Hindawi
Scientific Programming
Volume 2021, Article ID 9988987, 13 pages
https://doi.org/10.1155/2021/9988987

Research Article

Multiobjective Test Case Prioritization Using Test Case Effectiveness: Multicriteria Scoring Method

Ali Samad,[1] Hairulnizam Bin Mahdin,[1] Rafaqat Kazmi,[2] Rosziati Ibrahim,[1] and Zirawani Baharum[3]

[1]*Faculty of Computer Science and Information Technology, Universiti Tun Hussein Onn Malaysia (UTHM), Parit Raja, 86400 Batu Pahat, Johor, Malaysia*
[2]*Faculty of Computing, The Islamia University of Bahawalpur, 63100 Bahawalpur, Pakistan*
[3]*Malaysian Institute of Industrial Technology, Universiti Kuala Lumpur, Persiaran Sinaran Ilmu, Bandar Seri Alam, 81750 Johor Bahru, Malaysia*

Correspondence should be addressed to Zirawani Baharum; zirawani@unikl.edu.my

Received 30 March 2021; Accepted 11 June 2021; Published 24 June 2021

Academic Editor: Sikandar Ali

Modified source code validation is done by regression testing. In regression testing, the time and resources are limited, in which we have to select the minimal test cases from test suites to reduce execution time. The test case minimization process deals with the optimization of the regression testing by removing redundant test cases or prioritizing the test cases. This study proposed a test case prioritization approach based on multiobjective particle swarm optimization (MOPSO) by considering minimum execution time, maximum fault detection ability, and maximum code coverage. The MOPSO algorithm is used for the prioritization of test cases with parameters including execution time, fault detection ability, and code coverage. Three datasets are selected to evaluate the proposed MOPSO technique including TreeDataStructure, JodaTime, and Triangle. The proposed MOPSO is compared with the no ordering, reverse ordering, and random ordering technique for evaluating the effectiveness. The higher values of results represent the more effectiveness and the efficiency of the proposed MOPSO as compared to other approaches for TreeDataStructure, JodaTime, and Triangle datasets. The result is presented to 100-index mode relevant from low to high values; after that, test cases are prioritized. The experiment is conducted on three open-source java applications and evaluated using metrics inclusiveness, precision, and size reduction of a matrix of the test suite. The results revealed that all scenarios performed well in acceptable mode, and the technique is 17% to 86% more effective in terms of inclusiveness, 33% to 85% more effective in terms of precision, and 17% minimum to 86% maximum in size reduction of metrics.

1. Introduction

Software testing is an important process to ensure correctness, completeness, and usefulness in user's specifications and requirements [1]. It is a process to assure that the software is free of fault [2]. There are different techniques in software testing and regression is one of them. Regression testing is a method that tracks new updates in the software and assures that the updates will not affect the existing software functions. In regression testing, one needs to confirm that all the changes that are made in the software have no conflict with the existing functionalities [3]. This testing guarantees that new changes made in that software will not affect previous operations, and all components will remain intact accurately without any unintended behavior. Regression testing is a common part of the software development process, and, in larger organizational environments, it is applied by code-testing or code-execution specialists [4].

However, performing all regression test cases needs a lot of time, cost, and resources because regression testing is a repetitive and expensive process, whereas it needs to be applied whenever a piece of code is modified where the complete test suite has to be reexecuted [4]. In a few cases, execution of a complete test suite is not practically possible due to restricted timeframe, budget, and resources [3]. To reduce cost of regression testing, it is important to make selection of a subset of test cases from a test suite that is potentially effective in identifying the bugs and faults. Here, the accurate selection of the subset of regression test cases can reduce maintenance cost. The typical processes involved in software maintenance phase are shown in Figure 1.

The process of regression testing is started with the first release of the product, so it is said to be a maintenance activity in software development process model. Any change request, software, or service update or next release of the same product leads to the regression testing. The change in requirement leads to the code modifications. After the source code is modified, regression testing is commenced; any regression error during regression testing is looped back to code modification. The regression errors include the bugs like software enhancement problems, configuration mismatches, substitution errors, structural failures in code, and service unavailability or failures. After the completion of regression testing, the new version of software has been released.

Moreover, test case selection technique eradicated repeated test cases from test suites [5], while test case prioritization technique sorts the test cases according to some adequacy measures results to reduce the testing costs and improve testing process efficiency, but TCP does not exclude test cases from actual test suite [6].

The main idea behind test case reduction methods is to minimize the test cases in the test suite, which results in increasing the cost of regression testing [7]. The objectives behind prioritization and selection are the same, except that prioritization methods orderly arrange test suites [8].

The literature broadly categorized the goals of regression test case prioritization methods into N-release development, continual quality enhancements, continuous development, and continuous integration. Both test suite minimization (TSM) and test case selection (TCS) cut short the original test suites [9], but TCP only schedules the test cases in test suites without eliminating any test case. It is possible that few of the test cases that are not participating in the testing at a particular version may participate and may play vital role in later versions of a software [10]. In other words, prioritization is more secure than permanent truncating, and regression test case prioritization is a strong, reliable, efficient, and cost-effective technique for regression testing [11]. Therefore, testers and engineers focus more on TCP than the test suite minimization (TSM) or test case selection (TCS).

Some factors like code/requirement information, risk management, software production, and metrics are used by the regression TCP methods to measure the results of testing process. This study is an extension of the previous research providing the justification for test case prioritization parameters cost (execution time), code coverage, and fault detection information as single objective for regression TCP [12]. The second study, which is part of the same research, provides the justification for cost trade-offs and discusses the different cost measures and their trade-off with respect to each other. The main objective of this controlled experiment is to prioritize the relevant subset of test cases from original test suites and minimize the cost of testing process and make it cost-effective and efficient, keeping cost (execution time), code coverage, and fault detection based effectiveness under control. The second objective is to establish a continuous test case prioritization process, which includes cost (execution time), code coverage, and fault detection, and the separate measure, as well as accumulative measure, for test case prioritization measures. The third objective is to embed the tester experience as part of test case prioritization process which provides the flexibility of choice in deciding parameters and may increase the effectiveness. It is concluded from the noteworthy work that it may be costly to execute or run an individual test case when some changes are done in the target code since a test suite may consist of a number of test cases, and each of the test cases takes long execution time. Therefore, there is a need of optimizing the number of test cases that are capable of capturing the potential faults and consume less execution time rather than the total cost.

The approaches for prioritization of test case are used to enhance the fault detection ability of the test cases for software under testing (SUT). The simpler way to explain the prioritization of test case is to reorder the test cases including the intent as to increase the fault rate or the potential faults detection as early as possible [13].

We prioritize the test cases reliant on their influence on business, crucial and common features, or testing team priorities. The test case selection selects the appropriate test cases that have captured maximum potential faults while covering the maximum number of code lines in minimum execution time, but it requires more time and resources for implementation. Rather than reexecuting the whole test suite, it is effective to choose part of test suite to be run. Categories for test case selection are obsolete, reusable, fault revealing, fault exposing, and change revealing test cases. The test case prioritization is done based on fault rates or code coverage requirements. But the main goal of regression testing is to lessen the cost with coverage of all the modifications done with source code of program under testing. These parameters are called prioritization parameters or adequacy parameters [14]. The test case prioritization with single objective of optimization like cost or fault or code coverage is recognized as a uniobjective prioritization of test suite. The prioritizing of test case based on two objectives for optimization like cost coverage or fault coverage or any combination of these three parameters is known as bicriteria prioritization. If prioritization uses two or more prioritization parameters, then it is also known as multicriteria prioritization of test cases. The main problem is that to combine these parameters is not simple due to their conflicting nature of measure. More coverage means more cost and probability of more faults. The objective of optimization problem in test case prioritization is not only to reduce cost but to increase the fault rates and reduce or maintain the coverage values as well.

The previous studies order the test suites on their fault detection ability and in some studies using fault rates with costs of test suites [5]. These studies ignore the code coverage and somehow other features of cost like test suite size and code size of the application under testing [6]. The single objective test case prioritization consisting of fault detection ability is incomplete due to ignoring the cost including

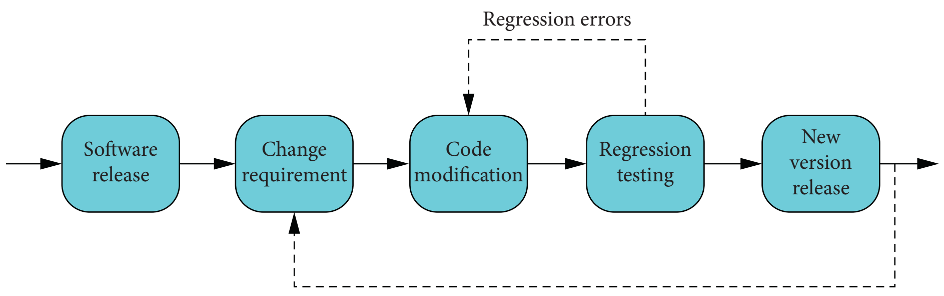


Figure 1: Maintenance process model [2].

execution time, size of test suites, size of the code, and code coverage. The factors cost and code coverage are necessary because coverage provides the confidence on testing process in terms of completeness, and cost reduction is the primary focus of regression testing [7, 8]. There is a gap in TCP for multiobjective test suite optimization including cost, fault detection, and code coverage. The models and frameworks like Junit [9] and Selenium [10] are not providing any mechanisms for prioritization of test cases corresponding to fault detection ability, cost, and coverage of test cases. Models and frameworks execute test cases in a simple sequence irrespective of their priorities. There is no clear winner from these popular testing models and frameworks too [11]. The other factors which may affect the accuracy of prioritization techniques are the size of the software under testing, size of the test suites available for testing, testing scenarios under these prioritization techniques, and testing environment supporting these prioritization techniques [12, 15, 16].

The limitations of the previous frameworks for unit testing with these factors impacting accuracy and usefulness of prioritization techniques put challenge in terms of multiobjective and multicriterion test suite prioritization research space [17]. It is also reported in comparison studies on single objective TCP with fault detection ability not providing any information about faults that are skipped due to incomplete coverage or cost as time constraints on overall testing cycles [6, 18, 19]. In multiobjective test, case prioritization techniques may incorporate dependency and impact of prioritization parameters, namely, cost, coverage, and fault detection ability on each other [20]. The multiobjective test suite prioritization is putting a challenge because the relationship between the parameters cost, fault detection [5], and coverage is not statistically defined. The dependency between conflicting variable cost, coverage, and fault detection ability has profound impact on accuracy and usefulness of prioritization techniques and frameworks. Ignoring any of these three parameters, cost, coverage, or fault detection ability, may produce incorrect testing results, ineffective testing techniques, or sometimes wastage of time and resources. To determine the relationship among fault detection ability, cost, and coverage with the intention of using them as prioritization scale becomes a challenge in prioritization context. To solve the problem of multiobjective particle swarm optimization and multicriteria test case prioritization, it has been used widely by researchers [21, 22].

In this work, the current major issue with PSO is the randomization factor in changing the particle position that is considered. In short, this research is focused on the three major issues of test case prioritization. First, it identifies the key factors affecting the primary objective to optimize the cost of regression testing. Second, this research focuses on finding the relationship between effectiveness parameters for prioritization of test case with multiobjective. Thirdly, an enhanced MOPSO is proposed for prioritization by considering multicriteria.

The rest of this paper is managed in six sections. The detailed discussion about the related studies is presented in Section 2. The proposed technique is elaborated in Section 3. In Section 4, we discussed the experimental setup for our proposed study. In Section 5, the results produced are presented. The last section concludes the paper.

2. Related Work

Software testing is a method that is performed to deliver the information to stakeholders related to the quality of the software service or product under the umbrella of test. The software testing contains a way toward executing application or program with the purpose of discovering bugs (defects or errors) and checking the fitness of product for use. Testing of software includes the execution of segment of software or part of system to assess at least one property of interest. As a rule, these properties demonstrate the degree to which a part or system is under the umbrella of test. The following are the vital goals of the SUT:

(1) Design and development guidance
(2) Response on all type of inputs
(3) Accomplishing task in acceptable time
(4) Usability of software
(5) Deployment and execution in planned environment
(6) Getting desired results

The regression testing is a process that confirms the changes in the code did not produce unexpected behaviors [5]. Regression testing also confirms that the previous functions of the software are working as per specifications [2]. The main concept behind regression testing is retesting of SUT with the aim to expose the faults earlier [23].

The regression testing practices increases in software development models like iterative development and component-based software engineering (CBSE). The regression testing is equally applicable in the development of testing paradigms such as continuous integration (CI) [24], continuous development (CD) [25], test-driven development (TDD) [26], and nightly build architectures [27].

On first release of some software, the system goes through a traditional testing cycle, and this means that there is no need to test the software by regression testing. On each new release or update in the software, the already tested functions need to be tested by regression testing to ensure that the previous functions are intact and not producing unintended behaviors or results. So, it is obvious that regression testing is required from version 2 to each next version of the software. The activity of software maintenance involves deletion of existing features, the error corrections, enhancements, and optimization. The system may work incorrectly due to modifications. Consequently, it is crucial to perform regression testing, and it can be performed by using the following techniques:

(1) Complete retesting
(2) Selection of test cases
(3) Prioritizing test cases
(4) Minimizing test cases

When test suite size increased to thousands or millions, then the testers need to adopt some methods to reduce or select or prioritize the test cases. Complete retesting is the method to reexecute complete test suite with all of test cases on each new version release or update to a SUT. But it is unwise and counterproductive in situations where time and costs are shirking. Practically, the retest-all is not possible with medium to large software having test cases in thousands and millions. Then, the obvious choices with a testing team are to select or prioritize or reduce the test suites with respect to some parameters such as code coverage, cost, fault detection ability, and code changes. These parameters are known with adequacy criteria in some cases [2] or regression testing parameters [28]. These parameters are used to select, prioritize, or reduce the test suites size to optimize time and cost required for testing purposed. The aim of techniques for test case selection is on code changes, and these techniques select the test cases containing a changed part of the SUT [29]. The prioritization of test case focuses on the fault detection as early as possible during testing software [30]. For the prioritization of test cases, methods try to increase the overall software's quality by identifying the faults on early stage of testing and decrease the cost indirectly by decreasing bug-fix time. The test suite reduction techniques are believed to eliminate the unused or nonproductive test cases from a test suite [31]. The reduction techniques are useful in cases where suite size does not matter, but they are criticized due to their permanent elimination of test cases of test suite. The primary focus of any testing technique is to expose the faults of SUT earlier [5]. So, the test case prioritization techniques got the attention of the testing research community [19, 32].

The test case prioritization execution and analysis have many challenges and issues as well. It is just not a simple reordering of the test cases. It requires deep insights and analysis procedures to get the fruits of prioritization methods. The four primary factors that affect the prioritization include the regression testing cost, code coverage by test case, fault detection ability, and modifications in code, which are made during the maintenance phase.

The cost of the testing includes a number of different types of the cost like analysis cost, test suite size, time for executing test case, and budget needed for testing [33]. The code coverage also has many different types and impacts on prioritization techniques. The coverage of source code for the test case is simply defined as the percentage of how many source code lines are targeted by the test case to total number lines present in module under testing [34]. The important types of coverage are statement coverage, modified statement coverage, block coverage, condition coverage, modified condition coverage, test coverage, modified block coverage, loop coverage, and many more [2]. The fault detection ability is measuring how many faults a test case detects and is also determined in terms of fault rates, fault frequency, and fault severity. But the fault types mentioned in the literature for experiments are real faults, structural faults, hand-seeded faults, and mutation faults [35].

The TCP is an essential part of regression test optimization (RTO) [18, 36], and one of the most important objectives is to expand fault recognition rate which is characterized as the rate at which a test suite can discover faults inside the testing cycle. As such, the effect of this cycle can offer and prioritize the reaction from the system as it is going through testing, and when testing cycle is halted for any reason, because of early debugging, the prioritization will empower the execution of the most basic test cases at earlier stage [37]. Basically, the test cases are prioritized as indicated by their need, which is resolved by different criteria, relying upon the strategy utilized, and executed based on priority [38]. Coverage is an essential part of software testing. The test coverage is used in regression testing and maintenance to measure the effectiveness of test suite [39]. The coverage is utilized as the intermediary for test suites assessment and stopping criteria for the testing process. Coverage is defined as the degree to which a test suite can execute the code under testing [2]. The bigger value of coverage means more reliability and confidence in the testing process. Table 1 depicts the summary of noteworthy articles published associated with test case prioritization.

Fault detection ability is the ability or probability for a test suite to detect the faults or errors from a program under testing [40]. The fault detection ability is used as adequacy measure for testing methods. It is also used to compare the different techniques with each other, and the technique is considered better by comparing techniques with better fault detection ability or fault rates [43]. The fault detection is the primary objective for every testing methodology. The regression testing uses fault detection ability and fault rates for assessment of the methods to prioritize the test suites to reduce cost of testing. The software faults are the occurrence of some instances due to the violation of rules for creating

Table 1: Summary of the related noteworthy articles.

Reference	Title	Year
[37]	The collaborative filtering recommender system for TCP	2018
[38]	Similarity-based product prioritization for effective product-line testing	2019
[40]	Under different testing profiles, the effect of code coverage on fault detection	2005
[41]	Selecting a cost-effective test case prioritization technique	2004
[42]	Hybrid PSO comparison with ant colony through selection of test cases	2018

codes and misunderstanding in specification or design models. The fault-based test suite prioritization was started by Rothermel et al. [44]. The ability of test case to uncover an error is called fault detection ability or fault rate of that test case or test suite. The probability of finding an error by a test suite is determined by the chance that a statement is executed by that test case. The extended study [41] uses six methods for level coverage to prioritize the test suites for fault detection ability. These prioritization methods use FEP to index the fault levels with their criticality and fault proneness for the respective method of the source code under analysis. The process of fault indexing in these studies deals with combining the method coverage and damage reports produced during previous test cycles. The term total fault indexing was used to represent the fault severities. The statistical analysis provides enough data to evaluate the results with APFD metric suites. The studies concluded that method coverage with FEP was more effective as compared to statement coverage-based techniques. The dark side of these techniques is fault indexing techniques due to old data, which was produced during previous test cycles.

The study proposed a time-aware test case prioritization method with GA [45]. The proposed method was evaluated by a controlled experiment with different program granularity levels; Emma was used to track the coverage for statement coverage, block coverage, and condition coverage. The results were assessed by APFD and claim that 120% effectiveness was observed on nonprioritization methods. The technique was also compared with reverse ordering prioritization and fault aware prioritization. The study also proposed GA-based test suite prioritization [46, 47] for Java software. The study introduces a large class of mutation, crossover, and selection operators on eight different software devices. The effectiveness was measured as coverage-based test suite prioritization method. The results were compared with a random search-based algorithm and hill climbing algorithm. A PSO was proposed in [42] for regression testing of modified units of software for real time embedded system. The automated prioritization of test case in regression testing helps select higher priority test cases. The PSO is effectively applied to the prioritization issue by accepting solution as space of particle and position for test cases according to software unit. The output shows that the PSO prioritizes test cases of the test suites by new best positions successfully effectively. PSO is a population-based stochastic optimization technique proposed by Dong et al. [48]. It is used to investigate the given search space to produce the optimal solution of declared problem. The search space comprises n particles, and the collection of these particles is known as swarm. PSO searches for solution with the help of some parameters. Population of particles is initialized randomly, and the search for a solution is done by updating particles position and velocity. Each particle has memory to store its position p-best, and the best position among whole particles is known as g-best. Position is updated by adding velocity in previous position. Velocity is constrained by V-max, to ensure that particles will search for optimal solution in defined search space.

3. Proposed Framework for Test Case Prioritization Based on Multicriteria Effectiveness Average Score

For test case prioritization, two test suites are generated and used for assessment of MOPSO by using the metrics code coverage, execution time, and ability of fault detection. The proposed technique is illustrated in Figure 2. Multiobjective particle swarm optimization (MOPSO) is used with coverage, cost, and fault detection as objective parameters. The particle position is specified as decimal vectors, which denotes the subset of test cases for testing process. According to assumption $T = \{T_1, T_2, \ldots, T_k\}$ is the test suite consisting of k test cases, the position of particle is given as $pt = \{t_1, t_2, \ldots, t_m\}$ where t_j belongs to set $\{0, 1\}$. The absence of the test case represented by 0 and 1 shows the presence of Ti in subset of test cases. Test cases are converted to binary form. The PSO has bird flock-based behavior in searching for food. The food searching process is done by passing the experience related to searching among neighbor birds. In order to let the optimal solution, the bird's flock concept is used by PSO. In PSO, every particle iteration search space tries to converge in the direction of the global optimal solution by observing the behavior of the neighbor particle. The best previous location of any particle can be traced by every other particle denoted by p_best and computed by a function named fitness function. The global best location in all particles is denoted by g_best, while the velocity (execution speed) of each particle can be computed by

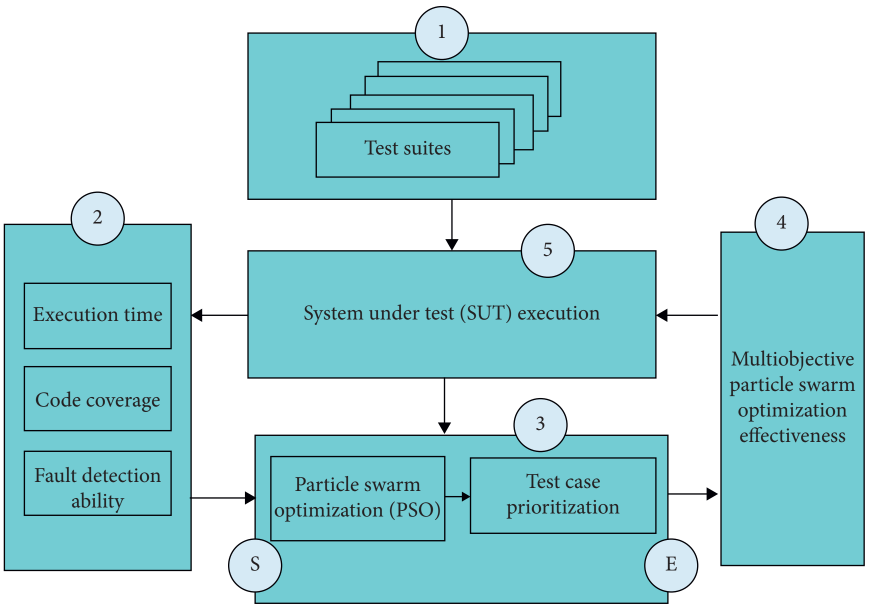

1
Test suites
2
Execution time
Code coverage
Fault detection ability
5
System under test (SUT) execution
4
Multiobjective particle swarm optimization effectiveness
3
Particle swarm optimization (PSO)
Test case prioritization
S
E

Figure 2: The experimental framework.

$$\mathrm{vol}(g+1) = w \times \mathrm{vol}_i(g) + c_1 \times r_{1i} \times (p\_\mathrm{best}_i(g) - x_i(g)) + c_2 \times r_{2i} \times (g\_\mathrm{best}_i(g) - x_i(g)). \quad (1)$$

In the above equation, $\mathrm{vol}_i(g)$ shows the velocity of ith particle at gth generation and previous best value $p\_\mathrm{best}_i(g)$ and global best value g_best value of population. c_1 and c_2 show the cognitive and social components. w is the inertia factor, and r_{1i} and r_{2i} are random numbers of interval $\{0, 1\}$. Our proposed MOPSO algorithm tries to optimize multiple objectives at the same time. The effectiveness of the proposed approach is computed in experimentation, and the results achieved demonstrate higher effectiveness.

4. Experimental Setup

Experimental setup for this study is discussed in this section. Experimentation is performed by using the system with the following specifications: Intel Core i7 processor with 8 GB RAM, and MATLAB 2019 is for simulation. For implementation of the proposed MOPSO, TreeDataStructure [49], JodaTime [50], and Triangle [49] datasets are selected. TreeDataStructure and Triangle are academic datasets with Java written test suites, and JodaTime is a replacement of Java date and time library. Selected datasets with details are given in Table 2, including dataset name, version, line of code (LOC), and number of test cases for each dataset for test case prioritization process.

The experimental flow for our proposed approach for test case prioritization is illustrated in Figure 2. In the first step, the test suites and the system under test (SUT) are selected. The eclipse environment [51] is selected for Java platform. In order to perform the unit testing, JUnit framework is selected. In the second step, the information required for our proposed approach is collected using EclEmma [39]. It collects the coverage information and size of SUT. The size of the test suite is computed through JUnit. All the versions of the SUT with faults are generated and analyzed. In third step, the data is taken from the second phase that is comprised of the test suite size, fault detection ratio, the execution time, and the code coverage relevant information. This information is then preprocessed to assign the priority to each test case based on the ability of capturing maximum faults in minimum time.

The metrics for data visualization, data analysis, and the data collection include the code coverage, fault detection, and the execution time of the test case. The code coverage is considered as covering statement for unit test case, and time is computed as execution time for every test case divided by total execution time of test suite, and then the result is multiplied by 100. The fault detection ability is computed by the number of faults captured by each test case of test suites.

5. Results and Discussion

The details about the test suite used in our experiment are given in Tables 3 and 4. For PSO, the particle size and maximum iterations are set to 10. The lower bound is 1, and the upper bound is n for particles, where n represents the test case quantity. The results produced by our proposed approach are evaluated by comparing no ordering, random ordering, and reverse ordering for the selected datasets.

For test suite 1, the relation of test cases and faults is presented in Table 5. A total of 8 test cases are included in test suite 1, namely, TC1–TC8, and 10 potential faults were introduced as F1–F10. In Table 3, the binary value 1 shows that the fault is detected by the respective test case, while 0 shows the absence of fault (Table 3).

The execution time is the time taken by test case of test suite. Due to the involvement of random parameter in

Table 2: The details about dataset.

Number	Dataset	Version	LOC	Test case
1	JodaTime	2	280464	279
2	TreeDataStructure	2	2200	22
3	Triangle	2	116	12

Table 3: Test suite 1 binary form of detected faults and time for execution of test cases.

Test cases	Binary form	Execution time
TC1	1010110010	6
TC2	0101000101	4
TC3	0100101001	4
TC4	1001010110	3
TC5	1010010001	4
TC6	0101100100	5
TC7	0000101100	4
TC8	1010110001	6

Table 4: Test suite 2 representation with detected faults in binary form and time.

Test cases	Binary form	Execution time
TC1	1011111001	8
TC2	0111001110	4
TC3	1000101001	6
TC4	0101010010	6
TC5	0110101001	4
TC6	0101010100	5
TC7	1010101001	4
TC8	1001001010	3
TC9	1110101001	3
TC10	1010010101	2

Table 5: Test suite 1 for datasets: test case and fault representation.

Test case	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10
TC1	1	0	1	0	1	1	0	0	1	0
TC2	0	1	0	1	0	0	0	1	0	1
TC3	0	1	0	0	1	0	1	0	0	1
TC4	1	0	0	1	0	1	0	1	1	0
TC5	1	0	1	0	0	1	0	0	0	1
TC6	0	1	0	1	1	0	0	1	0	0
TC7	0	0	0	0	1	0	1	1	0	0
TC8	1	0	1	0	1	1	0	0	0	1

MOPSO, the average results are computed for over 100 executions of algorithm. For performance evaluation, the parameters selected are the execution time, the average percentage of fault detection (APFD), and the code coverage. APFD is computed using the following equation:

$$\text{APFD} = 1 - \frac{T\text{suite}Flt_1 + T\text{suite}Flt_2 + T\text{suite}Flt_3 + \cdots + T\text{suite}Flt_j}{kj} + \frac{1}{2k}. \quad (2)$$

The minimum test case set for no ordering approach is given in Table 6. The representation includes the test case, its binary form of the faults detected, and the execution time (in hours). In the reduced test set, TC1, TC2, TC3, TC4, and TC5 are the selected test cases. The maximum execution time of 6 hours is observed for test suite 1.

The fault coverage for no order technique for three selected datasets is illustrated in Figure 3. The fault detection rate for test suite 1 for TreeDataStructure, JodaTime, and Triangle is shown in Figures 3(a)–3(c), respectively, in which x-axis represents the test cases and the fault coverage is presented in the y-axis. The fault coverage is given by the

Table 6: No ordering approach test case set with minimum test cases.

Test cases	Binary form	Execution time
TC1	1010110010	6
TC2	0101000101	4
TC3	0100101001	4
TC4	1001010110	3
TC5	1010010001	4

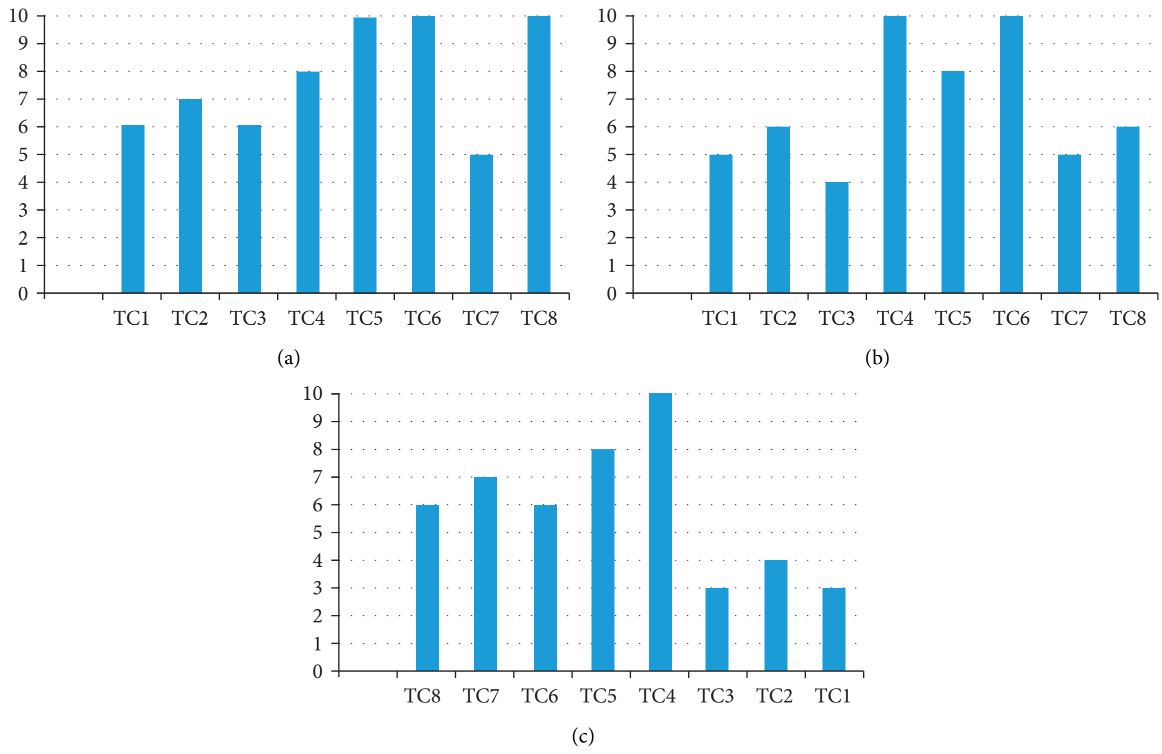


Figure 3: Fault coverage representation for no order approach. (a) TreeDataStructure no order. (b) JodaTime no order. (c) Triangle no order.

percentage with each test case. For Triangle dataset, no order shows 51% fault detection as compared to the TreeDataStructure with 68% and JodaTime with 60% fault detection. The higher detection rate is achieved by the no order approach for TreeDataStructure dataset.

For random ordering approach, the representation of detected faults, minimum selected test cases, and execution time is presented in Table 7. In test case set, the test cases TC1, TC5, TC3, TC8, and TC4 are selected. Minimum execution time for random order is 3 hours.

Test case set with minimum test cases for reverse ordering approach is given in Table 8. The selected test case with binary form and time for executing each test case is presented. The selected test cases in reverse ordering approach are TC8, TC7, TC6, TC5, and TC4.

In the reverse order approach, the fault coverage for TreeDataStructure, JodaTime, and Triangle is presented in Figures 4(a)–4(c). The 66% fault detection is captured for the TreeDataStructure by reverse order technique as visualized in Figure 4(a) and 55% for JodaTime dataset as shown in Figure 4(b). On Triangle dataset execution, reverse order approach shows 43% fault detection, which is lower than that of TreeDataStructure and JodaTime as shown in Figure 4.

Table 7: Random ordering technique test case set with minimum test cases.

Test cases	Binary form	Execution time
TC1	1010110010	6
TC5	1010010001	4
TC3	0100101001	4
TC8	1010110001	6
TC4	1001010110	3

Details of minimum test case set for proposed MOPSO are given in Table 9 which includes the test case, binary form, execution time, and priority. The details of experimentation are presented in Tables 3 and 5–9 for test suite 1.

The proposed MOPSO approach covered area is higher than that of no, reverse, and random ordering as visualized in Figure 5. For TreeDataStructure, fault detection of the proposed MOPSO is 77%, as shown in Figure 6(a). JodaTime fault detection is 81%, and for Triangle dataset, the proposed

Table 8: Reverse ordering technique test case set with minimum test case.

Test cases	Binary form	Execution time
TC8	1010110001	6
TC7	0000101100	4
TC6	0101100100	5
TC5	1010010001	4
TC4	1001010110	3

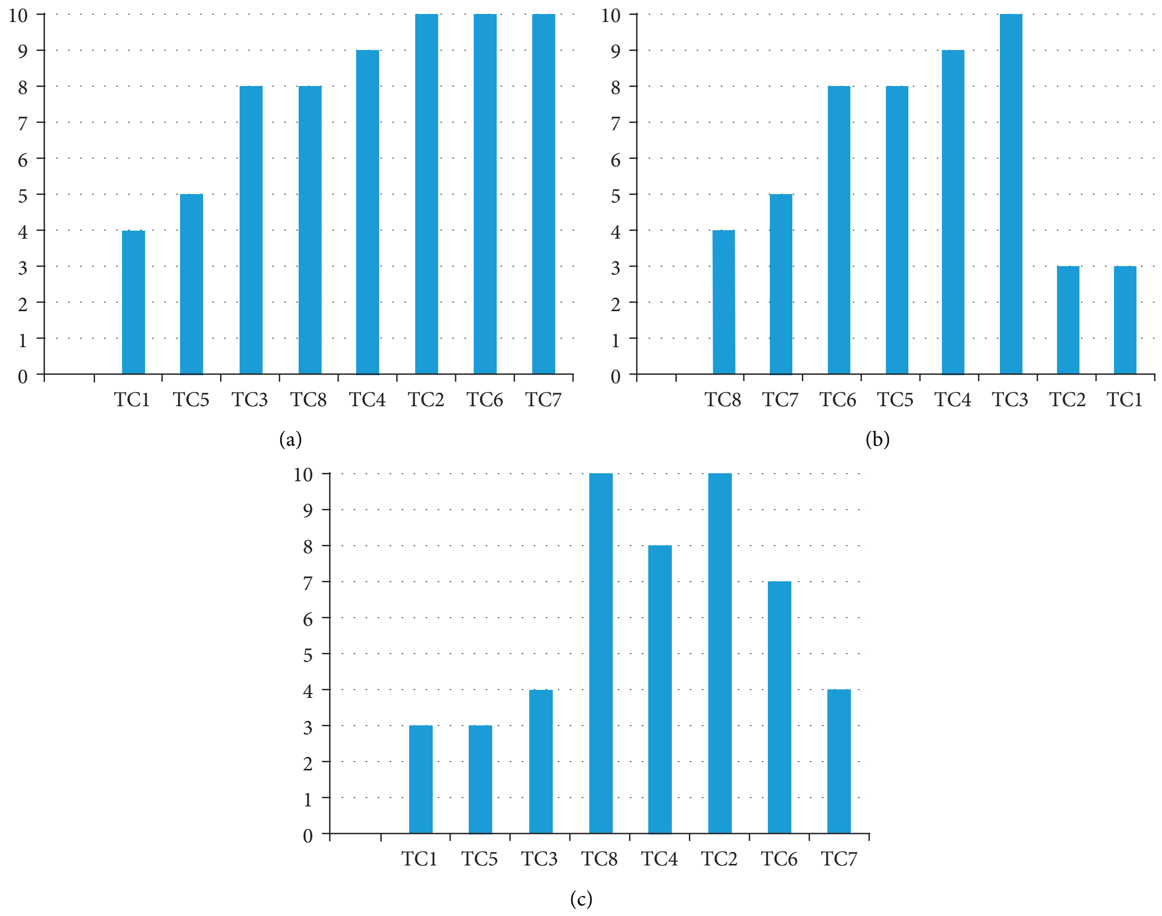


Figure 4: Fault coverage representation for random order approach. (a) TreeDataStructure random order. (b) Triangle random order. (c) JodaTime random order.

Table 9: Proposed MOPSO test case set with minimum test cases.

Test case	Binary form	Execution time	Priority
TC4	1001010110	3	3.0
TC3	0100101001	4	2.0
TC5	1010010001	4	0.7
TC8	1010110001	6	0.5

MOPSO achieved 67% fault detection as visualized in Figures 5(b) and 5(c), respectively. Higher coverage rate of the presented approach shows outstanding performance. The presented approach has less execution time and maximum fault coverage. Increased fault detection percentage for the dataset TreeDataStructure fault coverage rate of the proposed MOPSO is shown in Figure 5(a), and JodaTime results are shown in Figure 5(b). The coverage results for Triangles are illustrated in Figure 5(c) showing the higher effectiveness and performance of proposed technique for test case prioritization. The proposed technique is effective and robust for the test suite prioritization as illustrated in Figures 4, 5, and 7.

Tables 6–9 show the obtained set of minimal test case by the proposed MOPSO, reverse ordering, no ordering, and random ordering. Test cases of test suite 2 with detected faults and execution time are presented in Table 4, where the detection of fault is presented by 1 and 0 for absence. The comparison of the proposed technique is done with prioritization approaches as random, reverse, and no ordering. Results for comparison are computed by taking APFD for each approach. To improve fault detection in test suite, APFD metric is considered. Let *T*suite be the test suite having *k* test case, and *Flt* is the set of *j* faults wrapped by

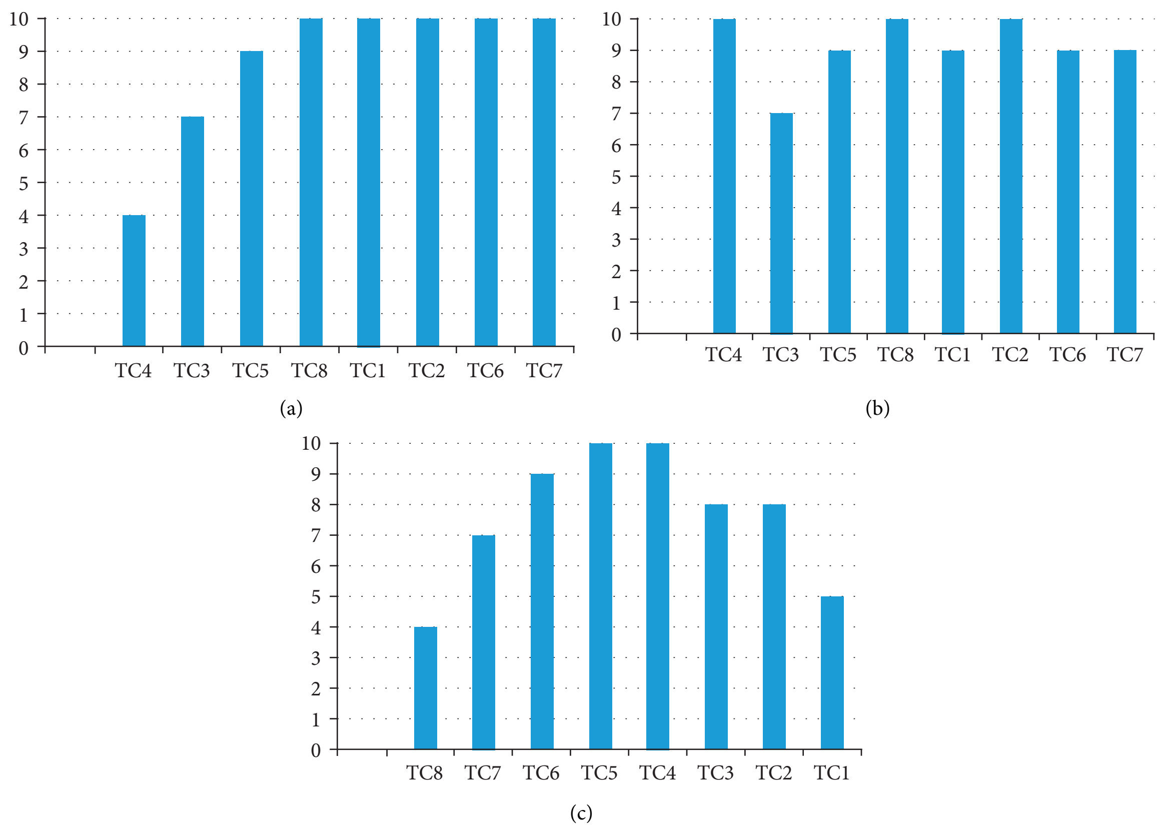


Figure 5: Fault coverage representation for proposed MOPSO approach. (a) TreeDataStructure MOPSO order. (b) JodaTime proposed MOPSO. (c) Triangle proposed MOPSO.

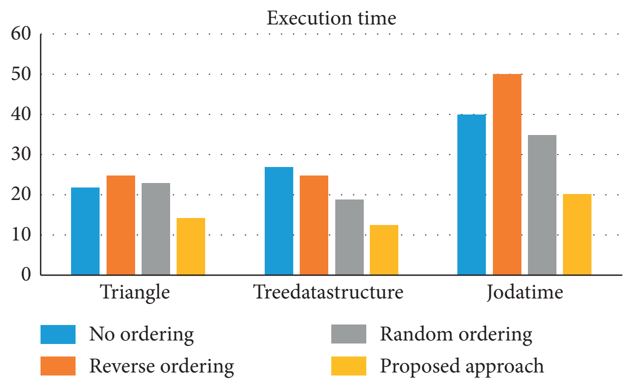


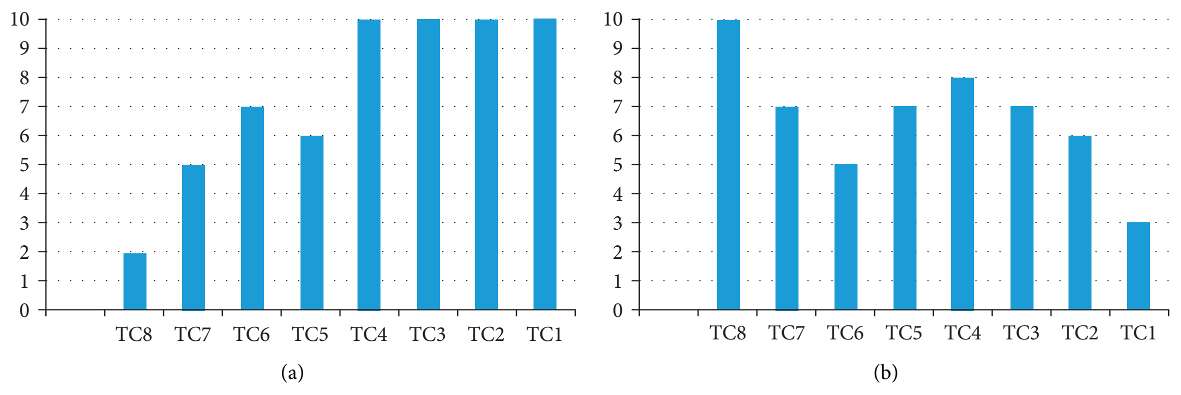
Figure 6: Running time cost of TreeDataStructure test suites 1 and 2 for no, reverse, and random with the proposed approach.

the *T*suite. The *T*suite *Fltj* is the first test case sequencing of *T*suite that shows fault *i*.

The presented technique shows the outstanding results as compared to random, no, and reverse ordering, as it gains the lowest time for execution with complete faults coverage, which is shown in the table. The APFD results for each approach are illustrated in Figures 3–5 and 7. The prioritization sequence of the test cases for test suite 1 is shown in Table 10. The time for execution of test suite is not considered in APFD. In our proposed approach, the execution time is considered for selection and prioritization of test cases. Later, the APFD is applied for outcome comparison with other approaches as presented in Figure 4 that shows the higher stability of presented approach as compared to others.

The prioritized order for presented approach, random order, reverse order, and no order approach is represented in Table 10. In random order approach, the test cases are randomly ordered, and there is no sequence. The proposed MOPSO ordering includes the use of priority for computing the outcomes as compared to other approaches.

The execution time of the test cases in different datasets is compared with random, reverse, and no ordering approach, as given in Table 11. The results show that the



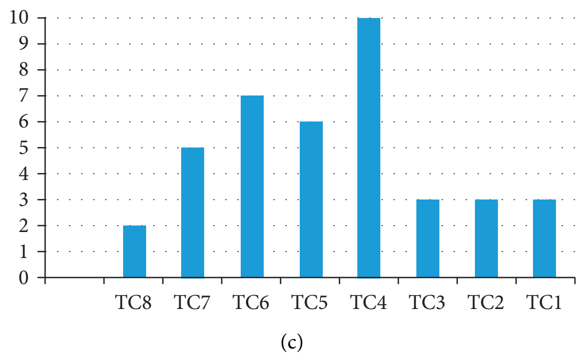


Figure 7: Fault coverage representation for reverse order approach on the three datasets. (a) TreeDataStructure reverse order. (b) JodaTime reverse order. (c) Triangle reverse order.

Table 10: Test suite 1 prioritized order of test cases.

No ordering	Reverse ordering	Random ordering	Proposed ordering
TC1	TC8	TC1	TC4
TC2	TC7	TC5	TC3
TC3	TC6	TC3	TC5
TC4	TC5	TC8	TC8
TC5	TC4	TC4	TC1
TC6	TC3	TC2	TC2
TC7	TC2	TC6	TC6
TC8	TC1	TC7	TC7

Table 11: Time taken by tests for no, reverse, and random ordering and the proposed approach.

Tests on datasets	Random ordering	Reverse ordering	No ordering	Presented approach
Triangles	23	25	22	14.35
TreeDataStructure	19	25	27	12.67
JodaTime	40	50	35	20.40

proposed approach has less execution time rather than other techniques proving that our proposed approach outperformed the other approaches.

For better visualization, the running time for test suites 1 and 2 for MOPSO with random, reverse, and no ordering for selected datasets is shown in Figure 4 that shows that average time required for execution is less in our proposed MOPSO as compared to other approaches for all datasets. On 100 executions, the average execution time for test suite 1 is 14.35, whereas the least possible execution time is 14, which shows the low deviation in outcomes, which show the stabilization of the proposed approach.

6. Conclusion

Regression testing has its own significance as it prevents the software application from the side effects of changes. This study on regression testing is done by triparameter including fault detection ability, code coverage, and cost. In the presented approach, in the first step, the test case redundancy was removed. In the next step, the test case selection was done with minimal test for coverage of all faults to reduce the running time for test suite and for the MOPSO used. In the third step, priority is allocated to the selected test cases. The proposed MOPSO shows outstanding performance as compared to other approaches including random ordering, reverse ordering, and no ordering. MOPSO gets highest coverage, less cost, and extreme fault detection.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was supported by Universiti Tun Hussein Onn Malaysia (UTHM) through Matching Grant (vot H995). It is also partly funded by the Research and Innovation, Universiti Kuala Lumpur.

References

[1] N. B. Ellison, V. Jessica, G. Rebecca, and L. Cliff, “Cultivating social resources on social network sites: facebook relationship maintenance behaviors and their role in social capital processes,” *Journal of Computer-Mediated Communication*, vol. 19, no. 4, pp. 855–870, 2014.

[2] R. Kazmi, D. Abang Jawawi, R. Mohamad, and I. Ghani, “Effective regression test case selection: a systematic literature review,” *ACM Computing Surveys*, vol. 50, no. 2, pp. 1–32, 2017.

[3] J. Ahmad and S. Baharom, “Factor determination in prioritizing test cases for event sequences: a systematic literature review,” *Journal of Telecommunication, Electronic and Computer Engineering*, vol. 10, no. 1–4, pp. 119–124, 2018.

[4] R. D. Adams, “Nondestructive testing,” in *Handbook of Adhesion Technology*, Springer, Berlin, Germany, 2018.

[5] M. Khatibsyarbini, A. M. Isa, D. N. A. Jawawi, and R. Tumeng, “Test case prioritization approaches in regression testing: a systematic literature review,” *Information and Software Technology*, vol. 93, pp. 74–93, 2018.

[6] E. Cruciani, B. Miranda, R. Verdecchia, and A. Bertolino, “Scalable approaches for test suite reduction,” in *Proceedings of the 41st International Conference on Software Engineering*, Montreal, Canada, May 2019.

[7] G. P. Sagar and P. Prasad, “A survey on test case prioritization techniques for regression testing,” *Indian Journal of Science and Technology*, vol. 10, no. 10, pp. 1–6, 2017.

[8] Q. Luo, K. Moran, and D. Poshyvanyk, “A large-scale empirical comparison of static and dynamic test case prioritization techniques,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, November 2016.

[9] JUnit, Java Testing, 2016, http://junit.org/junit4/.

[10] V. Neethidevan and G. Chandrasekaran, “Database testing using Selenium web driver–a case study,” *International Journal of Pure and Applied Mathematics*, vol. 118, no. 8, pp. 559–566, 2018.

[11] Z. Sultan, S. Nazir Bhatti, S. Asim, and R. Abbas, “Analytical review on test cases prioritization techniques: an empirical study,” *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 2, 2017.

[12] B. Jiang and W. K. Chan, “Input-based adaptive randomized test case prioritization: a local beam search approach,” *Journal of Systems and Software*, vol. 105, pp. 91–106, 2015.

[13] F. Harrou, Y. Suna, B. Taghezouitb, A. Saidic, and M.-E. Hamlatid, “Reliable fault detection and diagnosis of photovoltaic systems based on statistical monitoring approaches,” *Renewable Energy*, vol. 116, pp. 22–37, 2018.

[14] A. Bajaj and O. P. Sangwan, “Study the impact of parameter settings and operators role for genetic algorithm based test case prioritization,” in *Proceedings of 2019 International Conference on Sustainable Computing in Science, Technology and Management (SUSCOM)*, Jaipur, India, February 2019.

[15] M. Laali, H. Liu, M. Hamilton, and M. Spichkova, “Test case prioritization using online fault detection information,” in *Proceedings of the Ada-Europe International Conference on Reliable Software Technologies*, Pisa, Italy, June 2016.

[16] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, “Fast approaches to scalable similarity-based test case prioritization,” in *Proceedings of the 40th International Conference on Software Engineering*, New York, NY, USA, May 2018.

[17] P. Raulamo-Jurvanen, M. Mäntylä, and V. Garousi, “Choosing the right test automation tool: a grey literature review of practitioner sources,” in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, Karlskrona, Sweden, June 2017.

[18] Q. Luo, K. Moran, D. Poshyvanyk, and M. Di Penta, “Assessing test case prioritization on real faults and mutants,” 2018, http://arxiv.org/abs/1807.08823.

[19] D. Hao, L. Zhang, and H. Mei, “Test-case prioritization: achievements and challenges,” *Frontiers of Computer Science*, vol. 10, no. 5, pp. 769–777, 2016.

[20] M. M. Öztürk, “A bat-inspired algorithm for prioritizing test cases,” *Vietnam Journal of Computer Science*, vol. 5, no. 1, pp. 45–57, 2018.

[21] X. Zhang, X. Zhenga, R. Cheng, J. Qiu, and Y. Jinc, “A competitive mechanism based multi-objective particle swarm optimizer with fast convergence,” *Information Sciences*, vol. 427, pp. 63–76, 2018.

[22] S. Rahimi, A. Abdollahpouri, and P. Moradi, “A multi-objective particle swarm optimization algorithm for community detection in complex networks,” *Swarm and Evolutionary Computation*, vol. 39, pp. 297–309, 2018.

[23] A. Kiran, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, “A comprehensive investigation of modern test suite optimization trends, tools and techniques,” *IEEE Access*, vol. 7, pp. 89093–89117, 2019.

[24] G. Rothermel, “Improving regression testing in continuous integration development environments (keynote),” in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, New York, NY, USA, November 2018.

[25] D. K. Yadav and S. K. Dutta, “Test case prioritization using clustering approach for object oriented software,” *International Journal of Information System Modeling and Design*, vol. 10, no. 3, pp. 92–109, 2019.

[26] R. Ramler and C. Klammer, “Enhancing acceptance test-driven development with model-based test generation,” in *Proceedings of the 2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Sofia, Bulgaria, July 2019.

[27] R. Currie and C. Fitzpatrick, “Monitoring LHCb Trigger developments using nightly integration tests and a new interactive web UI,” in *Proceedings of the EPJ Web of Conferences*, Les Ulis, France, July 2019.

[28] V. Gupta, "A hybrid approach of regression-testing-based requirement prioritization of web applications," in *Multidisciplinary Approaches to Service-Oriented Engineering*IGI Global, Hershey, PA, USA, 2018.

[29] P. Velmurugan and P. Goel, "Test data generation for improving the effectiveness of test case selection algorithm using test case slicing," *Journal of Computational and Theoretical Nanoscience*, vol. 16, no. 5-6, pp. 1848–1853, 2019.

[30] J. A. Parejo, A. Sánchez, S. Segura, and A. Ruiz-Cortés, "Multiobjective test case prioritization in highly configurable systems: a case study," *Journal of Systems and Software*, vol. 122, pp. 287–310, 2016.

[31] M. A. Alipour, A. Shi, R. Gopinath, and D. Marinov, "Evaluating non-adequate test-case reduction," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA, 2016.

[32] A. Arrieta, S. Wang, G. Sagardui, and L. Etxeberria, "Search-based test case prioritization for simulation-based testing of cyber-physical system product lines," *Journal of Systems and Software*, vol. 149, pp. 1–34, 2019.

[33] E. J. Lenk, H. C. Moungui, M. Boussinesq et al., "A test-and-not-treat strategy for onchocerciasis elimination in Loa loa coendemic areas: cost analysis of a pilot in the Soa health district, Cameroon," *Clinical Infectious Diseases*, vol. 70, no. 8, pp. 1628–1635, 2019.

[34] A. B. Sánchez, S. Segura, J. Antonio Parejo, and A. Ruiz-Cortés, "Variability testing in the wild: the drupal case study," *Software and Systems Modeling*, vol. 16, no. 1, pp. 173–194, 2017.

[35] M. Kintis, M. Papadakis, A. Papadopoulos, and E. Valvis, "How effective are mutation testing tools? an empirical analysis of Java mutation testing tools with manual analysis and real faults," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2426–2463, 2018.

[36] R. Huang, W. Zong, D. Towey, Y. Zhou, and J. Chen, "An empirical examination of abstract test case prioritization techniques," in *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, Buenos Aires Argentina, May 2017.

[37] M. Azizi and H. Do, "A collaborative filtering recommender system for test case prioritization in web applications," 2018, http://arxiv.org/abs/1801.06605.

[38] M. Al-Hajjaji, T. Thüm, M. Lochau, J. Meinicke, and G. Saake, "Effective product-line testing using similarity-based product prioritization," *Software and Systems Modeling*, vol. 18, no. 1, pp. 499–521, 2019.

[39] M. Ivanković, G. Petrović, R. Just, and G. Fraser, "Code coverage at Google," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Tallinn, Estonia, August 2019.

[40] X. Cai and M. R. Lyu, "The effect of code coverage on fault detection under different testing profiles," *ACM SIGSOFT-Software Engineering Notes*, vol. 30, no. 4, pp. 1–7, 2005.

[41] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Software Quality Journal*, vol. 12, no. 3, pp. 185–210, 2004.

[42] A. P. Agrawal and A. Kaur, "A comprehensive comparison of ant colony and hybrid particle swarm optimization algorithms through test case selection," in *Data Engineering and Intelligent Computing*, Springer, Berlin, Germany, 2018.

[43] M. A. Askarunisa, M. L. Shanmugapriya, and D. N. Ramaraj, "Cost and coverage metrics for measuring the effectiveness of test case prioritization techniques," *INFOCOMP Journal of Computer Science*, vol. 9, no. 1, pp. 43–52, 2010.

[44] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Test case prioritization: an empirical study. in software maintenance," in *Proceedings of the IEEE International Conference on Software Maintenance-1999 (ICSM'99)*, Oxford, UK, September 1999.

[45] L. Zhang, C. Guo, T. Xie, and S. Hou, "Time-aware test-case prioritization using integer linear programming," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, New York, NY, USA, July 2009.

[46] A. P. Conrad, R. S. Roos, and G. M. Kapfhammer, "Empirically studying the role of selection operators duringsearch-based test suite prioritization," in *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, New York, NY, USA, July 2010.

[47] A. Konak, D. W. Coit, and A. E. Smith, "Multi-objective optimization using genetic algorithms: a tutorial," *Reliability Engineering & System Safety*, vol. 91, no. 9, pp. 992–1007, 2006.

[48] W. Dong, L. Kang, and W. Zhang, "Opposition-based particle swarm optimization with adaptive mutation strategy," *Soft Computing*, vol. 21, no. 17, pp. 5081–5090, 2017.

[49] Nayuki, *Project Nayuki*, University of Toronto, Toronto, Canada, 2017, https://www.nayuki.io/.

[50] JodaTime, Joda.org., 2017, https://www.joda.org/jodatime/.

[51] Eclipse, Java Editor, 2017, https://eclipse.org/.