International Journal of Innovative Computing, Information and Control
Volume **14**, Number **1**, February **2018**

ICIC International ©2018 ISSN 1349-4198
pp. **341–354**

TEST CASE PRIORITIZATION TECHNIQUE FOR OBJECT ORIENTED SOFTWARE USING METHOD COMPLEXITY

Vedpal[1] and Naresh Chauhan[2]

[1]Department of Information Technology and Computer Application
[2]Department of Computer Engineering
YMCA University of Science and Technology
NH-2, Sector-6, Mathura Road, Faridabad 121006, India
ved_ymca@yahoo.co.in; nareshchauhan19@gmail.com

Received June 2017; revised October 2017

Abstract. *Minimization of cost and time for testing the software is a challenging task for every software industry. Although the new technology and testing tools are being developed and used by the software industry, testing cost of the software is increasing rapidly. Further the customer expectation increases, the size of test suit also increases. So it is very difficult to execute the large test suite. In this paper, an algorithm for test case prioritization of object oriented software based on method complexity is presented. The method complexity is determined by using some factors which are identified by the structural analysis of the software. For experimental verification and validation the proposed approach has been applied on two case studies implemented in C++ and Java. The analyses of the results show the efficacy of the proposed approach.*

Keywords: Software testing, Test case prioritization, Method complexity, TCPOOS

1. Introduction.

Software testing is a process which exists at every stage during the development of the project. The software can be changed many times if the requirement is changed by the customer. Due to the advancement of the software technologies, customer's expectation also increases. Customer is always keen to add new requirement which may complicate the software resulting in the increase test suite.

There are many constraints on the software that affect the testing and quality of the software. These constraints are like time, allocated budget, resource limitation, use of new technology, and poor designing of the software architecture. For effective testing all the test cases should be executed but due to constraint it is very time and resource consuming and costly to execute each and every test case. To overcome these types of issues, the test suite must be prioritized in an order such that maximum bugs are detected earlier by executing the fewer test cases. Prioritization of test suite helps to perform the effective testing and deliver the quality product within time and allocated budget.

The prioritization of test case is performed on the basis of some of factors which direct to select most appropriate test case. The factors for prioritizing the test suit may be coverage of module, cost coverage per code component, past history data of the software, etc.

Complexity of a software shows how it is difficult to test and maintain the software. So to overcome complexity problem, software is divided in modules and methods. In software, a method is directly or indirectly interrelated to the other methods. The developers use many features of the language which increase the complexity of a method. With the multiple interfaces among the methods and to implement the complex feature, the complexity of software may go out of control resulting in unexpected results. So for the effective testing and quality of object oriented software, a method should possess the lowest complexity. There are various reasons that increase the complexity of the software. The complexity of a method can be measured by various structural factors that are used by the developer to implement functionality.

In this paper a test case prioritization technique for object oriented software is presented. By structural analysis of the object oriented programming system, some factors can be identified which are used to prioritize the test cases. Every factor has the capability to introduce the error if they are not used in an efficient way. Among all the considered factors, some factors are very critical such that if they introduce the errors, software may be get out of control. Moreover, every factor is also associated with a cost in terms of efficiency.

The presented approach prioritized the test cases on the basis of the method complexity. The overview of the presented approach is given below.

- The source code is converted into an intermediate representation called method call graph.
- The source code is analyzed and determines the count of the number of considered factors existing in all methods of software.
- Determine the volume and difficulty of each method which is further used to compute the complexity of the methods.
- All the feasible paths are identified after analyzing the method call graph.
- Path prioritization value of each path is calculated by using the complexity of the method being used in the each path.
- Test cases are prioritized on the basis of path prioritization value.

For experimental validation and analysis, the proposed system used to prioritize the test cases of two software is implemented in C++ and Java. The analysis of experimented result shows the considered factors are effective to select the most appropriate test cases resulting in detection of maximum bugs by executing the fewer test cases.

2. **Related Work.**

M. Shahid and S. Ibrahim [2] proposed an approach to prioritize the test cases. They prioritized the test cases based on the method coverage. More methods covered by a test case, the highest is the priority of execution of the test case and detect maximum faults as earlier.

M. Rava and W. M. N. W. Kadir [3] presented the review study of various types of technique to prioritize the test cases. They observed that all presented approaches have a common combination of coverage and faults detection. The primary concern of the prioritization technique is shifted from the code analysis to history based. By reviewing the work in area of test case prioritization, they also observed that the industry has adopted the artificial technique to prioritize the test cases rather than coverage based. However, as the size of program exceeds a certain amount, artificial technique drastically looses effectiveness.

S. Musa et al. [4] presented a regression test case prioritization technique for object oriented software. They used the dependency graph model to analyze the source code and to select the test cases. The selected test cases are optimized using the genetic algorithm. For experimental verification they applied the approach on software of vending machine. The result shows the effectiveness of the approach.

A. Marchetto et al. [5] presented a multi-objective technique that ordered the test cases to detect the maximum faults critical to business and technique. The proposed approach takes account of the coverage of source code, application requirement and cost to execute the test cases. They validated the approach by applying it on 21 Java applications and found it adequate.

A. A. Acharya et al. [6] presented a novel technique to prioritize the test cases. They determined business criticality value (BCV) of the functional and non functional requirements presented in the software. By using the fault model and BCV of functions the prioritization of test cases performed. They compared the proposed approach using APFD method and found that it detects the maximum faults as compared with the random test case prioritization.

M. Yoon et al. [7] proposed a technique to prioritized test cases through correlation of requirement and risk. They find out relevant test cases by calculating the risk exposure value of requirement and by analyzing risk items. The basic concept of the risk based testing is to have more focus on area of software which has higher risk exposure rather than other area. They also presented comparison of the proposed technique with the other prioritization techniques.

T. Muthusamy and K. Seetharaman [8] presented an algorithm to reorder the test cases to detect the maximum faults. They discussed prioritization algorithm based on four groups of weight factors. These factors are customer allotted priority, developer observed code related complexity, change in requirement, fault impact, completeness and traceability.

K. U. Maheshwar and S. Vasundra [9] used coarse grained technique to prioritize the test suits which is based on functional coverage. They focus on how much extent the test suites are dependent on each other.

N. Prakash and T. R. Rangaswamy [10] presented a coverage based test case prioritization technique. They used the statement, function, path, and branch and fault coverage as criteria to prioritize the test cases. The weight is evaluated for each test case using coverage information of considered criteria. They determined and used the average weight to prioritize the test cases.

S. Tahvili et al. [11] proposed a novel technique to prioritize the test cases. They combine the TOPSIS decision making with principal of fuzzy. The discussed method is based on many criteria such as probability fault detection, execution time and complexity. For evolution of efficiency of test cases they used the fault failure rate as an indicator to compare the capability of fault detection with the other set of test cases.

H. Sarikanth et al. [12] presented the study of prioritization of the test cases of build acceptance tests for an enterprise cloud application. Their prioritization process is based on the historical data of field failure. They found that the two or three interacting services have a tendency to be involved in the field failure. They also found that the efficiency of the testing is impacted by the order of build test acceptance, use of historical data to prioritize the build acceptance test and simple random heuristic works better than a fix order of non historical information.

J. A. Parejo et al. [13] presented a case study of multi-objective test case prioritization technique for highly configurable system. They address two limitations of test case prioritization technique for highly configurable system. The first one is that the current prioritization technique is driven by single objective and the second is that they used synthetic data to evaluate instead of industry strength case studies.

C. Hettiarachchi et al. [14] presented risk based test case prioritization technique. The risk related to the requirements is estimated by using the fuzzy expert system. From the result outcome it has been observed that the proposed approach can detect maximum faults earlier in highly risk components compared to other techniques.

J. Ding and X. Y. Zhang [15] compared the two test case prioritization techniques: adaptive random testing and dynamic random testing. They found that both techniques are extension of the random testing. ART is good for detection of failure whereas DRT is good at understanding the faults. Both the techniques used the different heuristics.

P. Saraswat et al. [16] used meta-heuristics techniques to optimize and prioritize the test cases. They comprised the genetic algorithm and particle swarm algorithm. Initially the generating algorithm generates the initial population randomly and genetic operators are applied on population. The output of the genetic algorithm is given to the particle swarm optimizer as input.

R. Haung et al. [17] presented an aggregate strength prioritization strategy for interaction test suite. The proposed technique combined the interaction coverage at different strengths whereas fixed strengths prioritization technique used the high coverage at fixed strength.

R. Khan and M. Amjad [18] proposed a structural testing technique to generate the test cases. For generating the test cases, a genetic algorithm is applied. The generating test cases cover its defuse associations. They used the K-means clustering algorithm to categorize the generated test cases in the different groups.

S. Mahajan et al. [19] presented a test case prioritization technique for component based software module level testing. They developed the component based software prioritization framework with the objective to detect the more extreme bugs at earlier stage and quality enhancement by using the genetic algorithm and java decoding technique. For prioritization they proposed prioritization keys which are project size, scope of the code, information stream, bug inclination and impact of bug and faults.

S. Nayak et al. [20] proposed a test case prioritization technique to improve the fault detection rate. They considered the four factors for prioritizing the test cases which are test case effectiveness, rate of fault detection, number of faults detected and test case ability of risk detection.

S. Ghai and S. Kaur [21] proposed a test case prioritization technique using hill climbing approach. They prioritized the test cases according to their functional importance. Functional importance is calculated using automated slicing.

W. Rahman and V. Saxena [22] proposed a model for prioritizing the test cases based on fuzzy logic. For capturing the behavior of the system, state diagram and risk information associated with the test cases are used. They classified the test cases in resettlement, reuse and obsoleteness.

A critical review of literature indicates that researchers use various factors like code coverage, method coverage, past history, genetic algorithm fuzzy expert system, and fault coverage to prioritize the test cases. The complexity of the method and various factors that are contributed to introducing in software are not addressed and used. In object oriented software there are various concepts if they are not used in efficient way they may become reason of severe faults in software which are also not considered to prioritize the test cases. So there is need to use some object oriented programming system (OOPS) related factors to detect the hidden faults and enhance the effectiveness of the prioritization process for detecting the maximum faults as earlier as possible. With this aim a new test case prioritization technique for object oriented software based on method complexity is presented in this paper. To determine the complexity of a method, some object oriented programming system related factors are proposed.

3. **Proposed Work.**

The software industries have developed various frameworks and software testing tools for effective testing within time and budget but still they are failed to cut the testing cost. Companies are spending the 33% of allocated budget to the activity related to the testing of software but it is expected to rise to a range of 41-50% by 2018 [23]. The main challenge in testing of object oriented software (OOS) is that there are thousands of thousand test cases which are not possible to execute all of them within constrained time and budget. So there is need of test case prioritization technique to organize the test cases in such order that maximum faults are discovered by consuming less time and cost. In the presented approach firstly source code is represented in the intermediate form called the method call graph (MCG) followed by the determination of the complexity of the each method used in the call graph. The complexity of method is calculated by using volume and difficulty of a method, which are further determined by the factors identified by the structural analysis of the source code. The factors which are used to determine the method complexity are given in Table 1.

Table 1. Considered factors and assigned weight

S. No	**Factor Name**	**Weight**
1	Degree of method (DM)	0.6
2	No. of input variable (IV)	0.3
3	Decision statement (DS)	0.4
4	Type casting (TC)	0.6
5	Numerical computations (NC)	0.4
6	Number of loops (LS)	0.5
7	Number of variables reused (VR)	0.2
8	Copying of objects (CO)	0.3
9	Object/Data reads from database/file (RW)	0.6
10	Exception handling (EH)	0.7
11	Virtual function (VF)	0.9
12	Dynamic memory allocation and deallocation (MA)	0.8
13	Reference counting (RC)	0.2
14	Proxy objects (PO)	0.3
15	Type binded inherited function (TIF)	0.8
16	Copy constructor having pointer type variable (CPV)	0.4
17	Non virtual destructor (NVD)	0.2
18	Return object by reference (RO)	0.2

Every considered factor has been assigned a factor weight which indicates the difficulty to test the factor and possesses the higher probability of the errors. For determination of the weight of considered factors a survey was performed in various industries (See Appendix). The survey was performed among developers, senior developer. Technology lead, associate architect group leader and project manager are with an average experience of seven years. From the survey approximate 80 responses were received from participants and the same data was compiled for determination of the assigned weight. The overview of the proposed approach is shown in Figure 1.

For process of prioritization of test cases, value of volume and difficulty of a method can be used. The determination of the value of the volume, difficulty and complexity of a method can be given as below.

Volume of a method is

$$V(m_i) = FM/FP \tag{1}$$

where FM is the number of considered factors in the i_{th} method and the FP is the total count of considered factors in the whole software, i.e., in whole method existing in the software.

Difficulty of a particular method can be calculated by Formula (2)

$$DM(m_i) = F_i * W_i \tag{2}$$

where F_i is the number of determined i_{th} factors in an i_{th} method and W_i is the weight assigned to the i_{th} factors.

Thus complexity (CM) of each method can be calculated by Formula (3)

$$CM(m_i) = VM * DM \tag{3}$$

where VM is the volume of the i_{th} method and DM is the estimated difficulty of the i_{th} method.

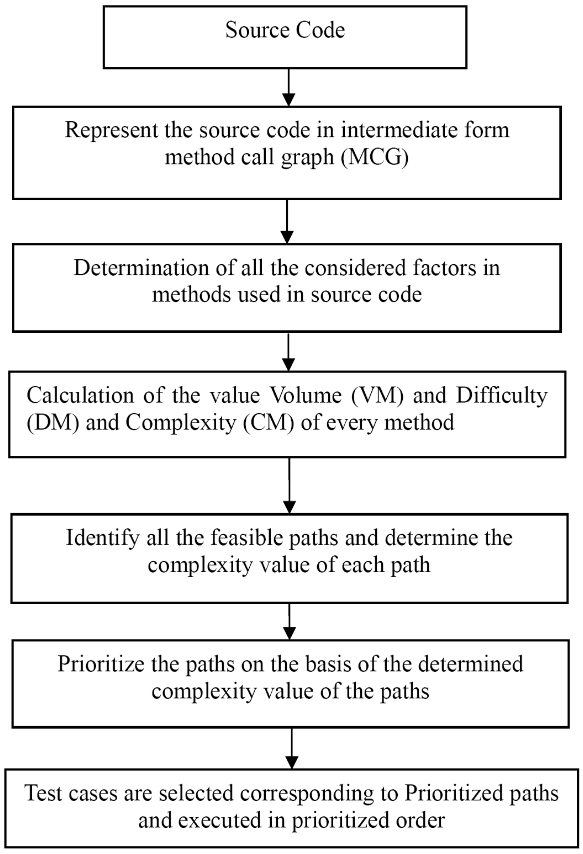
Source Code
Represent the source code in intermediate form method call graph (MCG)
Determination of all the considered factors in methods used in source code
Calculation of the value Volume (VM) and Difficulty (DM) and Complexity (CM) of every method
Identify all the feasible paths and determine the complexity value of each path
Prioritize the paths on the basis of the determined complexity value of the paths
Test cases are selected corresponding to Prioritized paths and executed in prioritized order

Figure 1. Overview of the proposed approach

After calculating the value of CM for all methods, all the feasible independent paths from method call graph are identified. The path prioritization value (PPV) is determined for each path which is sum of the calculated method complexity (CM) of methods that are used in the path. The PPV is calculated by Formula (4)

$$PPV = \sum_{i=1}^{n} CM_i \tag{4}$$

where CM is the complexity of the i_{th} method.

The more PPV value of the path, the more complexity of the path and having the higher chances of error. So paths are prioritized on the basis of the PPV. After path prioritization test cases are selected corresponding to each path and executed in the order of the paths. If any path has more than one test case, then these are prioritized on the basis of considered factors covered by the individual test case.

The algorithm of the proposed approach is shown in Algorithm 1. The presented algorithm takes the source code as an input and converts them into method call graph (MCG) by using Create_MCG function. The volume and difficulty of each method are calculated by identifying the considered factors in each method and using Formulas (1) and (2). The function Compute_complexity determined the method complexity by using the volume of a method and difficulty of a method. The MCG is used to identify all the feasible paths in the software and determine the methods covered in each path. The output of Compute_complexity is used to calculate the path prioritization value which is further used to prioritize the test cases.

```
Let S be a source code, T be the set of non prioritized test cases and T' be the
prioritized test cases.
1. Create_MCG (S)
Find out all the methods used in the source code and create the method call graph
(MCG) of source code
2. while (method)
Begin
            Determine the volume of each method using Formula (1)
End
3. While (method)
Begin
     Determine the difficulty of each method by using Formula (2)
End
4. While (method)
Begin
     Compute_complexity (VM, DM)
     Begin
            Find out the value of complexity of each method (CM) by using
            Formula (3)
     End
End
5. All the feasible independent method call paths are identified.
6. Determine the PPV of the each path using Formula (4).
7. Paths are prioritized on the basis of the determined value of PPV.
8. Test cases are selected corresponding to each path and T' be the set of prioritized
test cases.
```

Algorithm 1. Algorithm of the proposed approach

4. Result and Analysis.

For experimental verification and analysis, the presented approach has been applied on a billing management system [24] implemented in the C++ programming language. The considered software performs various functions like place order, create product, modified product, and delete product. For experimental verification, intentionally some errors are introduced in the software and introduced errors were discovered by applying the proposed approach. The finding of the case study is given below.

Figure 2 shows the method call graph of the considered case study. In this graph, all the methods that are used are connected by using the direction arrows which shows the sequence of the calling of the methods. By analyzing the sequence of the calling methods all the feasible independent paths and methods covered in each path are determined.

Table 2 shows the methods used in the software and the count of considered factors identified in each method. The value of volume, difficulty and complexity of each method after computation is also given in Table 2.

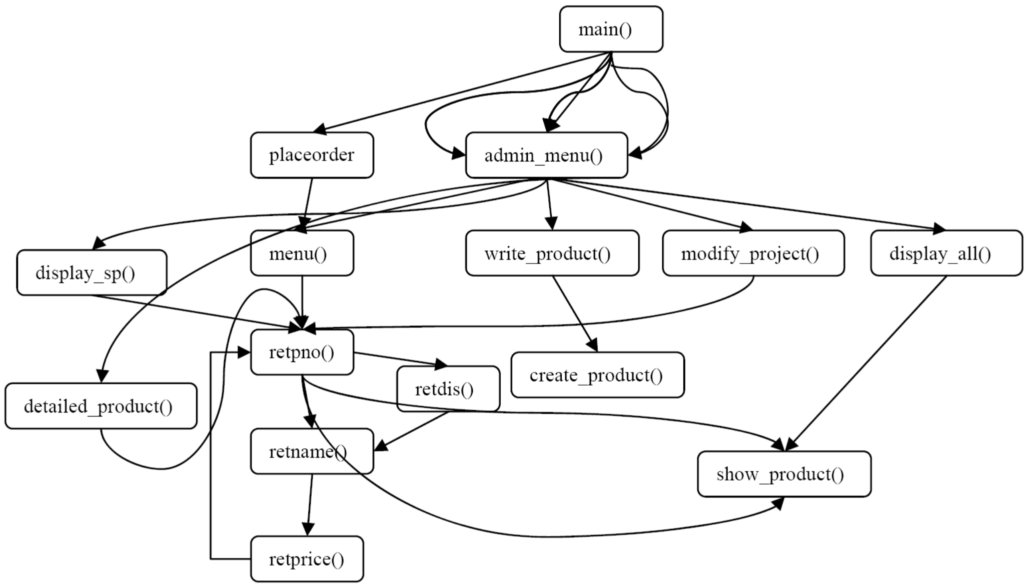

FIGURE 2. Method call graph (MCG) of the case study

TABLE 2. Determined value of VM, DM and CM

S. No.	**Method name**	**Factor determined**	**VM**	**DM**	**CM**
1	Place_order	IV = 7, LS = 3, NC = 1, RW = 2 NC = 3, VR = 1, DM = 2	$19/47 = 0.40$	$(7*0.3)+(3*0.5)$ $+(1*0.4)+(2*0.6)$ $+(3*0.5)+(1*0.2)$ $+(2*0.6) = 8.1$	3.24
2	Menu	RW = 1, DS = 1, LS = 01, DM = 3	$6/47 = 0.12$	3.3	0.39
3	admin_menu	IV = 2, CS = 1, DM = 12	$15/47 = 0.31$	9.2	2.8
4	write_product	RW = 1, DM = 2	$3/47 = 0.06$	1.8	0.10
5	create_product	IV = 4, DM = 1	$5/47 = 0.1$	1.4	0.14
6	modify_product	IV = 3, RW = 2, CS = 2, LS = 1, VR = 1, NC = 1, DM = 2	$12/47 = 0.25$	5.3	1.32
7	display_all	RW = 1, LS = 1, DM = 2	$4/47 = 0.08$	2.3	0.18
8	show_product	IV = 4, DM = 3	$7/47 = 0.14$	3.0	0.42
9	delete_product	IV = 1, RW = 2, LS = 1, CS = 1, DM = 2	$7/47 = 0.14$	3.6	0.50
10	display_sp	RW = 1, IV = 2, VR = 1, CS = 2, DM = 2	$8/47 = 0.17$	3.4	0.57
11	retpno	IV = 1, DM = 8	$9/47 = 0.19$	5.1	0.96
12	retname	IV = 1, DM = 3	$4/47 = 0.08$	2.0	0.16
13	retprice	IV = 1, DM = 2	$3/47 = 0.06$	1.5	0.09
14	retdis	IV = 1, DM = 2	$3/47 = 0.06$	1.5	0.09

Table 3. PPV of all feasible independent paths

S. No.	Path ID	Path	Estimated PPV
1	Path7	main(), palce_oreder(), menu(), retpno(), retname(), retprice(), retdis	$3.24 + 0.43 + 1.12 + 0.18 + 0.10 + 0.10 = 4.93$
2	Path1	main(), admin_menu(), write_product(), create_product()	3.04
3	Path2	main(), admin_menu(), display_all(), show_product()	3.4
4	Path3	main(), admin_menu, modify_product(), retpno(), show_product	5.5
5	Path4	main(), admin_menu, display_sp, retpno(), show_product()	4.75
6	Path5	main(), admin_menu, delete_product(), retpno()	4.26
7	Path6	main(), admin_menu, menu()retpno(), retname(), retprice()	4.4

Table 4. Paths covered by test cases

S. No.	Path ID	Test cases	Estimated path prioritization value (PPV)
1	Path3	T3, T4	5.5
2	Path7	T9, T10	4.93
3	Path4	T5, T6	4.75
4	Path6	T8	4.4
5	Path5	T7	4.26
6	Path2	T2	3.4
7	Path1	T1	3.04

Table 3 shows all the feasible and independent paths that are determined after analyzing the method call graph and the estimated path prioritization value of each identified path.

Table 4 shows estimated path prioritization value of each considered path obtained from MCG and the test cases that execute the identified independent paths.

Now the test cases are prioritized on the basis of estimated cost of paths that are executed by the test cases. The prioritized order of the test suit is T3, T4, T9, T10, T5, T6, T8, T7, T2, T1. Now the proposed approach is being compared with random approach and method coverage [2] based approach. For this purpose first we are detecting the faults.

Table 5 shows the faults detected by test cases when these test cases are executed in random order.

To show the efficacy of the approach, a metric called average percentage faults detection (APFD) has been used.

APFD value of random order of test cases is 54%.

Table 6 shows the faults detected by the test cases when these test cases are executed in prioritized order that has been obtained after applying the proposed approach.

The APFD value of prioritized order of test case by applying the proposed approach is 69%.

Table 7 shows the faults detected by the prioritized test cases obtained by applying the method coverage based approach.

Table 5. Fault detected by test cases in random order

Test case	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**	**F9**	**F10**	**F11**	**F12**	**F13**	**F14**	**F15**	**F16**
T1	*			*												
T2		*			*											
T3	*	*					*	*	*							
T4							*			*						
T5		*				*										
T6						*										
T7											*	*				
T8			*										*			
T9			*								*		*	*	*	*
T10													*	*	*	*

Table 6. Faults detected by test cases in prioritized order

Test case	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**	**F9**	**F10**	**F11**	**F12**	**F13**	**F14**	**F15**	**F16**
T3	*	*					*	*	*							
T4							*			*						
T9			*								*		*	*	*	*
T10													*	*	*	*
T5		*				*										
T6						*										
T8			*										*			
T7											*	*				
T2		*			*											
T1	*			*												

Table 7. Faults detected by ordered test cases obtained from method coverage based approach

Test case	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**	**F9**	**F10**	**F11**	**F12**	**F13**	**F14**	**F15**	**F16**
T9			*								*		*	*	*	*
T10													*	*	*	*
T8			*										*			
T3	*	*					*	*	*							
T4							*			*						
T5		*				*										
T6						*										
T1	*			*												
T2		*			*											
T7											*	*				

The APFD value of prioritized order of test case by applying the method coverage based approach is 65%.

Figure 3 shows the APFD graph of random approach, method coverage based approach and proposed approach showing the efficacy of the proposed approach.

The same approach was applied on another software room reservation [25] which performed all the operations related to reserve a room in hotel. The considered software has total 1936 line of code and 74 test cases are executed to detect the 56 faults, inserted intentionally. The APFD graph of the random approach, method coverage based approach and proposed approach is shown in Figure 4.

Figure 3 and Figure 4 show the APFD graph of the random approach and proposed approach. It has been observed from the results that the proposed technique has the higher rate of fault detection as compared to the random approach and method coverage approach. The graph shows that test cases in prioritized order detected the maximum

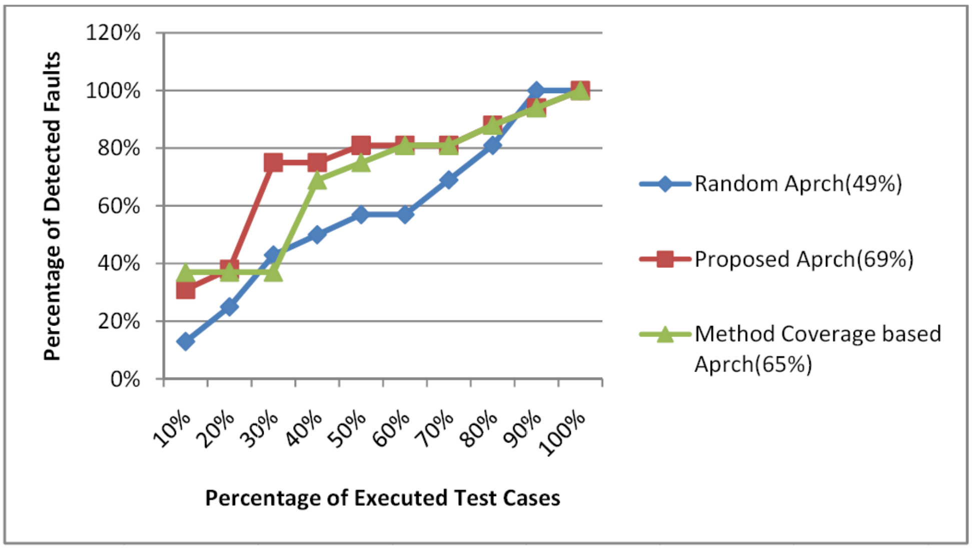

FIGURE 3. APFD graph of random approach, method coverage based approach and proposed approach

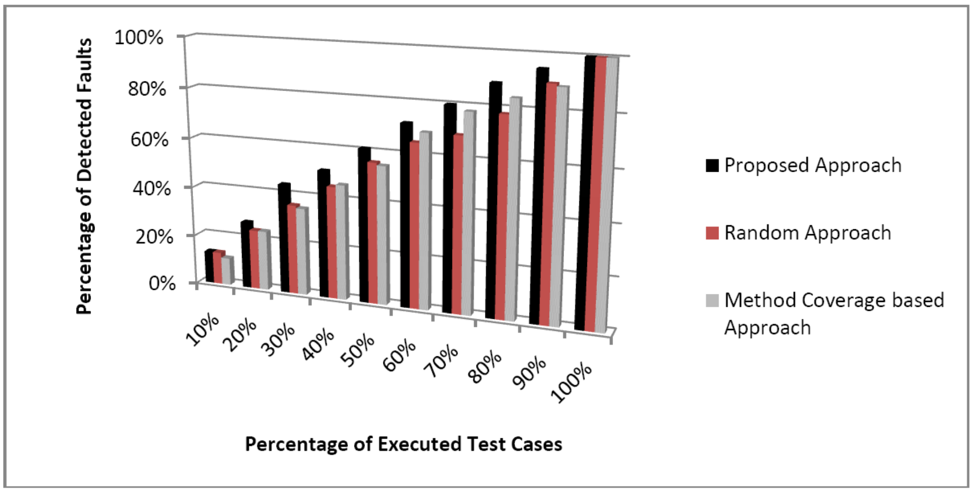

FIGURE 4. Comparison graph of random approach, method coverage based approach and proposed approach

faults by executing fewer test cases. The considered factors contribute efficiently to detecting the maximum faults by consuming less time, resource and cost.

5. **Conclusion.**

In the presented approach complexity of method is used to prioritize the test cases for object oriented software. For measuring the complexity of the method some parameters are used. Every parameter has been assigned a weight which shows the capability of introducing the error. The applicability of the parameters and their weight is assured by performing a survey in various reputed industries. After determination of the complexity of a method, all the feasible independent method call paths are determined. For these paths the source code is represented in intermediate form called method call graph. Further these paths are mapped with the test cases covering the determined paths. If a path has more than one test cases, then they are prioritized on the basis of covering the factors. For experimental verification the proposed approach was applied on two case studies implemented in C++ and Java and results are compared with the existing similar approach of object oriented software. The promising result shows that the achievement of the objectives meets the desired goal.

Threats and Limitations. The major threats and limitation for the proposed approach are given below.

(1) The considered case studies are not industry based projects and not having all considered factors.
(2) For analyzing the faults detection performance, the faults are introduced in the software manually.
(3) It is very difficult to create MCG manually to identify the considered factors in case of big projects.
(4) The considered approach is probabilistic in nature so the result may not be effective for every software whose test cases are prioritized.

Future Scope. The presented approach is manually validated which may not show the result very accurately, so a tool may be created with the aim to prioritize the test cases. In the presented approach some other non structural factors may be included which may increase the effectiveness of the presented approach, like risk, type of projects, and developers skill set.

REFERENCES

[1] M. V. D. Brink, D. Frasher and Y. Seynaeve, *World Quality Report 2015-2016*, Benelux, 2015.
[2] M. Shahid and S. Ibrahim, A new code based test case prioritization technique, *International Journal of Software Engineering and Its Applications*, vol.8, no.6, pp.31-38, 2014.
[3] M. Rava and W. M. N. W. Kadir, A review on prioritization techniques in regression testing, *International Journal of Software Engineering and Its Applications*, vol.10, no.1, pp.221-232, 2016.
[4] S. Musa, A. B. M. Sultan, A. A. B. A. Ghani and S. Baharom, Software regression test case prioritization for object-oriented programs using genetic algorithm with reduced-fitness severity, *Indian Journal of Science and Technology*, vol.8, no.30, 2015.
[5] A. Marchetto, M. M. Islam, W. Asghar, A. Susi and G. Scanniello, A multi-objective technique to prioritize test cases, *IEEE Trans. Software Engineering*, vol.42, no.10, pp.918-940, 2016.
[6] A. A. Acharya, S. Khandai and D. P. Mohapatra, A novel approach for test case prioritization using business criticality test value, *International Journal of Computer Application*, vol.46, no.15, 2012.
[7] M. Yoon, E. Lee, M. Song and B. Choi, A test case prioritization through correlation of requirement and risk, *Journal of Software Engineering and Applications*, vol.2012, no.5, pp.823-835, 2012.
[8] T. Muthusamy and K. Seetharaman, A new effective test case prioritization for regression testing based on prioritization algorithm, *International Journal of Applied Information Systems (IJAIS)*, vol.6, no.7, 2014.

[9] K. U. Maheshwar and S. Vasundra, Automated functional test case prioritization for increased rate of fault detection, *International Journal for Innovative Research in Science & Technology*, vol.1, no.7, 2014.

[10] N. Prakash and T. R. Rangaswamy, Weighted method for coverage based test case prioritization, *Journal of Theoretical and Applied Information Technology*, vol.56, no.2, 2013.

[11] S. Tahvili, W. Afzal, M. Saadatmand, M. Bohlin, D. Sundmark and S. Larsson, Towards earlier fault detection by value-driven prioritization of test cases using fuzzy TOPSIS, *The 13th International Conference on Information Technology: New Generations*, 2016.

[12] H. Sarikanth, M. Cashman and M. B. Cohen, Test case prioritization of build acceptance tests for an enterprise cloud application: Industrial case study, *The Journal of System and Software*, vol.119, pp.122-135, 2016.

[13] J. A. Parejo, A. B. Sanchez, S. Sagura, A. R. Corets, R. E. Lopez-Herrejon and A. Egyed, Multi-objective test case prioritization technique for highly configurable systems: A case study, *The Journal of System and Software*, vol.122, pp.287-310, 2016.

[14] C. Hettiarachchi, H. Do, B. Choi et al., Risk based test case prioritization using a fuzzy expert system, *Information and Software Engineering*, vol.69, pp.1-15, 2016.

[15] J. Ding and X. Y. Zhang, Comparison analysis of two test case prioritization approaches with the core idea of adaptive, *The 29th Chinese Control and Decision Conference (CCDC)*, 2017.

[16] P. Saraswat, A. Singhal et al., *A Hybrid Approach for Test Case Prioritization and Optimization Using Meta-Heuristics Techniques*, 2016.

[17] R. Haung, J. Chen, D. Towey, A. T. S. Chan and Y. Lu, Aggregate-strength interaction test suite prioritization, *The Journal of System and Software*, vol.99, pp.36-51, 2015.

[18] R. Khan and M. Amjad, Automatic test case generation of test cases for data flow test path using K-means clustering and genetic algorithm, *International Journal of Applied Engineering Research*, vol.11, no.1, pp.473-478, 2016.

[19] S. Mahajan, S. D. Joshi and V. Khanna, Component based software system test case prioritization with genetic algorithm decoding technique using java platform, *International Conference on Computing, Communication, Control and Automation*, 2015.

[20] S. Nayak, C. Kumar and S. Tripathi, Enhancing efficiency of the test case prioritization technique by improving the rate of fault detection, *Arab Journal of Science and Engineering*, 2017.

[21] S. Ghai and S. Kaur, A hill climbing approach for test case prioritization, *International Journal of Software Engineering and Its Applications*, vol.11, no.3, pp.13-20, 2017.

[22] W. Rahman and V. Saxena, Fuzzy expert system based test case prioritization from UML state machine diagram using risk information, *I.J. Mathematical Sciences and Computing*, vol.2017, no.1, pp.17-27, 2017.

[23] M. V. D. Brink, D. Fraser and Y. Seynaeve, *World Quality Report 2015-2016, Capgemini, Sogeti and HP*, https://www.sogeti.lu/globalassets/global/downloads/testing/wqr-2015-16/wqr-2015_country-pullouts_benelux_v1.pdf, 2015.

[24] *http://www.cppforschool.com/project/super-market-billing.html.*

[25] *https://github.com/.*

[26] A. Madi, O. K. Zein and S. Kadry, On the improvement of cyclomatic complexity metric, *International Journal of Software Engineering and Its Applications*, vol.7, no.2, 2013.

[27] N. Chauhan, *Software Testing Principal and Practice*, Oxford University Press, 2010.

[28] *http://oovcde.sourceforge.net/articles/Complexity.html.*

Appendix. Survey to check the viability of some factors and assigned weight. Test case prioritization is a process to order test cases with the intention of finding maximum faults as earlier as possible. Prioritization of the test cases is performed on the basis of some factors. In this survey some factors are considered to prioritize the test cases. Every considered factor has been assigned a positive weight within range of the 0 to 1 which shows the probability to introduce the error in the object oriented software if the developer did not use it in right way. So you are requested to assign a weight that suits to you on the basis of your experience. Table 8 shows the questionnaire of the survey.

The result of survey analysis is given in Figure 5. In the given figure weight ranges are represented by the slabs as given below

Slab1 = $0 \leq \text{Weight} < 0.3$

Slab2 = $0.3 \leq$ Weight < 0.5
Slab3 = $0.5 \leq$ Weight < 0.8
Slab4 = $0.8 \leq$ Weight <1
The weight is determined by calculating the mean average of the weight assigned by the participants.

Table 8. Questionnaire of performed survey

S. No.	Factor Name	Slab1	Slab2	Slab3	Slab4
1	Degree of method (DM)				
2	No. of input variable (IV)				
3	Decision statement (DS)				
4	Type casting (TC)				
5	Numerical computations (NC)				
6	Number of loops (LS)				
7	Number of variables reused (VR)				
8	Copying of objects (CO)				
9	Object/Data reads from database/file (RW)				
10	Exception handling (EH)				
11	Virtual function (VF)				
12	Dynamic memory allocation and deallocation (MA)				
13	Reference counting (RC)				
14	Proxy objects (PO)				
15	Type binded inherited function (TIF)				
16	Copy constructor having pointer type variable (CPV)				
17	Non virtual destructor (NVD)				
18	Return object by reference (RO)				

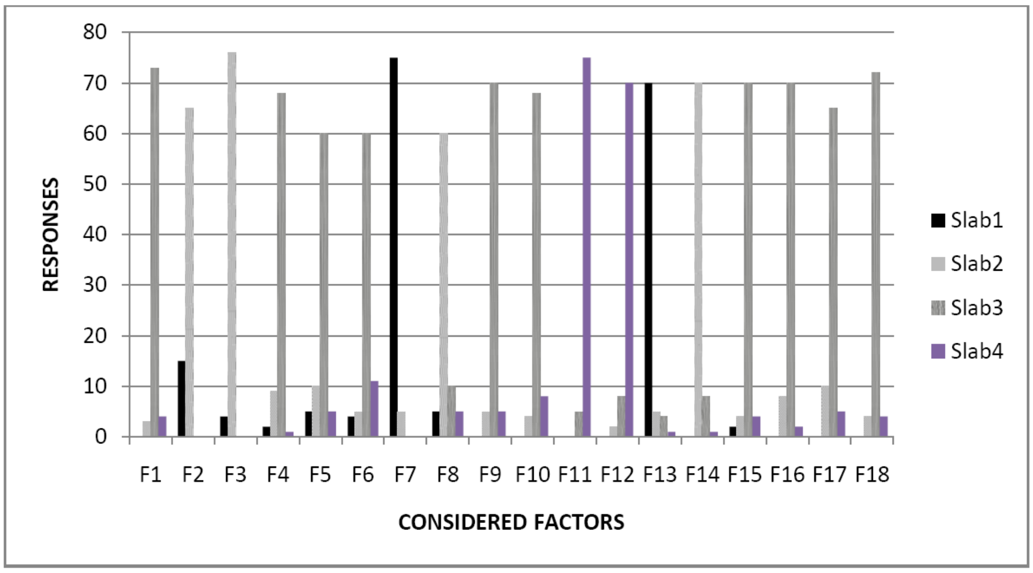

Figure 5. Analysis of feedback from participations