the control flow graph to cover each possible path during testing so that each test case is executed efficiently.

Basis path testing is an important part of white box testing. It monitors the whole control structure of the program. Based on the control structure, a flow graph is prepared and all the possible paths are covered and executed during testing. It is considered the general criteria for detecting more errors as all statements and all branches are covered while testing. But the problem with this testing is that while all statements and all branches are covered, the critical points of the paths like loops, arrays, in-degree, out-degree etc. are not factored in. Therefore following important questions do not get answered:

1. Which path is complex and error prone?
2. What kind of statements a path has, eg. loops, arrays or pointer usage?
3. What should be the order of the test cases dedicated for differnet paths so that faults are detected as soon as possible?

So in this proposed technique, characteristics of each critical point has been considered and a formula has been proposed to detect which path is more critical, so that while testing the main focus is given to that path and hence, it will lead to saving of time, cost and effort.

Section 2 of the paper discusses the related work, Section 3 discusses the proposed approach for test case prioritization, Section 4 discussed analysis of the proposed work, and section 5 discusses the effectiveness of proposed approach. Section 6 concludes this paper in brief.

BASIS PATH TESTING

It is one of the oldest structural testing techniques that are based on the control structure of the program [3]. On the basis of that control structure, a flow graph is developed and it is assumed that all possible paths can be covered at least once during testing. Here, modified version of path coverage criterion is used which is the most general criterion when compared to other logic coverage criteria. The problem with path coverage is that a program that contains loops can have an infinite number of possible paths and it's impractical to test all those paths. Basis path testing is the testing technique of selecting the paths to provide a basis set of execution paths through the program.

An execution path is a set of nodes and directed edges in a flow graph that connects (in a directed fashion) the start node to a terminal node. Two execution paths are said to be independent if they do not include the same set of nodes and edges.

CYCLOMATIC COMPLEXITY

Cyclomatic complexity is the software metric that provides a quantitative measure of the logical complexity of a program [1, 2]. When used in the context of a basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in a basis set of a program, and provides an upper limit for the number of tests that must be conducted to ensure that all the statements have been executed at least once**.**

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined.

There can be different basis sets for a procedural design, so a measure called cyclomatic complexity is used to define the paths which need to be considered. The value of this provides us the upper limit for the number of independent paths that comprise the basis set.

RELATED WORK

Srivastava *et al.* proposed an approach for identification of effective paths in control flow graph for software under test and prioritized the most feasible path to be executed first using ant colony optimization algorithm [5].

Kumar *et al.* discussed the basis path testing as an imperative testing method in white box testing [6–8]. Basis path testing focused on internal logic; therefore it generates a feasible set of independent paths present in source code, which is known as basis path. Out of these paths, some may be not feasible.

Zhonglin *et al.* proposed an improved approach for basis path testing [6]. The proposed technique combines the baseline method with dependence relation analysis. The method proposed by them generated a set of linearly independent paths, termed as basis paths.

Qingfeng *et al.* elaborated the work proposed by Zhonglin for selection of infeasible paths [7]. In his research paper, he proposed a new approach for selection of independent paths. To show the effectiveness of his proposed approach, he illustrated his work on a triangle program.

Himanshi *et al.* proposed a technique to prioritize the paths using ant colony optimization [9]. The proposed approach allows tester to find out the probability for each path and priority of the shortest path comes out to be maximum.

Ahmed S. Ghiduk proposed an ant colony optimization based approach for generating a set of optimal paths to cover all definition-use associations in the program under test [10]. This approach uses the ant colony optimization to generate a suite of test-data for satisfying the generated set of paths. He also introduced a case study to illustrate his approach.

PROPOSED WORK

In this research work, the independent paths of the flow graph have been considered for test case prioritization. First of all, the independent paths are calculated from the flow graph. Now each node of the flow graph is characterized based on its criticality. After this the importance weight of each node of the graph is calculated. Finally the aggregate weight of the independent path is calculated. Then the paths and their corresponding test cases are prioritized based on the higher value of the aggregate weight. The basic flow of the proposed procedure is shown in Figure 1.

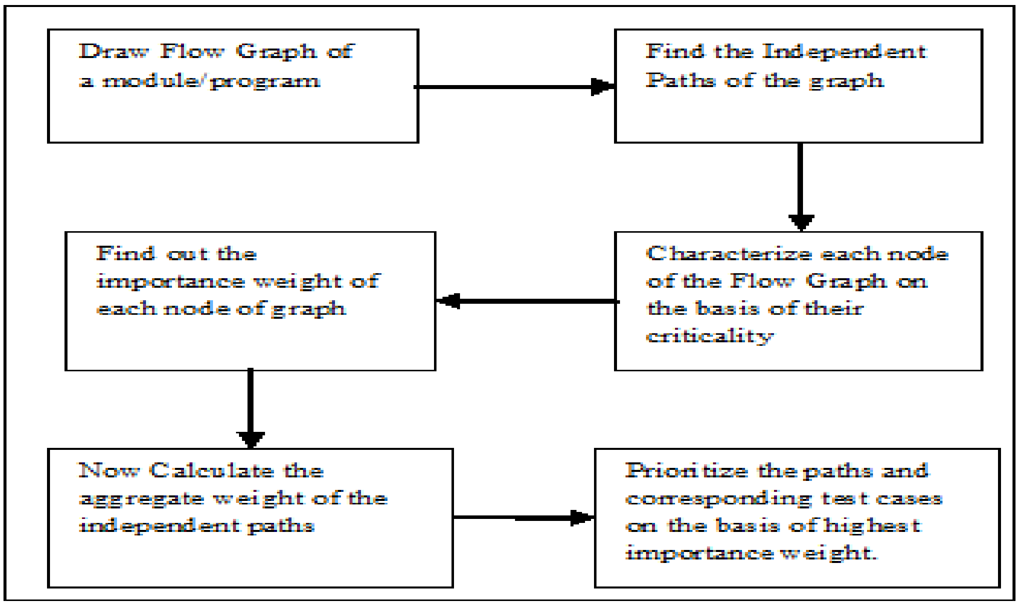
Draw Flow Graph of a module/program
Find the Independent Paths of the graph
Characterize each node of the Flow Graph on the basis of their criticality
Find out the importance weight of each node of graph
Now Calculate the aggregate weight of the independent paths
Prioritize the paths and corresponding test cases on the basis of highest importance weight.

Fig. 1: *Flow of Proposed Procedure.*

For finding out the weight of the independent path, six factors are proposed. They are:

1. Loop count
2. Array count
3. Predicate nodes
4. Pointers
5. In-degree of the node, which is the number of head endpoints adjacent to a node. The number of heads pointing inwards to a particular node is called the in-degree of the node.
6. Out-degree of the node, which is the number of tail endpoints adjacent to a node. It is called a branching factor in a tree. The number of nodes pointing outwards through a particular node is called the out-degree of that node.

These proposed factors are assigned an importance weight as shown in Table 1.

Table 1: *Importance Weight Factor Values.*

S. No.	Factors	Importance Weight
1	Loop	10
2	Predicate Node	5
3	Pointers	2
4	Arrays	1
5	In-Degree	2
6	Out-Degree	2

PROPOSED ALGORITHM

```
Find all independent paths (IP) from a flow graph.
{
Loop_count=0, predicate node_count=0, pointer_count=0, array_count=0; indegree_count=0, outdegree_count=0;
While (IP)
{
/*Find aggregate_importance_weight of the all the independent paths of the flow graph. */
For each (N) {  /*N=node*/
If N contains loop then loop_ count = loop_count++
If N contains predicate node then assign predicate_nodes_count = predicate_nodes_count++
If N contains pointer then assign pointer_count = pointer_count++
If N contains array then assign array_count = array_count++
Count in-degree of particular node and assign it to in-degree_count.
Count out-degree of particular node and assign it to out-degree_count.
    }
Aggregate_path_weight = loop_count * loop_weight + predicate node_count * predicate node weight
+ pointer_count * pointer weight + array_count * array weight + indegree_count * weight + outdegree_count * outdegree weight.
}
```

Now Prioritization of paths is decided on the basis of the aggregate_path_weight. Higher the value of the aggregate_path_weight, higher the priority assigned to a path. The test case corresponding to this path should be given a higher priority. So if prioritization is based on the proposed technique, the errors will be detected early.

ANALYSIS OF PROPOSED APPROACH

To analyze and validate the method proposed above, a sample program has been taken [2]. The control flow graph of the sample program is shown in Figure 2. The independent paths are calculated. The test cases are designed for these independent paths which are shown in Table 2. Then the importance weight of all independent paths has been calculated. Then the prioritization order has been provided to each path on the basis of highest aggregate weight of the paths. Further, to validate the prioritized test suite, the APFD (Average percentage of fault detection) metric has been taken. The validation has been demonstrated by comparing the non-prioritized test suite and the prioritized test suite with the help of APFD metric.

***Table 2:** Test Cases for the Sample Program.*

TEST ID	**INPUTS**					**PATH COVERED**	**EXPECTED OUTPUT**			
	FLAG1	FLAG2	SAV_CH	SAVING TYPE	AMOUNT		SAVING TYPE.	AMOUNT	TOTAL	OUTPUT MESSAGE
1	0	0	N	NIL	NIL	P1	NOTHING	0	0	
2	0	0	Y,N	NO READ	NIL	P2	GARBAGE	0	0	
3	1	0	Y	XYZ	NIL	P3	NOTHING	0	0	
4	1	0	Y	X$YZ	NIL	P4		0	0	SAVING TYPE CONTAIN ONLY CHARACTER
5	1	0	Y	XYZ	NIL	P5	XYZ	0	0	
6	0	1	Y	XY	NIL	P6	XY			PLEASE ENTER BETWEEN 3 TO 20 CHARACTERS
7	0	1	Y	NIL	-10	P7	NOTHING			AMOUNT CANNOT BE EQUAL OR LESS THAN 3
8	0	1	y	NIL	1200	P8	NOTHING	12000		

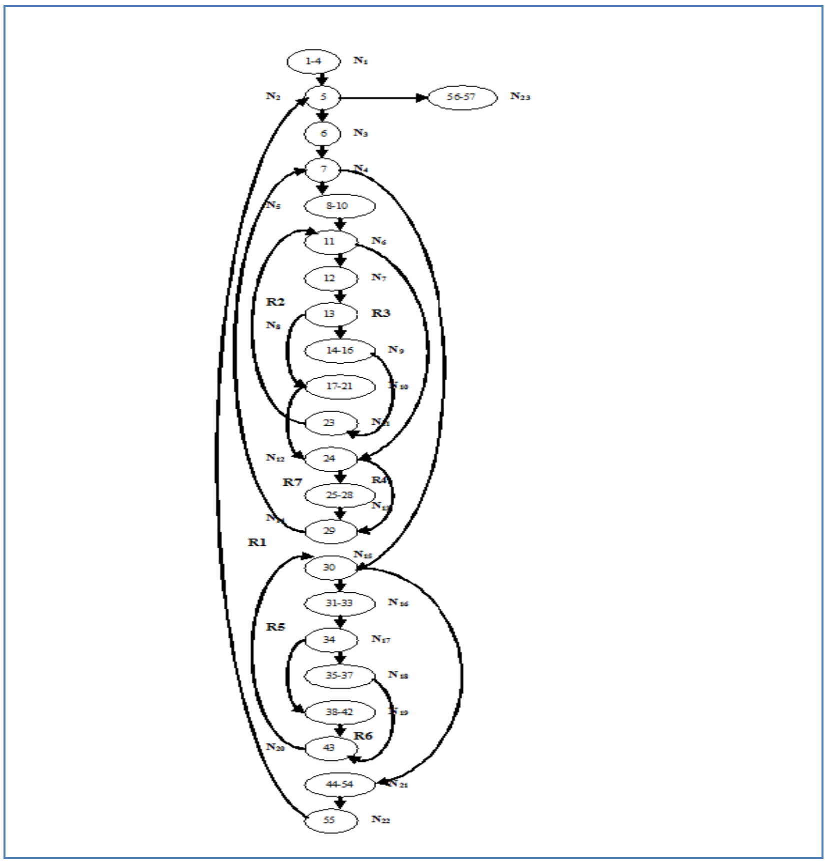

Fig. 2: *Control Flow Graph of the Sample Program.*

After analysing the sample program, the values obtained for six proposed factors for various nodes is shown in Table 3.

Table 3: *Proposed Factors Count for the Sample Program.*

Id	Node No.	Loop	Array	Predicate Node	Pointer	In-degree	Out-degree
1	1-4	0	0	0	0	0	1
2	5	1	0	0	0	2	2
3	6	0	0	0	0	1	1
4	7	1	0	0	0	2	2
5	8-10	0	1	0	0	1	1
6	11	1	1	0	0	2	2
7	12	0	0	0	0	1	1

8	13	0	1	1	0	1	2
9	14-16	0	0	0	0	1	1
10	17-21	0	0	0	0	1	1
11	23	0	0	0	0	0	2
12	24	0	1	1	0	2	2
13	25-28	0	0	0	0	1	1
14	29	0	0	0	0	2	1
15	30	1	0	0	0	2	2
16	31-33	0	0	0	0	1	1
17	34	0	0	1	0	1	2
18	35-37	0	0	0	0	1	1
19	38-42	0	0	0	0	1	1
20	43	0	0	0	0	2	1
21	44-54	0	1	0	0	1	1
22	55	0	0	0	0	1	1
23	56-57	0	0	0	0	1	0

Independent Paths are

1. (1-4)-5-23
2. 1-5-6-7-30-(44-54)-55-5-(56-57)
3. (1-4)-5-6-7-(8-10)-11-24-29-7-30-(44-54)-55-5-(56-57)
4. (1-4)-5-6-7-(8-10)-11-12-13-(17-21)-24-29-7-30-(44-54)-55-5-(56-57)
5. (1-4)-5-6-7-(8-10)-11-12-13-(14-16)-23-11-24-29-7-30-(44-54)-55-5-(56-57)
6. (1-4)-5-6-7-(8-10)-11-24-(25-28)-29-7-30-(44-54)-55-5-(56-57)
7. (1-4)-5-6-7-30-(31-33)-34-(38-42)-43-30-(44- 54)-55-5-(56-57)
8. (1-4)-5-6-7-30-(31-33)-34-(35-37)-43-30-(44- 54)-55-5-(56-57)

Importance Weight of Each Node

Node (1-4) weight = (1*2) = 2

Node 5 weight = (1*10) + (2*2) + (2*2) = 18

Node 6 weight = (1*2) + (1*2) =4

Node 7 weight = (1*10) + (2*2) + (2*2) = 18

Node (8-10) weight = (1*1) + (1*2) + (1*2) = 5

Node 11 weight = (1*10) + (1*1) + (2*2) + (2*2) = 19

Node 12 weight = (1*2) + (1*2) = 4

Node 13 weight= (1*1) + (1*5) + (1*2) + (2*2) = 12

Node (14-16) weight = (1*2) + (1*2) = 4

Node (17-21) weight = (1*2) + (1*2) = 4

Node 23 weight= (2*2) = 4

Node 24 weight= (1*1) + (1*5) + (2*2) + (2*2) = 14

Node (25-28) weight = (1*2) + (1*2) = 4

Node 29 weight= (2*2) + (1*2) = 6

Node 30 weight= (1*10) + (2*2) + (2*2) = 18

Node (31-33) weight= (1*2) + (1*2) = 4

Node 34 weight= (1*5) + (1*2) + (2*2) = 11

Node (35-37) weight= (1*2) + (1*2) = 4

Node (38-42) weight= (1*2) + (1*2) = 4

Node 43 weight = (2*2) + (1*2) = 6

Node (44-54) weight = (1*1) + (1*2) + (1*2) =5

Node 55 weight = (1*2) + (1*2) = 4

Node (56-57) weight = (1*2) = 2

Aggregate Weight of all Paths is

Path1-((1-4)-5-23) = 24

Path 2-((1-4)-5-6-7-30-(44-54)-55-5-(56-57)) = 89

Path 3-((1-4)-5-6-7-(8-10)-11-24-29-7-30-(44-54)-55-5-(56-57)) = 151

Path 4-((1-4)-5-6-7-(8-10)-11-12-13-(17-21)-24-29-7-30-(44-54)-55-5-(56-57)) = 171

Path 5-((1-4)-5-6-7-(8-10)-11-12-13-(14-16)-23-11-24-29-7-30-(44-54)-55-5-(56-57)) = 194

Path 6-((1-4)-5-6-7-(8-10)-11-24-(25-28)-29-7-30-(44-54)-55-5-(56-57)) = 155

Path 7-((1-4)-5-6-7-30-(31-33)-34-(38-42)-43-30-(44-54)-55-5-(56-57)) = 132

Path 8-((1-4)-5-6-7-30-(31-33)-34-(35-37)-43-30-(44-54)-55-5-(56-57)) = 132

So the path 5 has the highest aggregate weight, hence this path is highly critical. The chances of finding errors are more in this path as compared to other paths.

So the test case corresponding to this path should be given highest priority. So, finally the prioritized order of the test cases corresponding to their independent paths is: {TC5, TC4, TC6, TC3, TC7, TC8, TC2, TC1}.

EFFECTIVENESS OF THE PROPOSED APPROACH

To show the effectiveness of the proposed approach, the APFD is calculated for both a randomly chosen approach and the proposed approach. We have 8 faults and their corresponding test cases which are shown in Table 4.

***Table 4:** Test Case and Fault Map in Non-Prioritized Order.*

Fault Id	TC1	TC2	TC3	TC4	TC5	TC6	TC7	TC8
F1			*				*	
F2						*	*	
F3		*			*	*		
F4				*				
F5			*		*	*		
F6	*					*		*
F7					*			
F8			*					

APFD Calculation of Non-Prioritized Test Suite

Let Non-prioritized test suite be {T1, T2, T3, T4, T5, T6, T7, T8}

No. of test cases (N) = 8

No. of faults (M) = 8

APFD for non-prioritized test suite:

APFD = 1 – (3+6+2+4+3+1+5+3)/ (8*8) +1/ (2*8)

=0. 64

= 64%

Table 5 shows the mapping of test cases with the corresponding faults for their prioritized order.

Table 5: *Test Cases and Faults Mapping in Prioritized Order.*

Fault Id	TC5	TC4	TC6	TC3	TC7	TC8	TC2	TC1
F1				*	*			
F2			*		*			
F3	*		*				*	
F4		*						
F5	*		*	*				
F6			*			*		*
F7	*							
F8				*				

APFD Calculation for Prioritized test suite

The prioritized order of test suite is {T5, T4, T6, T3, T7, T8, T2, T1 }.

No. of test cases (N) =8

No. of faults (M) = 2

APFD = 1 – [(4+3+1+2+1+3+1+4) / (8*8)] + [1/(2*8)]

= 0.77

=77%

From the above calculations it is clear that prioritized test suite gives better APFD value as shown in Figure 3.

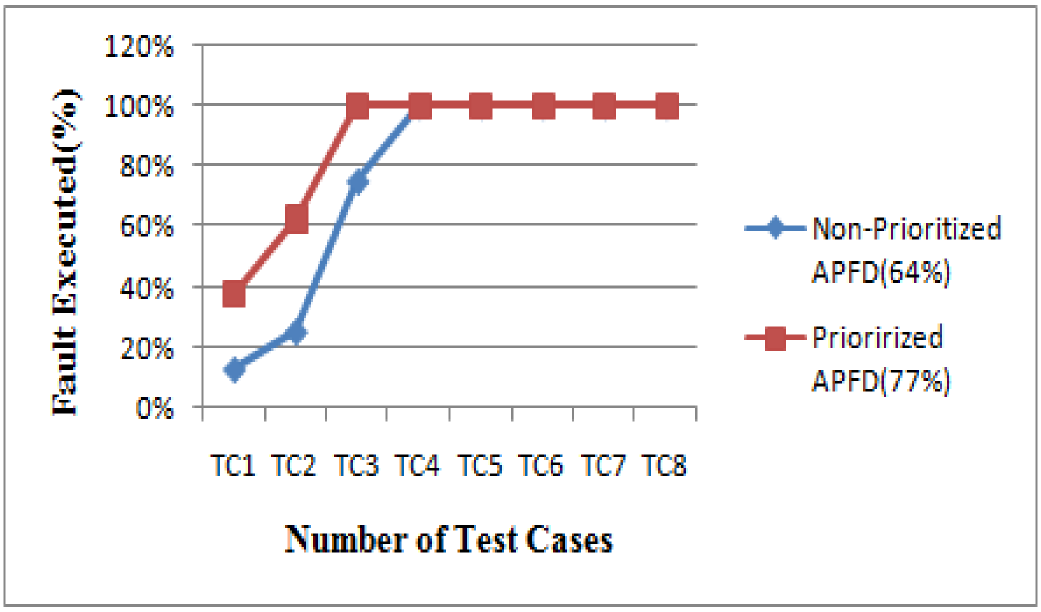

***Fig. 3:** Comparison of Non-Prioritized and Prioritized APFD Values.*

CONCLUSION

In this research paper a new technique for prioritizing test cases has been proposed. The proposed approach prioritizes the test cases based on the aggregate weight of the independent path of a program. For calculating the aggregate weight of the independent paths, six factors have been taken into consideration. To analyze the above method, a sample program has been taken. To validate the prioritized test suite, the APFD (average percentage of fault detection) metric has been taken. The validation has been demonstrated by comparing the randomized test suite and the prioritized test suite using APFD metric. It has been observed from the APFD values for both randomized and prioritized test suite that the proposed method is effective in identifying more bugs as compared to random order of test cases. Thus, the proposed method proves to be helpful in finding bugs early, thereby reducing the time, effort and cost required for the project.

REFERENCES

1. G.J. Myers, The Art of Software Testing, John Wiley & Sons, 1979.
2. Dr. Naresh Chauhan ,“Software Testing – Principle and Practice” , Oxford University Press, 2010.
3. T. McCabe, “A Complexity Measure”, IEEE Trans. On Software Engg., 1976; 308–320p.
4. Pankaj Jalote, “An Integrated Approach to Software Engineering”,

Narosa Publishing House, Second Edition, 2003.

5. Saurabh Srivastava, Sarvesh Kumar, Ajeet Kr. Verma, " Optimal path sequence in basis path testing", International Journal of Advanced Computational Engineering and Networking, ISSN(p): 2013; 1(1): 2320–2106p.
6. Zhang Zhonglin, M.L., "An Improved Method of Acquiring Basis Path for Software Testing", ICCSE'10. IEEE. 1891–1894p.
7. Qingfeng, D. "An improved algorithm for basis path testing", 2011IEEE. 175–178p.
8. T. Bharat Kumar, N.H., "A catholic and enhanced study on basis path testing to avoid infeasible paths in CFG". Global trends in information systems and software applications, 2012; 386–395p.
9. Himanshi, Nitin Umesh and Saurabh Srivastva, " Path Prioritization using Meta-Heuristic Approach", International Journal of Computer Applications(0975-8887), 2013; 77(11).
10. Ahmed S. Ghiduk, "A new software data-flow testing approach via ant colony algorithms", Universal Journal of Computer Science and Engineering Technology, 1(1), 2010; 64–72p. ISSN: 2219-2158.