Journal of Electrical Engineering, Vol. 75, No. 2, 2024, pp. 151-160

sciendo

PAPERS

Fault sensitivity index-based multi-objective testcase prioritization

Kamal Garg[1,2], Shashi Shekhar[1]

Test case prioritization (TCP) is a regression technique that sequences test cases by assigning priority based on specific criteria defined by software testers. Various parameters, such as code coverage, statement coverage, and method coverage, are utilized in Test Case Prioritization (TCP), wherein metaheuristic techniques are widely employed to determine the optimal order of test cases based on these specified parameters. However, simply applying these techniques does not ensure the satisfaction of all the needs of software testers. This paper introduces an empirical study that employs the multi-objective test case prioritization (MOTCP) technique to prioritize the test cases based on target points defined by software testers. The study calculates a Software Complexity Index (SCI) at the code level, identifying fault-prone areas. Furthermore, a Test-case Complexity Index (TCI) is also used for prioritization. The proposed technique incorporates various target points defined by the software tester to calculate SCI and TCI, which serve as our main objectives for TCP. A detailed analysis is also performed to examine the impact of these target points on the generated optimal order of test cases. Finally, the proposed model is compared with other state-of-the-art techniques across various evaluation parameters.

Keywords: regression testing, test case prioritization, NSGA-2, soft computing, MOTCP, SCI, test case complexity index

1 Introduction

Software and applications, ranging from mobile to desktop applications, have become integral to daily human life. Ensuring bug-free software is crucial, leading software development companies to allocate a significant portion of their budget to software maintenance. Regression testing is a crucial aspect of this process, involving testing new versions or patches to ensure compatibility with previous software iterations. This makes regression testing time-consuming and resource-intensive. There are three main approaches used for software regression testing: test case selection, test case reductions, and test case prioritization which involve execution of the test cases based on priority order.

In Test Case Prioritization (TCP) [1], the main objective is to prioritize the test cases in terms of faults and cost, which means test cases that can detect more faults should be given high priority compared to other test cases. Test suites that take less time should also be prioritized in execution order. Many parameters are considered during TCP to achieve this objective, e.g., code coverage, statement coverage, method coverage, etc. It is believed that a test case, when it covers a large amount of code, methods, or statements, can detect a large amount of fault compared to other test cases that cover less code. However, simply considering these parameters in TCP, they are ineffective in fault detection. Many scholars use more than one parameter to overcome this problem and generate effective orders of test cases.

The proposed technique used multiple parameters, code block coverage and cost in a multi-objective fashion to optimize the order of test cases. The main contribution of this paper is to focus on some target points defined by the software tester to prioritize test cases. These target points help to identify bugs in applications and produce cost-efficient TCP. Here, object-oriented applications are used to test the proposed methodology. For the implementation, various tools are used to extract software artefacts and later, these artefacts are used to find optimal order of the test cases. To solve this MOTCP problem, a non-dominating sorting genetic algorithm (NSGA-2) [2] is used. Detailed analysis is also conducted to find the answer to the following research question (RQ).

- **RQ1:** How does the proposed methodology work in comparison to other state-of-the-art algorithms in terms of fault detection?
- **RQ2:** Which of the algorithms achieves the earliest fault detection?
- **RQ3:** How does the proposed technique result in terms of cost-effectiveness?

[1] GLA University, Mathura, India, [2] Tata Consultancy Services, India
kamal.garg_phd.cs21@gla.ac.in, shashi.shekhar@gla.ac.in

https://doi.org/10.2478/jee-2024-0018, Print (till 2015) ISSN 1335-3632, On-line ISSN 1339-309X

- **RQ4:** How does the proposed MOTCP model perform on small, medium and large software systems?
- **RQ5**: Which technique can achieve target points defined by software testers?

The proposed manuscript includes section 2 that elaborates on related state-of-the-art work, followed by the proposed methodology in section 3. The experimental assessment and its various assessment metrics are explained in section 4. Section 5 clearly demonstrates the results analysis of the proposed scheme. Finally, conclusions and future scope are listed in section 6.

2 Contextualizing previous studies

The overview is started with review papers that emphasize regression testing [3] in detail. Recently, in 2021, Mukherjee et al. [4] presented an inclusive survey containing different techniques on TCP. The author reviewed almost 90 research articles published on regression test selection (RTS) and TCP from 2001 to 2024. In 2021, Tian et al. [5] presented an extensive survey on large-scale multi-objective optimization techniques, and Qasim et al. [6] presented various regression testing methods, whereas authors Pan et al. [7] studied the TCP using various machine learning techniques. Strandberg et al. [8] published an article on automated regression test prioritization in 2017. This study discussed the challenges and advancement. Khatibsyarbini et al. [9] also reviewed semantic regression testing in 2018. Bajaj et al. [10] also compared and studied various TCP techniques based on genetic algorithms in 2016. The said review papers give detailed information on the field of TCP.

The cost-centric prioritization method emphasizes parameters such as cost analysis, execution, and maintenance costs while designing cost models. There are different ways to test software, and we often use a measure called Average Percentage of Fault Detection (APFD) to see how well a technique is working. Yu-Chi Huang et al. proposed history based cost-cognizant test case prioritization to investigate and improve the APFD value [11]. In 2021, Huang et al. [12] proposed a learn-to-rank technique to prioritize the test cases. The proposal combined extended finite state machine features with a random forest algorithm to improve the fault detection rate. The performance is measured in terms of APFD, which reaches 0.884. In another study, Yadav et al. [13] calculated the APFD value with the help of a unified modelling language (UML) diagram based on object-oriented software. Khanna et al. [14] compared many algorithms like NSGA-II, genetic algorithm, random approach, and 2-opt algorithm on five web applications. The applied algorithms prioritize the test sequence to increase the APFD rate and minimize the procedure's cost.

However, APFD works best when all tests have equal cost, and all faults are equally severe. If the tests have different levels of faults and cost, using APFD might give us incorrect results. So, it's crucial to choose the right measure to get reliable and accurate results. For this problem, Ahmed et al. [3] proposed two evaluation metrics called Average Percentage of Fault Detection per value (APFDv) and Average Percentage of Requirements Coverage per value (APRCv) with the help of genetic algorithm. In a study, Nucci et al. [15] also proposed a genetic algorithm influenced by the hypervolume Indicator. The results showed that the proposed technique not only reduced the cost but also improves the prioritization efficiency. In 2002, Deb et al. [2] introduced a fast and elitist multi-objective genetic algorithm named non-dominated NSGA-II for better coverage and to optimize better performance. It also requires fewer parameters and a low computational approach. In 2015, Epitropakis et al. [16] illustrated the importance of applying Pareto benefits for test case selection. Baoying et al. [17] proposed a genetic algorithm solution based on diversity to achieve balanced testing to enhance fault detection rate. Geetha et al. [18] suggested a multi-level random walk algorithm to reduce test cases. The approach also included a genetic algorithm. In another study, Khanna et al. [19] focused on multi-objective algorithms and compared ant bee colony and genetic algorithm- II to prioritize the test cases.

Bajaj et al. [20] applied a three-level regression testing approach (prioritization, selection, and minimization) that is based on nature-inspired algorithms named as particle swarm optimization (PSO), genetic algorithm (GA), NSGA-II and artificial bee colony (ABC). Zheng Li et al. [21] presented an empirical study on greedy, metaheuristic, and evolutionary search algorithms for regression TCP to address the problems like finding modality, fitness metrics and most suited test cases. The result shows that greedy algorithm performs better than other algorithms.

Bian et al. [22] used a metaheuristic approach, epistasis based ACO for Regression TCP to combine the knowledge of biological theory and application domain. It is clearly visible that in comparison of single objective technique, meta-heuristic approaches are more effective. Marchetto et al. [23] applied IR based traceability recovery and latent semantic indexing to find maximum early fault cases which results in less execution time. The experiment is performed on 21 java application.

Hao et al. [24] made an empirical study on optimal coverage for test case prioritization with intermediate goals. The comparison is also conducted between the optimal and standard approaches in terms of coverage,

fault detection, or execution time. Other than conventional Prioritization approaches, some different techniques have also been preferred for TCP that is motivated to enhance fault rate in lesser time. Haidry et al. [25] evaluated six different industrial systems by using dependency structures for prioritization. These dependencies help in enhancing the fault detection rate in comparison to untreated order, random order, and other techniques. Singha et al applied African Buffalo Optimization for TCP with multiple iteration that reports 62.5% decrease in size and 48.57% in the runtime as compared to original test suite [29]. Learn to Rank algorithm used for TCP and also mention the challenges in [30]. The review article represents Large Language Model based Software Testing and reports 102 relevant studies [31].

3 Proposed methodology

***Definition*:** Multi-objective Test case Prioritization (MOTCP)

***Given*:** A test suite T; All sets of permutations of T, PT; and *n*, the number of objective functions f_i, where $i = 1,2,3 \ldots n$

***Objective*:** To find a permutation set $T' \in$ PT such that T' belongs to pareto-optimal fronts to the given objective function f_i.

Pareto optimal front is a set of solutions that are non-dominated by each other. For example, in three objective functions F_1 , F_2 , F_3 multiple solutions are belonging to PT such that $F_1(T'') > F_1(T')$, $F_2(T'') > F_2(T')$ or $F_3(T'') \geq F_3(T')$ then these solutions T' and T'' are non-dominated to each other (if all functions Fi set is maximized). If there is a solution T'' such that for all $F_i(T'') > F_i(T'), i = 1,2,3$; it shows that solution T'' dominate T'. Regression testing aims to check the impact of modification in software development. If changes are in the code or new functionality are introduced to the application, the test cases that cover these changes must run first. Some of the target points are mentioned below.

- A newly created test case should have high priorities.
- Test cases with modified codes should also have high priorities.
- Test cases dependent upon a high fault-prone area should have high priorities.
- High-fault test cases must have higher priorities.
- A test case having a higher failure history should have high priorities.

The objective is to calculate the fault sensitivity index (FSI) based on Code Complexity Index (CCI) and the Test-case Complexity Index (TCI). Using these two parameters, two-weight metrics W_B and W_T, are generated. Lastly, the area under the curve is calculated to compute the first objective function and cost is calculated for the second objective. Figure 1 shows the basic block diagram of proposed approach.

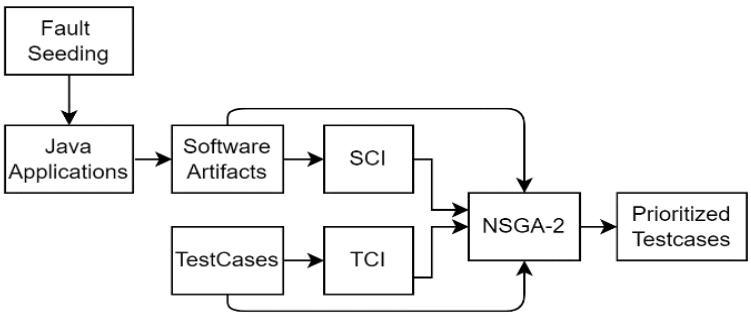

Fig. 1. Block diagram of the proposed technique

3.1 Software complexity index (SCI)

The internal complexity of code and dependency of functionality is used to find prime test cases. McCabe's Cyclomatic Complexity, Object-Oriented Metrics, Line of code (LOC), Code change density (CCD) and other custom metrics are calculated to examine the complexity of code

For complex coding, the software tester specifies weight metrics used to assess code quality and complexity as per its need during regression testing. The weight matrix is used to calculate W_{bi}. All software artefact values are initially calculated for each code block and outliers are detected. Outliers mean that their matrix values are out of the boundary of the threshold range. If the value lies within the threshold range, it is safe; otherwise, it is unsafe and needs testing. So, from

these metrics, if the tester finds an outlier, it marks it 1, otherwise it marks it 0. For example, if coupling between objects (CBO) ranges from 0 to 47 and the acceptable range is 15% to 85% (7.5 to 39). If class C1 has CBO 4 and class C2 has CBO 32, C1 is an outlier (below 15%). So, C1's CBO is marked 1, while C2's CBO is marked 0. Then after, calculate the weight for the respective code block [26] using Eqn. (1).

$$W_{bi} = \frac{\sum_{c \in C}(W_c \times V_{cb})}{\sum_{c \in C}(W_c)} \quad (1)$$

Here, W_c is software tester specified weight and V_{cb} is outliers. In the same way, the weights of all remaining code blocks are calculated. Subsequently, the total weight of each test case is determined by summing up the weights of the code blocks covered by that specific test case, denoted as W_{Bt_i}. After obtaining the total weight of each test case, the cumulative sum of $W^c_{Bt_i}$ is calculated corresponding to the test case order. This cumulated sum $W^c_{Bt_i}$ is further used to calculate the area under the curve (AUC) using the trapezoidal rule, as shown in Eqn. (2). This AUC is nothing but our first objective function, SCI.

$$\int_{x_1}^{x_n} f(x)dx = (x_2 - x_1)\frac{f(x_1) + f(x_2)}{2} + (x_3 - x_2)\frac{f(x_2) + f(x_3)}{2} + \cdots \cdots + (x_n - x_{n-1})\frac{f(x_{n-1}) + f(x_n)}{2} \quad (2)$$

3.2 Test-case complexity index (TCI)

After calculating SCI, TCI is calculated for the test case. This TCI can also be referred to as test case cost. This cost is defined in terms of weights, not execution time.

Weight W_{Ti} is calculated by analysing the behaviour of the test cases. It is calculated based on test case behaviour, considering historical and current status information. The parameters used for W_{Ti} calculation include Code Coverage, where extensive coverage is assumed to explore more faults. Class coverage increases fault detection probability with broader coverage, representing the covered class in a test case. Dependency involves identifying dependencies between neighbouring modules, assigning weights based on dependencies and updating nodes when project modifications occur. Faults indicate the maximum number of faults a test case covers in a specific version. Cost measures the execution time of a test case. New Functionality is a binary value indicating whether a test case covers new functionality. Status history, represented in binary values, indicates test case execution success (1) or failure (0) and reflects the test case age, such as "010", signifying a 66.6% failure rate. To calculate W_{Ti}, the same outlier method is used as described in the previous section but on metrics related to the test case, not code.

$$W_{Ti} = \frac{\sum_{c \in C}(W_c \times V_{ci})}{\sum_{c \in C}(W_c)} \quad (3)$$

The cost of TCI is the second objective function that needs to be optimized [27]. It is calculated by using Eqn. (4), where O_i defines the position of i^{th} test case among all test case orders, and W_{Ti} as in Eqn. (3) is its weight and C_i is the cost of test case i^{th}.

$$F = \sum_{i=1}^{n} \frac{W_{Ti}}{C_i \times O_i} \quad (4)$$

3.3 Implementation of NSGA-2

During the implementation of NSGA-2, the chromosome is encoded in a random sequential order. Then, two parents are selected via tournament selection to produce offspring. The order crossover operation is performed to ensure each test case appears only once. Then after, a swapping mutation is applied.

4 Experimental assessment

This section presents information about the experimental setup. The datasets are represented in Tab. 1 are used to perform experiments. For the experiment, three customized Java projects are used. The first project is small, with an average of 2.4K code and 13 classes. The second project is medium-sized, containing a 2.3 to 6.2k code and classes ranging from 10 to 27. The third big project has a maximum 8.6K code size and 36 maximum classes.

Software artefact metrics are required to find the target points. Mutation operation is used to change the application to introduce any fault, and this faulty version of codes is called mutants. The mutations are done by changing arithmetic operators and conditional statements [28]. Despite using mutation operation using automatic tools, some manual faults seeding is also introduced in codes.

.

Table 1. Project details: P=projects, V=version

Project	P1V0	P1V1	P1V2	P1V3	P2V0	P2V1	P2V2	P2V3	P3V0	P3V1	P3V2	P3V3
Class	8	12	15	18	10	16	21	27	16	23	28	36
Code(K)	1.4	2.2	2.8	3.5	2.3	3.6	4.4	6.2	3.1	4.2	6.4	8.6
Test cases	12	21	30	38	17	33	48	65	28	54	76	102
Faults	15	25	35	45	20	40	60	80	30	60	90	120

4.1 Experimental design

The proposed model is implemented in Python on an Intel i5 processor with 8 GB memory. During the extraction of software artefacts, six data files,

- TestCases_Faults.csv,
- TestCases_Classes.csv,
- Class_Weights.csv,
- TestCases_Weights.csv,
- TestCases_Costs.csv
- TestCases.csv,

are generated. These data files are used as input to calculate our two objective functions. To optimize the objectives, NSGA-2 is used. During the execution of NSGA-2, initial populations are taken equal to twice the number of test cases, the number of iterations used is 500, and the crossover rate and mutation rates taken during execution are 0.5 and 0.25, respectively.

4.2 Evaluation metrics

The model's performance is evaluated in terms of average percentage of fault detection (APFD). APFD is a metric commonly used in software testing to evaluate the effectiveness of test case prioritization techniques. It quantifies how well a prioritization technique orders test cases to detect faults early in the testing process. APFD is calculated based on the positions of faults detected by executed test cases and the total number of test cases and faults. The formula for APFD, Eqn. (5), incorporates these values to produce a single metric that ranges from 0 to 1, with higher values indicating better fault detection effectiveness. [11]

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{nm} + \frac{1}{2n} \tag{5}$$

The other evaluation criteria is the target points analysis, where the algorithm's performance is also calculated on behave of some criteria defined by the software tester. The proposed technique is evaluated against several benchmark algorithms outlined in Tab. 2.

Table 2. Peer techniques for comparison

	M1	M2	M3 [14]	M4 [14]	M5 [24] [15]	M6 M5 [24] [15]	M7 Proposed
	Random	Add. Greedy	3-Opt No Cost	3-Opt With Cost	GA With Cost	GA No Cost	NSGA-2 FSI
APFD Project-P1	61.52 - 68.6	66.38 - 69.05	67.4 - 72.03	65.59 - 75.25	60.37 - 67.89	72.47 - 74.28	73.44 - 76.81
APFD Project-P2	66.8 - 77.94	73.68 - 76.48	76.51 -79.12	70.47 - 75.25	71.55 - 73.91	77.63 - 80.88	81.33 - 82.84
APFD Project-P3	75.79 -81.49	78.89 - 84.54	83.18 -88.11	78.15 - 84.8	76.87 - 85.5	85.93 - 88.58	87.27 - 88.98

5 Result analysis

This section briefly analyses experimental results in terms of the quantitative value and the statistical graph. The algorithms are compared for APFD values and box plots generated corresponding to APFD metrics. The different APFD values are analyzed for Project-P1, Project-P2 add Project-P3 as shown in Tab. 2. For Project-P1, it is reported that algorithm M7 (APFD %: 73.44-76.81) performs better than others. While M6 (APFD %: 72.47-74.28) is the second-best performer, on the other hand M5 (APFD %: 60.37-67.89) performs worst among all.

Similarly, for Project-P2, it is clear that M7 (APFD %: 81.33-82.84) and M6 (APFD %: 77.63-80.88) still performing well among all as previously while the performance of M4 and M5 is almost the same but M1(APFD %: 66.8-77.94) performs worst as compared to others.

Almost same behavior is found in a large-scale project Project-P3, where M7 (APFD %: 87.27-88.98) still performing well and M1(APFD %: 75.79-81.49) perform worst among all, while the performance of M2 and M4 are almost same for Project-P3. The analysis across three projects demonstrates that proposed prioritization methods M7, consistently outperform, indicating their critical role in enhancing fault detection efficiency in complex software systems.

For detailed analysis, the box plots are drawn as shown in Fig. 2. This box plot represents APFD values of all versions of a project; Fig. 2(a) is for small project P1, Fig. 2(b) for medium scale project P2 and Fig. 2(c) for large project data. In the box plot, the red line indicates the median APFD value and the edges of the box represent percentile value on the mark of 25 and 75. By analyzing the above box plots, it can be concluded that M7 perform better compares to other benchmark techniques. In Fig. 2(a) and Fig. 2(b) it is visible that M7 constantly improve its performance compared to others but in Fig. 2(c) median of M7 and M6 appears to be the same.

Slight similarity is perceived in APFD by overlapping of the box plot. For further analysis, a detailed comparison between benchmark algorithms and proposed NSGA-2 is performed in terms of mean value difference using the ANOVA method. Here, Tukey's mean group comparison is used at a 5% significance level. The results are shown in Fig. 3 for all three projects. It is visible from Figs. 3(a, b, c) that M7 has no overlap with M5, M4, M2 and M1 in all three projects. Some overlapping of M7 with M3, M6 is found, but it is clear that the mean value of M7 is greater than M3 and M6 for all projects, which rejects the hypothesis that all mean values are equal because the p-value corresponding to the ANOVA analysis is less than 0.05.

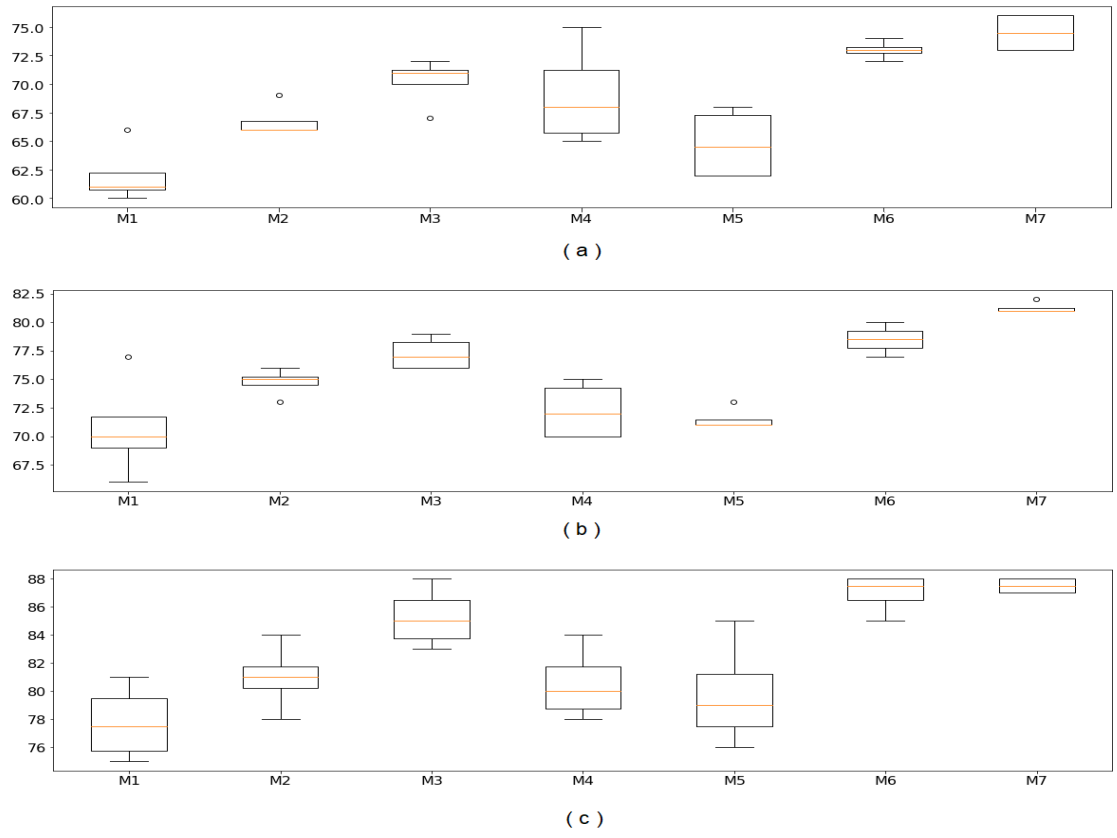

Fig. 2. Boxplot of benchmark techniques: (a) Project-P1, (b) Project-P2, (c) Project-P3

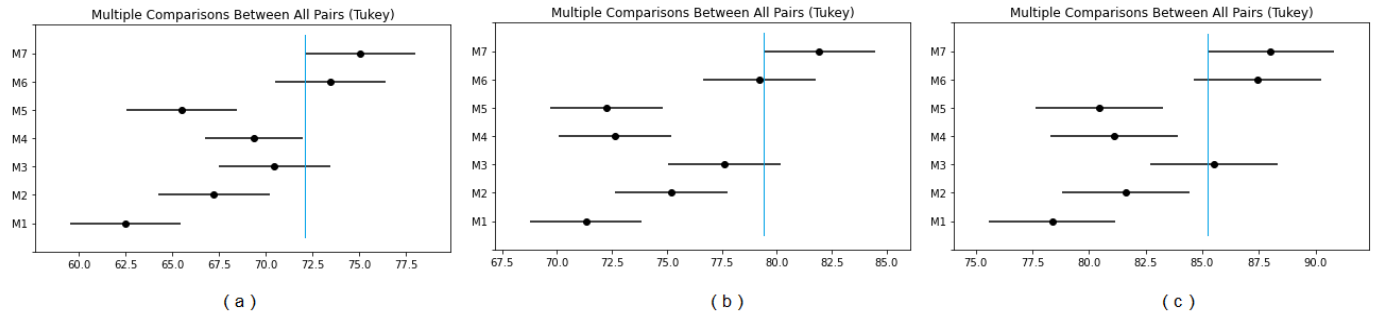

Fig. 3. Tukey test graph comparing the difference of mean: (a) Project-P1, (b) Project-P2, (c) Project-P3

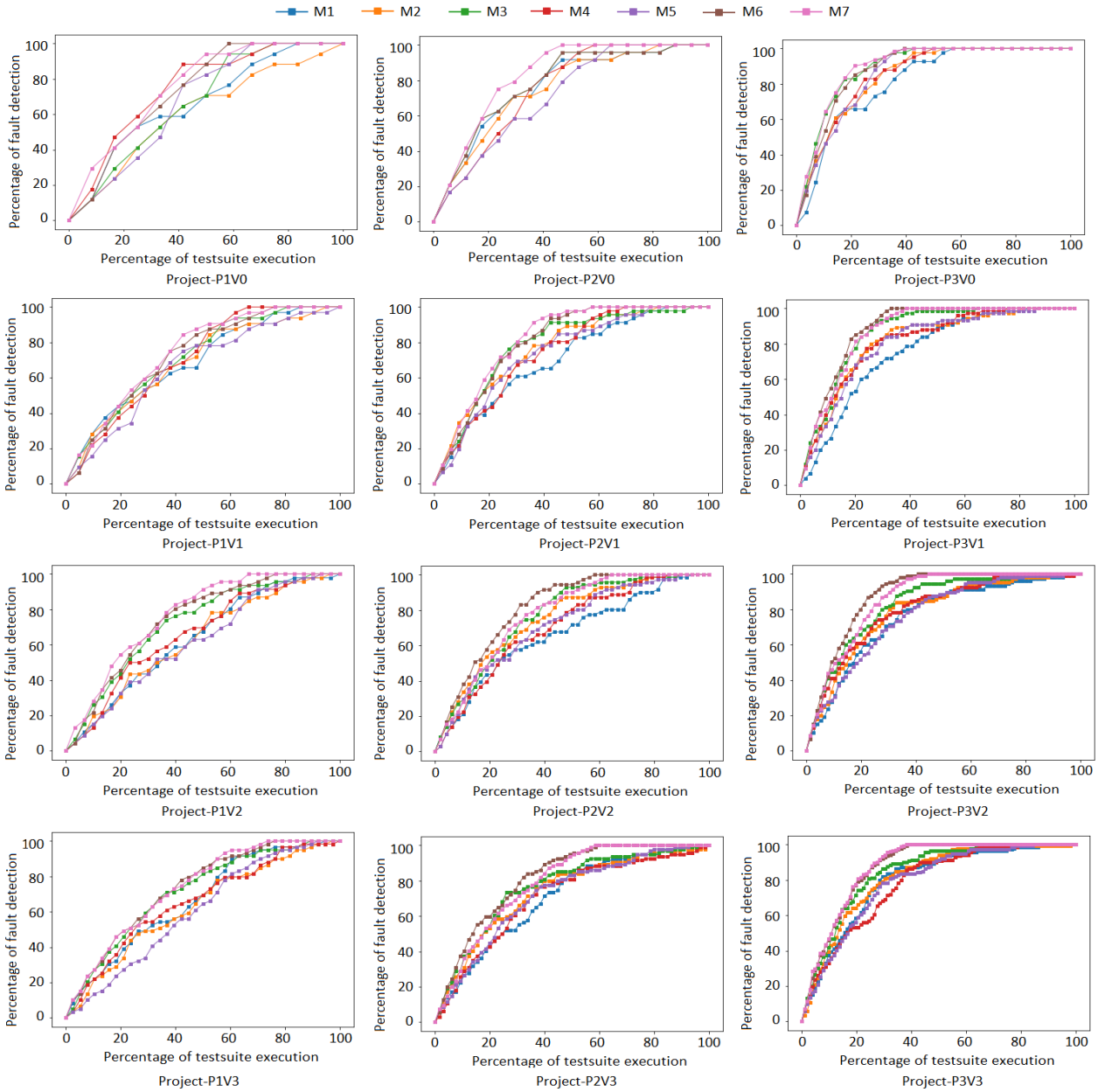

Fig. 4. APFD representation of proposed techniques

Figure 4 illustrates the APFD representation of all techniques, with each column representing a single project and its versions. The graph reveals that techniques leading to fault detection initially tend to have the highest APFD values. Additionally, it is noteworthy that with a small number of test cases, there is a significant gap between the APFD graphs, indicating a substantial difference in APFD values. As the number of test cases increases, e.g., Figs. 4 (Project-P3V2 and Project-P3V3) the graphs become denser, signifying a reduced gap in APFD values.

The APFD graph also suggests that executing all test cases may not be necessary, as several algorithms achieve 100% fault detection before completing all tests. Many flat horizontal lines in the graphs support this observation, indicating the early detection of all faults. This property is further utilized to assess the performance of all algorithms.

The experiments also report the percentage of test cases needed to detect 100% faults. Notably, M7 is the technique that consistently detects all faults early, followed by M6 in this criterion. The minimum percentage of test cases required to cover all faults ranges from 38.24% to 76.19% for M7, while for M6, it is 33.33% to 76.19%.

Reducing test cases also leads to lower execution costs. The experimental analysis conducted across various projects, comparing methods M1 to M7, reveals that M7 excels in reducing costs. Although M7, M6, M5, and M4 all achieve execution cost reductions, M5 stands out by detecting all faults within merely 20% of the total execution time.

Finally, the evaluation is conducted on target points defined by the software tester, as presented in Tab. 3. These table illustrates the coverage of code blocks and test cases within the defined limits, particularly highly complex ones. In this context, the limit is defined as the length of a test case covering all faults, and complexity is considered only if the weights surpass 50% of the maximum, as explained earlier. From the Tab. 3, it is evident that M7 can cover more target points than other benchmark algorithms. Although M7 lags behind M6 in code coverage but it detects 98% of faults compared to M6's 95%. It is noteworthy that M6 operates in a single direction, whereas M7 explores multiple directions in the search space. The algorithm securing the third position is M3, with the covering difference being less than 5% compared to M6 at some points. M2 secures the fourth position, while the performance of M5 and M4 is quite similar, and M1 ranks the least effective among all algorithms.

Table 3. Percentage of target point coverage

	M7	M6	M5	M4	M3	M2	M1
Code Coverage	72.43	76.58	55.24	51.16	70.58	62.48	43.26
Faults	98.10	95.46	76.48	78.32	92.32	86.67	64.26
New functionality	74.63	70.94	57.36	58.28	67.28	54.52	42.54
Test case Status	76.04	66.18	54.58	52.46	61.45	58.26	43.88
Code complexity	84.24	74.43	58.47	54.50	72.12	63.45	48.21
Code change density	78.18	72.32	56.36	52.92	67.45	54.30	44.62

6 Conclusion and future scope

The proposed techniques utilize NSGA2 for optimizing specified objectives and undergo thorough testing across three Java projects of varying scales, including small, medium, and large-scale projects. Upon concluding the performance evaluation of our proposed technique, we handled the research questions posed. In response to **RQ1**, we evaluate the performance of the technique using the APFD value and compare the result the other benchmark algorithms. While addressing **RQ2,** the proposed approach shows a tendency to handle highly complex faults early stage, effectively bounding and handling complex code blocks compared to alternative techniques. Considering **RQ3,** the proposed technique generally surpasses others, with GA demonstrating superior performance. Responding to **RQ4,** the performance suggested techniques vary with project size as for smaller projects, it excels compared to others while for larger projects, performance reaches saturation with marginal increments across versions. Answering **RQ5**, our proposed technique successfully achieves all target points defined by software testers. Additional objective functions, such as fault severity and other coverage parameters, can be incorporated into our approach as future work. Furthermore, including more software artefacts is recommended to enhance accuracy in terms of fault sensitivity.

Conflict of Interest: There is no conflict of interest.

Statement of interest: DATA can be made available on reasonable request.

Funding: No funding was obtained for this study.

References

[1] A. Attaallah, K. Al-Sulbi, A. Alasiry, M. Marzougui, M. W. Khan et al., "Security test case prioritization through ant colony optimization algorithm," Computer Systems Science and Engineering, vol. 47, no.3, pp. 3165–3195, 2023. doi: 10.32604/csse.2023.040259

[2] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," IEEE Trans. Evol. Comput., vol. 6, no. 2, pp. 182–197, Apr. 2002. doi: 10.1109/4235.996017.

[3] F. S. Ahmed, A. Majeed and T. A. Khan, "Value-based test case prioritization for regression testing using genetic algorithms," Computers, Materials & Continua, vol. 74, no.1, pp. 2211–2238, 2023. doi: 10.32604/cmc.2023.032664

[4] R. Mukherjee and K. S. Patnaik, "A survey on different approaches for software test case prioritization," J. King Saud Univ. - Comput. Inf. Sci., vol. 33, no. 9, pp. 1041–1054, Nov. 2021. doi: 10.1016/j.jksuci.2018.09.005.

[5] Y. Tian et al., "Evolutionary Large-Scale Multi-Objective Optimization: A Survey," ACM Comput. Surv., vol. 54, no. 8, pp. 1–34, Nov. 2022. doi: 10.1145/3470971.

[6] Shaheed Zulfikar Ali Bhutto Institute of Science and Technology, Karachi, Pakistan and Q. Et Al., "Test case prioritization techniques in software regression testing: An overview," Int. J. Adv. Appl. Sci., vol. 8, no. 5, pp. 107–121, May 2021. doi: 10.21833/ijaas.2021.05.012.

[7] R. Pan, M. Bagherzadeh, T. A. Ghaleb, and L. Briand, "Test case selection and prioritization using machine learning: a systematic literature review," Empir. Softw. Eng., vol. 27, no. 2, p. 29, Mar. 2022. doi: 10.1007/s10664-021-10066-6.

[8] P. Erik Strandberg, W. Afzal, T. J. Ostrand, E. J. Weyuker, and D. Sundmark, "Automated System-Level Regression Test Prioritization in a Nutshell," IEEE Softw., vol. 34, no. 4, pp. 30–37, 2017. doi: 10.1109/MS.2017.92.

[9] M. Khatibsyarbini, M. A. Isa, D. N. A. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," Inf. Softw. Technol., vol. 93, pp. 74–93, Jan. 2018. doi: 10.1016/j.infsof.2017.08.014.

[10] A. Bajaj and O. P. Sangwan, "A Systematic Literature Review of Test Case Prioritization Using Genetic Algorithms," IEEE Access, vol. 7, pp. 126355–126375, 2019. doi: 10.1109/ACCESS.2019.2938260.

[11] Yu-Chi Huang, Kuan-Li Peng, Chin-Yu Huang "A history-based cost-cognizant test case prioritization technique in regression testing" Journal of Systems and Software, Vol.85/3, pp 626-637, 2012. doi: 10.1016/j.jss.2011.09.063.

[12] Y. Huang, T. Shu, and Z. Ding, "A Learn-to-Rank Method for Model-Based Regression Test Case Prioritization," IEEE Access, vol. 9, pp. 16365–16382, 2021. doi: 10.1109/ACCESS.2021.3053163.

[13] D. K. Yadav and S. Dutta, "Regression test case selection and prioritization for object oriented software," Microsyst. Technol., vol. 26, no. 5, pp. 1463–1477, May 2020. doi: 10.1007/s00542-019-04679-7.

[14] M. Khanna, N. Chauhan, D. Sharma, A. Toofani, and A. Chaudhary, "Search for Prioritized Test Cases in Multi-Objective Environment During Web Application Testing," Arab. J. Sci. Eng., vol. 43, no. 8, pp. 4179–4201, Aug. 2018. doi: 10.1007/s13369-017-2830-6.

[15] D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "A Test Case Prioritization Genetic Algorithm Guided by the Hypervolume Indicator," IEEE Trans. Softw. Eng., vol. 46, no. 6, pp. 674–696, Jun. 2020. doi: 10.1109/TSE.2018.2868082.

[16] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, "Empirical evaluation of pareto efficient multi-objective regression test case prioritisation," in Proceedings of the 2015 International Symposium on Software Testing and Analysis, in ISSTA 2015. New York, NY, USA: Association for Computing Machinery, pp. 234–245, Jul. 2015. doi: 10.1145/2771783.2771788.

[17] B. Ma, L. Wan, N. Yao, S. Fan, and Y. Zhang, "Evolutionary selection for regression test cases based on diversity," Front. Comput. Sci., vol. 15, no. 2, p. 152205, Apr. 2021. doi: 10.1007/s11704-020-9229-3.

[18] U. Geetha, S. Sankar, and M. Sandhya, "Acceptance testing based test case prioritization," Cogent Eng., vol. 8, no. 1, p. 1907013, Jan. 2021. doi: 10.1080/23311916.2021.1907013.

[19] M. Khanna, A. Chaudhary, A. Toofani, and A. Pawar, "Performance Comparison of Multi-objective Algorithms for Test Case Prioritization During Web Application Testing," Arab. J. Sci. Eng., vol. 44, no. 11, pp. 9599–9625, Nov. 2019. doi: 10.1007/s13369-019-03817-7.

[20] A. Bajaj and O. P. Sangwan, "Tri-level regression testing using nature-inspired algorithms," Innov. Syst. Softw. Eng., vol. 17, no. 1, pp. 1–16, Mar. 2021. doi: 10.1007/s11334-021-00384-9.

[21] Z. Li, M. Harman, and R. M. Hierons, "Search Algorithms for Regression Test Case Prioritization," IEEE Trans. Softw. Eng., vol. 33, no. 4, pp. 225–237, Apr. 2007. doi: 10.1109/TSE.2007.38.

[22] Y. Bian, Z. Li, R. Zhao, and D. Gong, "Epistasis Based ACO for Regression Test Case Prioritization," IEEE Trans. Emerg. Top. Comput. Intell., vol. 1, no. 3, pp. 213–223, Jun. 2017. doi: 10.1109/TETCI.2017.2699228.

[23] A. Marchetto, Md. M. Islam, W. Asghar, A. Susi, and G. Scanniello, "A Multi-Objective Technique to Prioritize Test Cases," IEEE Trans. Softw. Eng., vol. 42, no. 10, pp. 918–940, Oct. 2016. doi: 10.1109/TSE.2015.2510633.

[24] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To Be Optimal or Not in Test-Case Prioritization," IEEE Trans. Softw. Eng., vol. 42, no. 5, pp. 490–505, May 2016. doi: 10.1109/TSE.2015.2496939.

[25] S.-Z. Haidry and T. Miller, "Using Dependency Structures for Prioritization of Functional Test Suites," IEEE Trans. Softw. Eng., vol. 39, no. 2, pp. 258–275, Feb. 2013. doi: 10.1109/TSE.2012.26.

[26] R. Lincke, J. Lundberg, and W. Löwe, "Comparing software metrics tools," in Proceedings of the 2008 international symposium on Software testing and analysis, Seattle WA USA: ACM, Jul. 2008, pp. 131–142. doi: 10.1145/1390630.1390648.

[27] R. Mukherjee and K. S. Patnaik, "Prioritizing JUnit Test Cases Without Coverage Information: An Optimization Heuristics Based Approach," IEEE Access, vol. 7, pp. 78092–78107, 2019, doi: 10.1109/ACCESS.2019.2922387.

[28] C. Fang, Z. Chen, and B. Xu, "Comparing logic coverage criteria on test case prioritization," Sci. China Inf. Sci., vol. 55, no. 12, pp. 2826–2840, Dec. 2012. doi: 10.1007/s11432-012-4746-9.

[29] S. Singhal, N. Jatana, A. F. Subahi, C. Gupta, O. I. Khalaf et al., "Fault coverage-based test case prioritization and selection using african buffalo optimization," Computers, Materials & Continua, vol. 74, no.3, pp. 6755–6774, 2023. doi: 10.32604/cmc.2023.032308

[30] A. Ramírez, M. Berrios, J..R. Romero, R. Feldt, "Towards Explainable Test Case Prioritisation with Learning-to-Rank Models" In 2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, pp. 66-69, doi: 10.1109/ICSTW58534.2023.00023

[31] Wang, J., Huang, Y., Chen, C., Liu, Z., Wang, S., & Wang, Q. "Software testing with large language models: Survey, landscape, and vision" IEEE Transactions on Software Engineering, 2024. pp 1-27. doi: 10.1109/TSE.2024.3368208

Kamal Garg obtained his BTech from APJ Abdul Kalam Technical University in Lucknow and his MS from BITS Pilani. He has been working with Tata Consultancy Services (TCS) as a Senior Consultant for the past 13 years and is currently settled in London, UK. He is a research scholar at GLA University in the Department of Computer Engineering and Applications. His areas of interest include software testing, regression testing, machine learning, and soft computing. Kamal is an IT Consultant with strong experience in the Delivery, Testing and consulting of IT systems for the last 18+ years.

Shashi Shekhar received the MTech. degree in Computer Science from UPTU Lucknow, India. He received the Ph.D. degree from GLA University Mathura, India. He has 22 years of teaching experience and currently, he is working as an Associate Professor in Department of computer Engineering and Applications, GLA University, Mathura, India. His research interests include Software Testing, Regression Testing, Machine Learning, Soft Computing and Natural Language Processing.