DOI : https://doi.org/10.56452/7-6-120

ISSN: 0974-5823 Vol. 7 No. 6 June, 2022

International Journal of Mechanical Engineering

Prioritization of Multi-Objective Test Cases Using Effectiveness of Test Cases: Method of Multicriteria Scoring

[1]Tapas Kumar Choudhury, [2]Subhendu Kumar Pani and [3]Jibitesh Mishra

[1]Computer Science & Engineering, Biju Patnaik University of Technology, Odisha,India
[2]Krupajal Computer Academy, Bhubaneswar, Odisha, India

[3]Odisha University of Technology and Research, Bhubaneswar, Odisha, India

Abstract:-Regression testing validates modified source code. When performing regression testing, time and resources are limited, so it is necessary to choose the fewest number of test cases possible from test suites to speed up execution. This process, known as test case minimization, aims to improve regression testing by eliminating redundant test cases or prioritising the test cases. Using a multiobjective particle swarm optimisation (MOPSO) technique to prioritise cases while taking into accountminimum executionmaximum code coverage, maximum fault detection efficiency, and maximum time. Testing is prioritised using the MOPSO algorithm. Situations with variablessuch as code coverage, error detection capability, and execution time. Ree datasets are chosen for analysis.Triangle, JodaTime, and TreeDataStructure are components of the proposed MOPSO algorithm. A suggested MOPSO is evaluated againstthe effectiveness of the random ordering, reverse ordering, and no ordering techniques. Higher outcome valuesillustrate the proposed MOPSO's superior effectiveness and efficiency as compared to alternative methods for the Triangle, JodaTime, and TreeDataStructure datasets.The outcome is displayed in 100-index mode, with values ranging from low to high; followingtest cases are prioritised in this way.)Three open-source Java programmes are used in the experiment, which is measured using metrics. Inclusion, accuracy, and matrix size reduction of the test suite.)The outcomes showed that all scenarios were successful inacceptable mode, and the method is 17% to 86% more effective in ensuring inclusivity and 33% to 85% more effective in ensuringaccuracy, and metric size reduction ranging from 17% minimum to 86% maximum.

Keywords: Machine Learning, Software Testing, Test case Prioritization, Test case Selection, Continuous Integration, Systematic.

1. Introduction

Software testing is a crucial step in ensuring the accuracy, completeness, and utility of user requirements and specifications [1]. The programme is put through a process to ensure that it is error-free [2].)Regression testing is one of the methods used in software testing. Regression testing is a technique that keeps track of new software updates and ensures that the upgrades won't have an impact on the functionalities of the software as it now exists. Regression testing involves ensuring that none of the software changes conflict with the functionality that has already been implemented.Testing ensures that any modifications made to the software won't have an impact on previous usage and that all components will function as intended and remain intact.Regression testing is a typical step in the software development process. In bigger organisational settings, code-testing or code-execution professionals carry out this step [4].

However, as regression testing is a repetitive and expensive procedure that must be used anytime a piece of code is modified, it requires a significant amount of time, money, and resources to carry out all regression test cases [4]. Due to limited time, money, and resources, it is occasionally not viable to execute an entire test suite [3]. Making a subset of test cases from a test suite that may be useful in discovering problems and flaws is crucial to lowering the cost of regression testing. In this scenario, the maintenance cost can be decreased by carefully choosing the subset of regression test cases.)Figure 1 depicts the typical procedures used in the software maintenance phase.)Regression testing is a procedure that begins with the first product release, which is why the software development process model refers to it as a maintenance activity. Regression testing is triggered by any change request, software or service update, or subsequent release of the same product. Changes in requirements trigger code alterations. Regression testing is started following source code alteration; any errors found during regression testing are then looped back to the source code modification.)Regression faults include bugs such software enhancement issues, configuration inconsistencies, replacement mistakes, structural code flaws, and service failures or unavailability. The new software version has been issued following the conclusion of regression testing.

Test case prioritisation 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 the actual test suite [6]. The main goal of test case reduction methods is to minimise the test cases in the test suite, which increases the cost of testing. The original test suites are shortened by both test suite minimization (TSM) and test case selection (TCS) [9], whereas TCP only arranges the test cases in test suites without removing any test cases. A small number of test cases that aren't currently being used in testing for a given version of software could join in and end up being crucial in later iterations [10]. To put it another way, prioritisation is more secure than permanent truncating, and regression test case prioritisation is a powerful, trustworthy, effective, and economical regression testing technique [11].)As a result, testers and developers give TCP more attention than test case selection (TCS) or test suite minimization (TSM).

Regression TCP approaches use a few aspects, including code/requirement information, risk management, software production, and metrics, to gauge the effectiveness of the testing process.)This study is a continuation of earlier work that supports the single objective for regression TCP's test case prioritisation metrics of cost (execution time), code coverage, and fault detection information [12].The second study in the same research addresses the various cost metrics and their trade-off in relation to one another and offers the rationale for cost trade-offs.)The primary goals of this controlled experiment are to prioritise the pertinent subset of test cases from the original test suites, reduce the cost of testing, make it cost-effective and efficient, and maintain cost (execution time) within acceptable ranges.

Code coverage and effective fault detection are in check.The establishment of a continuous test case prioritisation process with cost (execution time), code coverage, fault detection, and distinct and accumulative test case prioritisation measures is the second goal. The third goal is to integrate tester experience into the process of prioritising test cases, which offers flexibility in choosing parameters and might improve efficacy. Since a test suite may include several test cases and each test case requires a lengthy execution time, it is concluded from the previous work that it may be expensive to execute or run a single test case when certain

DOI : https://doi.org/10.56452/7-6-120

modifications are made to the target code. Therefore, rather than focusing on total cost, it is necessary to optimise the amount of test cases that can identify potential flaws and take up little execution time.In order to improve the test cases for software under testing (SUT), methodologies for test case prioritisation are applied.Rearranging the test cases with the intention of increasing the fault rate or the potential defects detection as early as feasible is an easier method to explain the prioritisation of test cases [13].We order the test cases according to their importance to the company, shared features, or testing team priorities. Although it takes more time and resources to implement, test case selection chooses the right test cases that have captured the most likely flaws while covering the most code lines in the shortest amount of time. It is effective to perform only a portion of the test suite, as opposed to running the entire test suite again.

Test case selection is divided into categories such as "obsolete," "reusable," "fault revealing," "fault exposing," and "change revealing test cases." Prioritisation is based on either the percentage of faults or the required level of code coverage. Regression testing's primary objective is to reduce costs while providing coverage for any source code modifications made to the programme being tested.)The prioritisation of test cases based on a single optimisation objective, such as cost, fault, or code coverage, is known as a uniobjectiveprioritisation of test suites. The prioritisation of test cases based on two optimisation objectives, such as cost coverage or fault coverage, or any combination of these three parameters, is known as bicriteriaprioritisation. The key issue is that combining these factors is not easy owing to their contradicting nature of measure. Prioritisation is also known as multicriteriaprioritisation of test cases if two or more prioritisation parameters are used. The goal of the optimisation problem in test case prioritisation is not only to decrease cost but also to increase fault rates and decrease or maintain the coverage values as well. Previous studies have ranked test suites according to their ability to detect faults, and in some cases, they have used fault rates with test suite costs [5].These studies ignore code coverage and other aspects of cost like test suite costs.As a result, testers and developers give TCP more attention than test case selection (TCS) or test suite minimization (TSM).

Regression TCP approaches use a few aspects, including code/requirement information, risk management, software production, and metrics, to gauge the effectiveness of the testing process. This study is a continuation of earlier work that supports the single purpose for regression TCP's test case prioritisation parameters of cost (execution time), code coverage, and fault detection information [12]. The reason for cost trade-offs is provided in the second study, which is a component of the same research. It also covers the various cost metrics and their trade-off in relation to one another. The main goal of this controlled experiment is to prioritise the pertinent subset of test cases from the original test suites while minimising testing process costs and making it cost-effective and efficient. This is done by keeping costs (execution time), test suite sizes, code sizes, and code coverage constant.Cost reduction is the main goal of regression testing, thus the two factors cost and code coverage are essential [7, 8]. Coverage gives trust in the testing process' thoroughness. There is a gap in TCP for multiobjective test suite optimisation that takes into account cost, fault detection, and code coverage. Models and frameworks like Junit [9] and Selenium [10] do not offer any mechanisms for ranking test cases in order of their cost, coverage, and fault detection capabilities.

There is no obvious winner among these well-known testing models and frameworks either [11]. Other variables that may affect the accuracy of prioritisation techniques include the size of the software being tested, the size of the test suites available for testing, the testing scenarios under these prioritisation techniques, and the testing environment supporting these prioritisation techniques[12, 15, 16].In terms of the multiobjective and multicriterion test suite prioritisation research domain, the shortcomings of the previous frameworks for unit testing with these aspects impacting accuracy and usefulness of prioritisation strategies pose a challenge [17]. It is also mentioned in comparison studies on single objective TCP with the ability to identify faults that do not provide any information about faults overlooked owing to insufficient coverage or cost as time restrictions on overall testing cycles [6, 18, 19]. Case prioritisation techniques for multiobjective tests may take into account the interdependence and effects of the prioritisation factors of cost, coverage, and fault detection capability [20]. The relationship between the parameters cost, fault detection [5], and coverage is not statistically defined, which makes it difficult to prioritisemultiobjective test suites. The impact of this dependency on the efficacy and utility of prioritisation techniques and frameworks is significant. Ignoring any one of these three factors—cost, coverage, or fault detection capability—can lead to inaccurate testing outcomes, inefficient testing methods, and occasionally time and resource waste. Finding the correlation between defect detection capability, cost, and coverage with the goal of employing them as prioritisation scales becomes difficult in the setting of prioritisation. It has been utilised frequently by researchers to address the issues of multiobjective particle swarm optimisation and multicriteria test case prioritisation [21, 22].

The randomization aspect that alters the particle position that is taken into consideration in this work is the main problem with PSO at the moment. In essence, the three main problems with test case prioritisation are the topic of this study. It starts by identifying the important variables that have an impact on the main goal of reducing the price of regression testing. The second goal of this research is to establish a connection between the efficacy criteria used for the multiobjectiveprioritisation of test cases. The other six sections of this work are organised as follows: Section 2 provides a detailed review of the linked studies; Section 3 elaborates on the suggested technique; and Section 4 presents the results. We talked about the experimental design for our proposed study in Section 4 of the paper. The produced results are reported in Section 5. This section wraps up the paper.

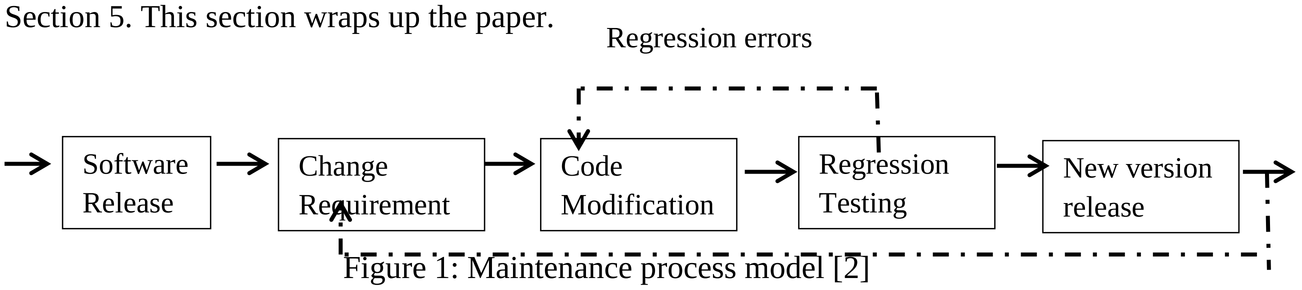


Figure 1: Maintenance process model [2]

2. Additional Work

Software testing is a technique used to tell stakeholders about the calibre of a software service or product that falls within the test category. Software testing is a method of putting an application or programme through its paces in order to find bugs (defects or faults) and determine whether the product is fit for use. Software testing involves running a section of code or a section of a system to evaluate at least one important property. These characteristics typically show how much a component or system is subject to testing. The following are the SUT's essential objectives:

(1) Design and development guidelines
(2) Response to all types of inputs
(3) Task completion in a reasonable amount of time
(4) Software usability
(5) Deployment and execution in a predetermined environment
(6) Achieving the desired results)Regression testing is a procedure that makes sure the code modifications didn't result in unanticipated behaviours [5]. The basic idea behind

DOI : https://doi.org/10.56452/7-6-120

regression testing is retesting SUT with the purpose of exposing the problems sooner [23]. Regression testing also verifies that the software's prior functions are functioning as intended [2].

Regression testing procedures are increasingly used in software development methods like component-based software engineering (CBSE) and iterative development. The development of testing paradigms like continuous integration (CI) [24], continuous development (CD) [25], test-driven development (TDD) [26], and nightly build architectures [27] can all benefit from regression testing.

Regression testing is not necessary since certain software systems go through a standard testing cycle before being released, which eliminates the necessity for such testing. Regression testing is necessary to make sure that previously tested software features are still working properly and aren't causing any unexpected behaviours or outcomes with each new software release or update. Regression testing is so obviously necessary from version 2 to every subsequent version of the product. Existing functionality are removed as part of software maintenance, along with any necessary bug fixes, improvements, and optimisations. The system could malfunction as a result of the changes. Regression testing should therefore be carried out, and the following methods can be used to do so:

(1) Complete retesting

(2) Case selection

3) Sorting test cases by priority

(4) Cutting down on test cases

The necessity to minimise, selects, or prioritise the test cases arises when the size of the test suite reaches the hundreds or even millions. On each new version release or upgrade to an SUT, the whole test suite including all of the test cases is re-run. This process is known as comprehensive retesting. However, when time and money are being shirked, it is foolish and destructive. Practically speaking, the retest-all is impossible for medium to large software with thousands or millions of test cases. The obvious decisions then for a testing team are to choose, prioritise, or condense the test suites based on factors like cost, code coverage, fault detection capability, and code modifications. These factors can sometimes be determined using sufficiency criteria [2] or regression testing parameters [28]. To minimise the amount of time and money needed for testing, these factors are used to choose, order, or condense the test suites. The goal of test case selection strategies is code modifications, and these techniques choose the test cases that contain a modified portion of the SUT [29]. When testing software, the prioritisation of test cases places a priority on issue discovery as early as feasible [30].

Methods for the prioritisation of test cases aim to improve the overall software's quality by detecting flaws at an early stage of testing and indirectly save costs by shortening the time it takes to remedy bugs. The unused or unproductive test cases from a test suite are thought to be removed via test suite reduction strategies [31]. The reduction strategies are helpful when the size of the test suite is irrelevant, however they are criticised since they permanently remove test cases from the test suite. Any testing method's main goal is to reveal the SUT's early flaws [5]. The testing research community began to pay attention to the test case prioritisation strategies as a result [19, 32].

There are numerous difficulties and problems with the execution and analysis of test case prioritisation. It is simply not a straightforward rearranging of the test cases. To reap the rewards of prioritisation techniques, extensive research and analysis are needed. The cost of regression testing, code coverage by test case, fault detection capability, and changes in code made during the maintenance phase are the four main factors that influence prioritization. The cost of the testing includes a number of different types of costs, such as analysis cost, test suite size, time for executing test case, and budget needed for testing [33]. The code coverage affects prioritisation strategies in a wide variety of ways. The percentage of source code lines that the test case targets relative to the total number of lines in the module being tested is known as the coverage of source code for the test case (34). Statement, modified statement, block, condition, modified condition, test, modified block, loop, and many other types of coverage are important types of coverage [2]. The fault detection ability measures how many faults a test case can find and is also determined in terms of fault rates, fault frequency, and fault severity. However, genuine faults, structural faults, hand-seeded faults, and mutation faults are the fault categories discussed in the literature for experiments [35]. One of the most crucial goals of the TCP, which is a crucial component of regression test optimisation (RTO) [18, 36], is to increase fault recognition rate, which is defined as the rate at which a test suite may find defects during the testing cycle. As a result, this cycle's effects can offer and prioritise the system's response as it undergoes testing, and when testing is stopped for whatever reason due to early debugging, the prioritisation will enable the execution of the most fundamental test cases at an earlier stage [37]. The test cases are essentially prioritised according to their needs, which are then addressed using various criteria dependent on the strategy being used, and then performed according to priority [38]. A crucial component of software testing is coverage. Regression testing and maintenance evaluate the effectiveness of the test suite using the test coverage [39]. The coverage is used as a bridge between the stopping criteria for testing and the evaluation of test suites. The degree to which a test suite can run the code being tested is referred to as coverage [2]. The greater the value of coverage, the more trustworthy and confident the testing procedure is. The list of notable articles published about test case prioritisation is shown in Table 1. The likelihood that a test suite will be able to identify the flaws or defects in a programme under test is known as fault detection ability [40]. The ability to detect faults is utilised as a test method adequacy metric. Additionally, it is utilised to compare several techniques against one another, and the technique is judged superior based on comparisons between techniques with superior fault detection ability or fault rates [43]. Every testing approach has flaw detection as its main goal. Regression testing evaluates strategies for prioritising test suites to cut testing costs by using fault detection efficiency and fault rates. Software flaws are sometimes brought on by breaking the rules for writing codes and misinterpreting specifications or design models. Rothermel et al. started the fault-based test suite prioritisation [44]. Fault detection ability or fault rate of that test case or test suite refers to a test case's capacity to find errors. The likelihood that a statement will be executed by a test case determines the likelihood that an error will be discovered by the test suite. The extended study [41] prioritises the test suites for fault detection capability using six level coverage approaches. For each method in the source code under study, these prioritisation approaches use FEP to index the fault levels together with their criticality and fault proneness. These studies' fault indexing procedure involves merging method coverage and damage reports from earlier test cycles. The fault severities were denoted by the term total fault indexing. The statistical analysis yields sufficient information to assess the outcomes using APFD metric suites. The research found that method coverage with FEP was superior to statement coverage-based approaches in terms of effectiveness. The disadvantage of these methods is that fault indexing methods rely on outdated data that was generated during prior test cycles.

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 foe 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 antcolony through selection of test cases	2018

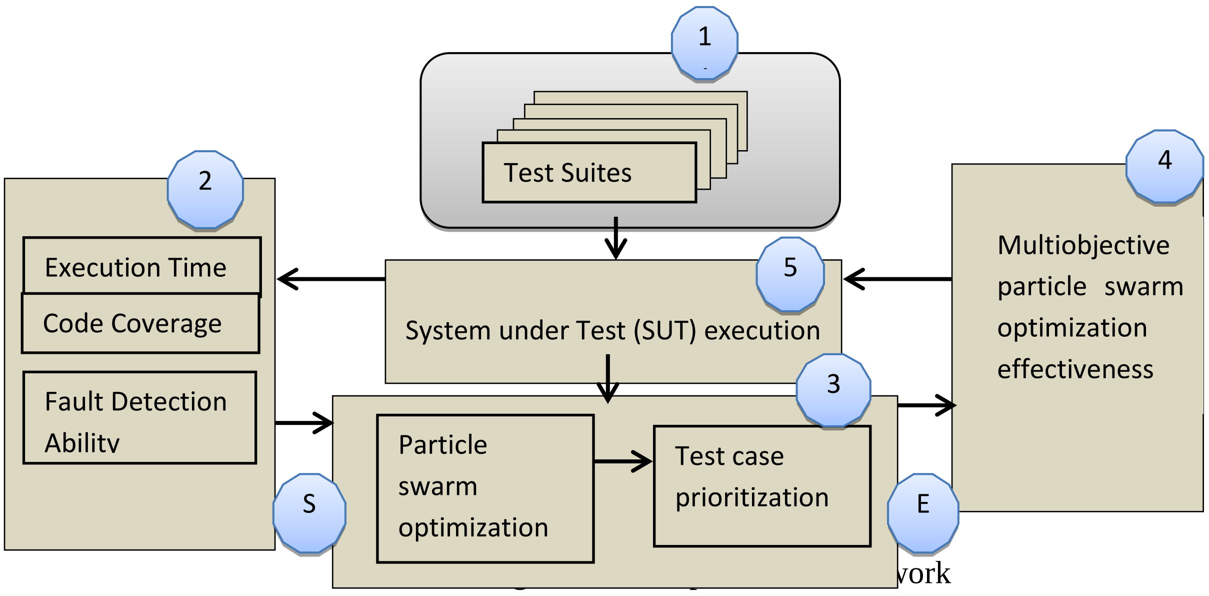
A time-aware test case prioritisation strategy using GA was suggested by the study [45]. By using various programme granularity levels in a controlled experiment, the proposed method was assessed. Emma was used to monitor the coverage for statement coverage, block coverage, and condition coverage.

The data were evaluated by APFD, who state that nonprioritization methods had a 120% efficacy rate.The method was also contrasted with fault-aware prioritisation and reverse ordering prioritisation. A study also suggested GA-based test suite prioritisation for Java applications [46, 47]. On eight different software devices, the study introduces a sizable class of mutation, crossover, and selection operators. Using a coverage-based test suite prioritisation strategy, the effectiveness was evaluated. The outcomes were contrasted with a hill climbing algorithm and a random search-based method. For regression testing of updated software units for real-time embedded systems, a PSO was put forth in [42].

Regression testing's automated test case prioritisation aids in choosing the test cases with the highest priority. By accepting the answer as a space of particles and positions for test cases according to software unit, the PSO is effectively applied to the prioritisation problem. The results demonstrate that the PSO successfully and efficiently orders the test cases in the test suites according to the new optimum places. Dong et al. [48] proposed the population-based stochastic optimisation technique known as PSO. In order to create the best solution to the stated problem, it is utilised to study the available search space. The n particles that make up the search space are collectively referred to as the swarm. PSO uses a few parameters to help find a solution. The initialization of the particle population is done at random, and the position and velocity of the particles are updated as you look for a solution. The best location among complete particles is known as g-best, and each particle has memory to retain its current position, or p-best. The addition of velocity to the previous position updates the position. In order to guarantee that particles will look for the best solution within a certain search space, velocity is controlled by V-max.\

3. Test Case Prioritisation Framework Based on Multi-Criteria Effectiveness Average Score

Two test suites are created for test case prioritisation and are used to evaluate MOPSO using the metrics code coverage, execution time, and fault detection capability. Figure 2 presents the suggested technique. The goal parameters of coverage, cost, and defect detection are employed in multiobjective particle swarm optimisation (MOPSO). The subset of test cases used in the testing process is indicated by the particle position being supplied as decimal vectors. According to assumption $T_i\{T_1, T_2, . . ., T_k\}$ is the test suite consisting of k test cases, the position of particle is given as p_t={t_1, t_2,. . ., tm} where t_j belongs to set {0, 1}.


1
Test Suites
2
Execution Time
Code Coverage
Fault Detection Abilitv
5
System under Test (SUT) execution
4
Multiobjective particle swarm optimization effectiveness
3
Particle swarm optimization
Test case prioritization
S
E

vork

$$vol(g+1) = w \times vol_i(g) + c_1 \times r_{1i} \times \left(p\_best_i(g) - x_i(g)\right) + c_2 \times r_{2i} \times \left(g\_best_i(g) - x_i(g)\right) \quad (1)$$

The lack of the test cases 0 and 1 indicates that T_i is present in a subset of test instances. The test cases are binary-edited. When looking for food, the PSO behaves like a flock of birds.The procedure of finding food is carried out by passing on search knowledge to nearby birds. PSO uses the idea of a flock of birds to find the best answer. By observing the behaviour of the neighboring particle, each particle iteration search space in PSO attempts to converge in the direction of the global best solution. Every other particle, indicated by p_best, and a function called fitness function can compute the best prior location of any particle. G_best stands for the global best position across all particles, and each particle's velocity (execution speed) can be calculated using. The preceding equation's voli (g) and p besti(g) variables show the velocity of the ith particle at the gth generation and population best value globally (g_best value). c1 and c2demonstrate the social and cognitive elements. the inertia, wfactor, r1i and r2i are random numbers between 0 and 1, and factor.Our suggested MOPSO algorithm seeks to enhance severaleffectiveness of the proposed strategy in achieving both of itsexperimentation, a method is calculated, and the outcomes. Results show increased efficiency.

4. Experimental Configuration

This section goes through the experimental design for this investigation.Utilising the system with the following specifications, experimentation is done: MATLAB 2019 and an Intel Core i7 processor with 8 GB of RAM are used for simulation. Triangle [49], TreeDataStructure [49], and JodaTime [50] datasets are chosen for use in the proposed MOPSO implementation.

JodaTime is a replacement for the Java date and time library, while TreeDataStructure and Triangle are academic datasets with Java written test suites. Table 2 lists the information of the selected datasets for the test case prioritisation process, including the dataset name, version, line of code (LOC), and number of test cases for each dataset.Figure 2 shows the experimental flow for our suggested method for test case prioritisation. The system under test (SUT) and test suites is chosen in the first stage.The Java platform is chosen using the eclipse environment [51]. JUnit framework is chosen in order to conduct unit testing. The data needed for our suggested strategy is gathered in the second step utilisingEclEmma [39]. It gathers data about SUT size and coverage. The JUnit is used to calculate the size of the test suite.

The SUT is generated in every version having a defect, and each version is examined. The information related to the test suite size, fault detection ratio, execution duration, and code coverage is gathered from the second phase's data in the third step.)The metrics for data visualisation, data analysis, and the data collection include the code coverage, fault detection, and the execution time of the test case.) This information is then preprocessed to assign the priority to each test case based on the ability to capture the most faults in the shortest amount of time.)The execution time for each test case is divided by the entire execution time of the test suite, and the result is then multiplied by 100. (E code coverage is regarded as the covering statement for a unit test case.)The number of faults detected by each test case in a test suite is used to calculate fault detection capabilities.

5. Result and Discussion

Tables 3 and 4 provide information on the test suite employed in our experiment. The particle size and maximum iterations for PSO are both set to 10. For particles, the lower bound is 1, and the upper bound is n, where n is the test case quantity. For the chosen datasets, no ordering, random ordering, and reverse ordering are contrasted to assess the outcomes of our suggested methodology.

Table 2 The details about dataset

Number	**Dataset**	**Version**	**LOC**	**Test case**
1	Joda Time	2	280464	279
2	Tree data structure	2	2200	22
3	Trainagle	3	116	12

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

Test case	**Binary Form**	**Execution time**
TC1	101010010	6
TC2	0101000101	4
TC3	0100101001	4
TC4	1001010110	3
TC5	1010010001	4
TC6	0101100100	5
TC7	0000101100	4
TC8	1010110001	6

Table 5 shows the relationship between test cases and defects for test suite 1. Test Suite 1 has a total of 8 test cases, designated as TC1–TC8, and 10 probable faults, designated as

F1–F10. The binary value 1 in Table 3 indicates that the corresponding test case has detected the problem, while 0 indicates that the fault has not been detected (Table 3).The execution time is the amount of time each test case in the test suite needs. Because MOPSO uses a random parameter, the average results are calculated after the algorithm has been run more than 100 times. The average percentage of fault detection (APFD), execution time, and code coverage have been chosen as the performance evaluation metrics.

Table 4 Test Suite 2 representation with detected faults in binary form and time

Test case	**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	0	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

The following equation is used to calculate APFD.

$$\text{APFD} = 1 - \frac{TsuiteFlt_1 + TsuiteFlt_2 + TsuiteFlt_3 + \cdots + TsuiteFlt_j}{kj} + \frac{1}{2k} \qquad (2)$$

Table 6 contains the minimum test case set for the no ordering technique.)The test case, the binary form of the found flaws, and the execution time (in hours) are all included in the representation. The chosen test cases in the condensed test set are TC1, TC2, TC3,

DOI : https://doi.org/10.56452/7-6-120

TC4, and TC5. For test suite 1, a maximum execution time of 6 hours is noted.)Figure 3 shows the fault coverage for no order approach for three chosen datasets. Figures 3(a)-3(c), where the x-axis represents the test cases and the y-axis displays the fault coverage, demonstrate, respectively,

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

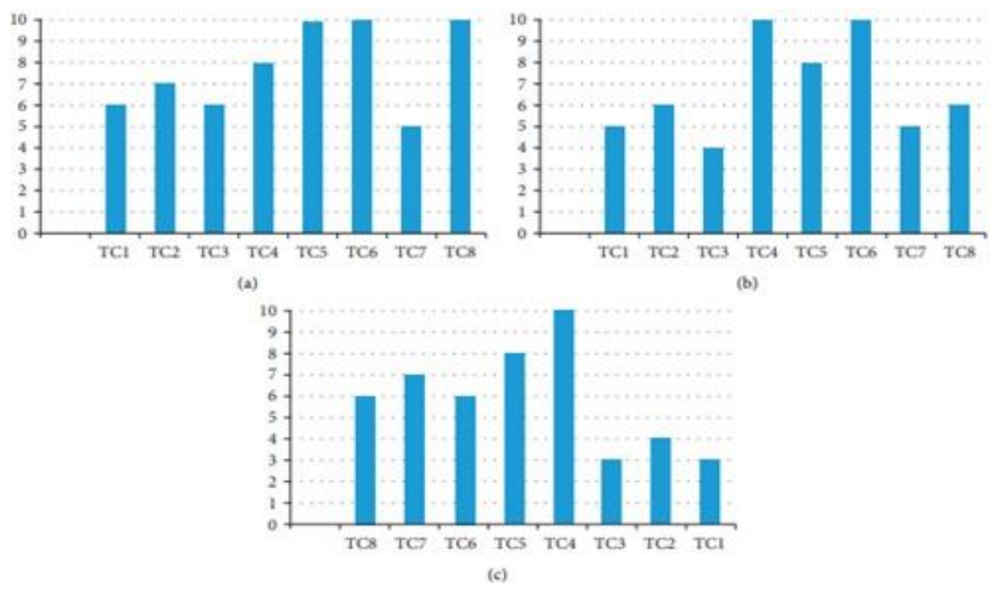


FIGURE 3: Fault coverage representation for no order approach. (a) TreeDataStructure no order. (b) JodaTime no order. (c) Triangle no order.

Figures 3(a)-3(c), where the x-axis represents the test cases and the y-axis displays the fault coverage, demonstrate, correspondingly, the fault detection rate for test suite 1 for Tree Data Structure, Joda Time, and Triangle.The percentage with each test case indicates the amount of fault coverage. The no order technique yields a higher detection rate for the Tree Data Structure dataset, with a 51% fault detection rate for the Triangle dataset, compared to 68% fault detection for the Tree Data Structure dataset and 60% fault detection for the Joda Time dataset.

Table 7 shows the representation of identified defects, minimum selected test cases, and execution time for the random ordering strategy. The test cases TC1, TC5, TC3, TC8, and TC4 are chosen for the test case set. Three hours must pass before a random order can be executed.

Table 8 provides the test case set with the minimal test cases for the reverse ordering strategy. The chosen test case in binary form and the duration of each test case's execution are shown. The test cases chosen for the strategy of reverse ordering are TC8, TC7, TC6, TC5, and TC4.

DOI : https://doi.org/10.56452/7-6-120

Figures 4(a)–4(c) show the fault coverage for Triangle, Joda Time, and Tree Data Structure using the reverse order technique. Figure 4(a) shows the reverse order technique for the TreeDataStructure dataset, which captures 66% fault detection, while Figure 4(b) shows the Joda Time dataset, which captures 55% fault detection. Reverse order approach displays 43% defect detection on Triangle dataset execution, which is less than TreeDataStructure and JodaTime as shown in Figure 4.

The details of the 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 visualised in Figure 5.)e proposed MOPSO approach covered area is higher than that of no, reverse, and random ordering. According to Figure 6(a), 77% of faults can be detected using the suggested MOPSO for TreeDataStructure. JodaTime defect detection is 81%, and the proposed method for the Triangle dataset

Table 7 Random ordering technique test case set minimum test cases

Test Cases	**Binary Form**	**Execution Time**
TC1	1010110010	6
TC5	1010010001	4
TC3	0100101001	4
TC8	1010110001	6
TC4	1001010110	3

As seen in Figures 5(b) and 5(c), MOPSO successfully detected 67% of faults. The presented approach provides maximum fault coverage and a higher coverage rate, both of which demonstrate remarkable performance. Figure 5(a) displays the increased fault detection percentage for the dataset TreeDataStructure fault coverage rate of the proposed MOPSO, and Figure 5(b) displays the JodaTime findings. Figure 5(c) shows the coverage results for Triangles and demonstrates the higher performance and efficacy of the suggested technique for test case prioritisation. The test suite prioritisation using the suggested technique is efficient and reliable, as shown in Figures 4, 5, and 7.

The resulting set of minimal test cases using the proposed MOPSO, reverse ordering, no ordering, and random ordering are displayed in Tables 6 through 9. Table 4 lists the test cases from test suite 2 together with the execution duration and identified defects. The detected fault is indicated by a_1 for presence and a_0 for absence.

DOI : https://doi.org/10.56452/7-6-120

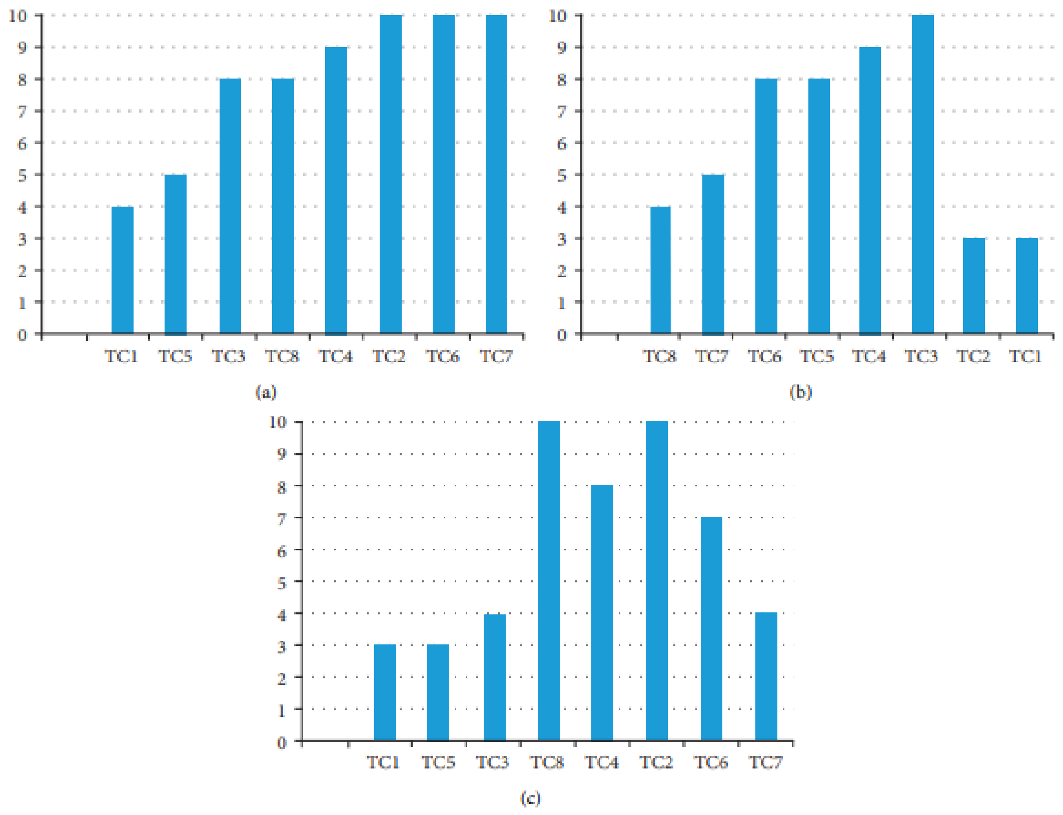


FIGURE 4: Fault coverage representation for random order approach. (a) TreeDataStructure random order. (b) Triangle random order. (c) JodaTime random order.

Table 8 Reverse ordering technique test case with minimum test case

Test Case	Binary Form	Execution time
TC8	1010110001	6
TC7	0000101100	4
TC6	0101100100	5
TC5	1010110001	4
TC4	1001010110	3

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

The proposed strategy is compared to prioritisation strategies such as random, reverse, and no ordering. Taking the APFD for each strategy allows for the computation of results for comparison. The APFD metric is taken into consideration to enhance test suite fault detection. Let Flt be the collection of j faults that the test suite, Tsuite, has wrapped around its k test cases. The first test case sequencing of Tsuite that exhibits fault i is TsuiteFltj.

As it takes the least amount of time to execute with full fault coverage, as indicated in the table, the provided technique outperforms random, no, and reverse ordering. Figures 3-5 and 7 show the APFD outcomes for each method. Table 10 displays the order in which the test cases for test suite 1 were given priority. The time spent running the test suite is not taken into account in the APFD. The execution time is taken into consideration when choosing and prioritising test cases in our suggested methodology. Later, the APFD is used to compare the results with other ways, as shown in Figure 4, which demonstrates the proposed approach's superior stability in comparison to others.The random order of the prioritised approach istaken. There are three approaches: order, reverse order, and no order approach from Table 10. The test examples are presented using a random order technique.

There is no sequence, and the items are randomly arranged. In contrast to other methodologies, MOPSO ordering uses priority to compute the results. As shown in Table 11, the execution times of the test cases across various datasets are compared using the random, reverse, and no ordering approaches. The results demonstrate that the proposed solution outperformed the alternatives since it takes less time to implement than alternative techniques. The running time for test suites 1 and 2 for MOPSO with random, reverse, and no ordering for selected datasets is displayed in Figure 4 for better visualisation. This figure demonstrates that the average execution time for our proposed MOPSO is smaller than that of other techniques for all datasets. The average execution time for test suite 1 on 100 executions is 14.35, while the shortest execution time is 14, whichlittle variance in results, which demonstrate thestability of the suggested strategy.

6. Conclusion

Regression testing is important because it shields the software programme from any unintended consequences of modifications. The three metrics used in this regression testing study are cost, code coverage, and fault detection capabilities. The redundant test cases were eliminated in the first step of the suggested approach. The selection of test cases was done in the next stage with few tests to ensure complete coverage of all faults and to shorten the time required for the test suite and the MOPSO to run. The third step assigns priority to the chosen test cases. When compared to alternative methods like random ordering, reverse ordering, and no ordering, the suggested MOPSO performs remarkably well.MOPSO achieves the highest coverage, the lowest cost, and the most robust fault detection.

Reference

[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. AbangJawawi, 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. M¨antyl¨a, 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. Ozt¨urk, “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 testdriven 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. 12 Scientific Programming

[28] V. Gupta, “A hybrid approach of regression-testing-based requirement prioritization of web applications,” in Multidisciplinary Approaches to Service-Oriented EngineeringIGI 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 ,eoretical Nanoscience, vol. 16, no. 5-6, pp. 1848–1853, 2019.

[30] J. A. Parejo, A. Sanchez, S. Segura, and A. Ruiz-Cort ´ es, “Multi- ´ objective 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, “Searchbased 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-andnot-treat strategy for onchocerciasis elimination in Loa loacoendemic 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. Sanchez, S. Segura, J. Antonio Parejo, and A. Ruiz- ´ Cortes, “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.)¨um, 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. Ivankovic, G. Petrovi ´ c, 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, “)e effect of code coverage on fault detection under different testing profiles,” ACM SIGSOFTSoftware 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 duringsearchbased test suite prioritization,” in Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation, New York, NY, USA, July 2010.

DOI : https://doi.org/10.56452/7-6-120

[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/.