Fault-Based Regression Test Case Prioritization

Sourav Biswas, Aman Bansal, Pabitra Mitra, and Rajib Mall

***Abstract*—We propose a set of four novel fault-based regression test case prioritization (TCP) techniques for object-oriented programs. We seed bugs into a program to create large number of mutants. We execute each mutant with the originally designed test suite. From this, we record the number of mutants for which a test case fails. Based on this, we prioritize the test cases using four base fault-based prioritization techniques that we have proposed. Finally, we combine the results of our four base prioritizers using three ensemble methods. We have conducted experimental studies to determine the effectiveness of our proposed approaches. Our experimental results show that our proposed TCP techniques exhibit superior performance over related techniques.**

***Index Terms*—Mutation testing, object-oriented (OO) programs, regression test case prioritization (TCP), regression testing.**

I. Introduction

REGRESSION testing is popularly being used to verify whether a recent code change has affected any correctly working parts of a program. During regression testing, existing test cases are rerun to check whether the unchanged functionalities are working as before. However, due to cost and time constraints, it is usually impractical to run the entire test suite after every modification of the software [1]. Therefore, selection of an optimal regression test suite is important. Even when using an appropriate regression test suite, it has been reported that regression testing accounts for half of the total software maintenance costs [2]. To make regression testing cost-effective and time efficient, in addition to regression test selection [1], [3], [4], [5], test suite minimization [6], [7], and test case prioritization (TCP) [8], [9], [10], [11] techniques are also being deployed. TCP attempts to order the test cases to maximize the rate of fault detection or code coverage. Another use of TCP is that when only a part of the regression test suite can be run due to cost or time constraints, then the required number of test cases to be used can be picked up from the top of the prioritized list of test cases.

TCP techniques usually assign scores to the test cases based on either their fault detection capabilities or their code coverage capabilities. The test cases are prioritized based on the scores assigned to them. Various TCP techniques have been proposed based on different criteria, such as code coverage [8], [9], requirements coverage [12], [13], and different types of model coverage [14], [15], [16]. However the performance of those techniques are not very encouraging [15], [17], [18]. For example, it has been reported that in case of the TCP technique reported by Rothermel et al. [9], execution of 50% of the test suite on an average detects 40% bugs.[1]

In this context, we first propose a set of four novel fault-based TCP techniques for object-oriented (OO) programs and then ensemble the computed results using three ensembling techniques for improved performance. An essential idea behind our fault-based TCP techniques is that we first seed bugs into a program to create a large number of mutated versions (mutants) of the original program. The mutants are executed with each test case of the test suite, and the output results are recorded in a fault matrix (FM). Using the generated FM, we have proposed four TCP techniques: optimized bug detection based prioritization technique, hard to detect bugs based prioritization, expected bug detection based prioritization, and Bayesian probabilistic prioritization. Finally, we ensemble the results of these four different techniques using three ensemble methods: averaging, Kendall Tau ranking [19], and Schulze's algorithm of ranking [20]. All our four base TCP techniques are mutation-based. Our optimized bug detection based prioritization technique is based on a direct measure of faults detected by the test cases. The second technique, hard to detect bugs based prioritization, aims to prioritize test cases based on the bugs that are detected by very few test cases. The third technique, expected bug detection based prioritization technique, orders all the test cases based on the expected value of bugs detected by each test case. Our fourth base prioritizer, Bayesian probabilistic prioritization technique, uses Baye's theorem to order the test cases based on their potential to expose bugs.

In the following, we briefly summarize the main contributions of our work.

1) We have proposed a set of four novel base fault-based TCP algorithms for OO programs to achieve increased bug detection rates.
2) We have proposed three novel ensemble TCP techniques to further improve the rate of bug detection.
3) We have experimentally evaluated the performance of all our proposed techniques using several open source projects and have compared the performance with existing mutation-based and coverage-based state-of-the-art techniques.

Manuscript received 15 November 2021; revised 11 April 2022 and 5 May 2022; accepted 4 September 2022. Date of publication 22 September 2022; date of current version 1 September 2023. Associate Editor: Ruizhi Gao. *(Corresponding author: Sourav Biswas.)*

The authors are with the Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur, WB 721302, India (e-mail: bsws.sourav@gmail.com; amanbansal18may@gmail.com; pabitra@cse.iitkgp.ac.in; rajib@cse.iitkgp.ac.in).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TR.2022.3205483.

Digital Object Identifier 10.1109/TR.2022.3205483

[1]In this article, we have used the terms "bug" and "fault" interchangeably for the sake of readability.

TABLE I
FAULT MATRIX FOR OPTIMIZED BD-TCP

	Fault0	Fault1	Fault2	Fault3	Fault4
TC_0	1	0	0	1	1
TC_1	1	1	0	0	0
TC_2	0	0	1	1	1

This article is organized into sections as follows. In Section II, we provide an overview of few basic concepts based on which our proposed techniques have been developed. Related work is reviewed in Section III. Our proposed techniques have been presented in Section IV. In Section V, we discuss our experimental results. Section VII concludes the article.

II. BASIC CONCEPTS

In this section, we provide an overview of a few basic concepts. These have been used in describing our proposed technique.

A. Fault Matrix

A FM is an order $n \times m$ matrix, where *n* indicates the total number of test cases in a test suite and *m* indicates the total number of bugs are present in a program. An example FM is shown in Table I. In the FM, TC_i represents the ith test case and $Fault_j$ represents the jth fault. The fault detection capability (FDC) of a test case is represented in FM using the following equation:

$$\text{FM}_{ij} = \begin{cases} 1, & \text{if the fault } j \text{ is exposed by test case } i \\ 0, & \text{otherwise} \end{cases}. \quad (1)$$

B. TCP Performance Metric

To evaluate the performance of a TCP technique, Elbaum et al. [8] proposed a metric named average percentage of faults detected (APFD). APFD metric indicates how rapidly a prioritized test suite can detect faults during its execution. For a given test suite ordering T, APFD metric is given by the following expression:

$$\text{APFD} = 1 - \frac{\sum_{i=1}^{m} \text{TF}_i}{n * m} + \frac{1}{2 * n} \quad (2)$$

where n is the total number of test cases present in a test suite T, m indicates the total number of bugs exposed by the test suite T, and TF_i represents the position of the first test case in the test suite T which exposes fault i. A higher value of APFD indicates faster rate of fault detection.

III. RELATED WORK

Researchers have proposed a large number of TCP techniques [8], [9], [12], [13], [14], [15], [16]. These TCP techniques are based on several approaches, such as coverage-based [8], [9], requirements-based [12], [13], and model-based approaches [14], [15], [16].

Rothermel et al. [9] proposed several TCP techniques based on code coverage and fault exposing potential of test cases. Elbaum et al. [8] extended Rothermel's work [9] by proposing function-level prioritization techniques. Function-level prioritization techniques prioritize test cases based on the total number of functions covered. Wong et al. [21] proposed a greedy TCP technique to prioritize test cases based on the achieved code coverage.

Korel et al. [14] proposed a TCP technique using the state model. State model analysis is inexpensive as compared to the analysis of the whole program. They presented two model-based prioritization techniques: selective test prioritization and dependence-based test prioritization. Panigrahi et al. [15], [16] proposed a model-based TCP technique for OO programs. They constructed an extended system dependency graph (ESDG) [22] for a given program and reflected on it any changes made to the program. They used program slicing to identify the impact of the changes on the constructed ESDG. A test case which covered larger number of affected elements was accorded higher priority. Experimental results showed that their approach achieved 25.07% improvement over traditional code-based TCP techniques [8], [9].

Vescan et al. [23] proposed an ant colony optimization algorithm-based TCP technique (TCP-ANT). Their approach considers the number of not yet detected faults, test case execution time, and fault severity to prioritize the test cases. Thus, a test case that detects larger number of yet-undetected severe faults in lower execution time is accorded higher priority. They used the APFD metric to evaluate their techniques.

Shin et al. [24] proposed a diversity-aware mutation-based TCP technique. Their approach is based on the distinguished behavior of one mutant from other mutants. They proposed both the traditional and diversity-aware approaches to prioritize test cases. They named their first approach Greedy, Kill (GRK). In GRK, in each iteration, an unranked test case is prioritized depending upon whether it can kill new mutants that the test cases ranked in previous iterations have not yet killed. Similarly, in their Greedy, Distinguish (GRD) approach, an unranked test case is prioritized in each iteration if it can distinguish additional mutants that the test cases ranked in all previous iterations have not yet distinguished. In an extended approach, a test case is prioritized if it increases the weighted sum of GRK and GRD and named it as HYBrid-weight (HYB-w). In these three approaches, if two cases give the same results, then any one of the test cases is included in the priority list. Zhou et al. [25] proposed a TCP technique based on the measurement of dissimilarity among test cases using the intuition of adaptive random testing (ART) [26]. To measure the dissimilarity among test cases, they proposed a disparity metric.

Several TCP techniques have been proposed based on soft computing approaches [27], [28], [29]. Krishnamoorthi et al. [27] applied genetic algorithm to a code-coverage based TCP technique. Their TCP technique additionally considered test case execution times for prioritizing test cases. Their experimental evaluation showed that their technique achieved 120% increase in APFD value over other TCP techniques [8], [9], [21]. On the other hand, Gökçe et al. [28] introduced a model-based TCP technique using a neural network (NN) based classification. They split the test cases into five groups (from high to low priority) using a multilayer perceptron. Chen et al. [29]

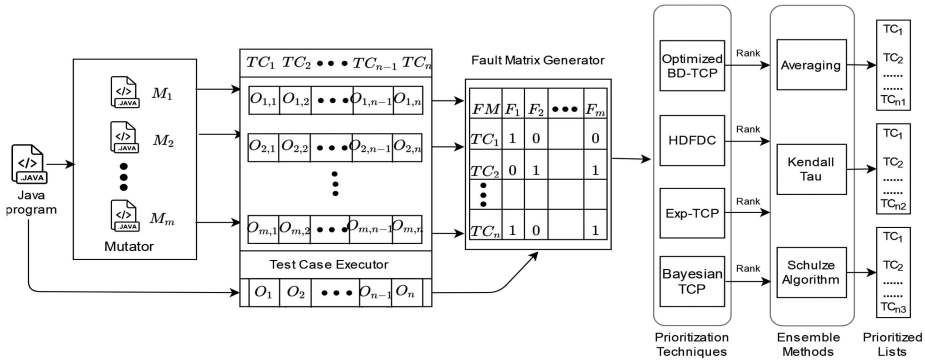

Fig. 1. Schematic representation of fault-based TCP.

introduced a clustering-based TCP technique for OO programs. For improving the effectiveness of their technique, they applied clustering algorithms to generate adaptive random sequences (ARS) of test cases in which similar types of test cases were put into the same cluster.

Mirarab et al. [30] proposed a Bayesian probabilistic TCP approach. Their approach considers code coverage information, fault proneness, and changes to the software to prioritize the test cases. Zhao et al. [31] proposed a Bayesian network based clustering approach to prioritize test cases. In their technique, test cases are clustered into different groups based on code coverage similarity. Test cases of each cluster are then prioritized based on their probability of failure.

IV. Proposed Techniques: Fault-Based TCP

In this section, we present fault-based TCP techniques for Java programs. Though our techniques are designed for Java programs, these can easily be generalized for programs written in other languages. Fig. 1 schematically portrays the essential schema of our proposed techniques.

As shown in Fig. 1, first, we generate a large number of mutated versions (M_1, M_2,..., M_m) for a given Java program. Each mutated version contains a single mutant. The original java program is automatically executed with the given test cases (TC_1, TC_2,..., TC_n) to produce the output files (O_1, O_2,..., O_{n-1}, O_n) corresponding to the test cases. All mutated versions are then executed with the given test suite and corresponding output files are stored. It can be observed from Fig. 1 that the mutated version M_2 is run by using the test cases TC_1, TC_2,..., TC_n to produce the output files $O_{2,1}$, $O_{2,2}$,..., $O_{2,n-1}$, $O_{2,n}$, respectively. These output files are then compared with the original program's output files and each file is flagged either zero or one depending on the result of the match, which indicates the success or failure of the test case. If the output of the original java program is the same as the output of the mutated version of the java program, then the test case is considered to have passed and is assigned a score 0, else the test case fails and is assigned a score of 1. Based on these results, an FM is constructed. The details of how a FM is constructed are given in Section V-A.

Based on the constructed FM, we propose four basic fault-based prioritization techniques: optimized version of the bug detection based TCP (optimized BD-TCP), hard to detect bugs and fault detection capability based TCP (HDFDC), expected value of bug detection based TCP (Exp-TCP), and Bayesian probabilistic based TCP (Bayesian TCP). These four prioritization techniques produce four different rank lists of the test cases. These rank lists are given as input to three different ensemble ranking methodologies to generate three different combined rank lists: averaging, Kendall Tau [19], and Schulze algorithm [20] based ranking schemes.

In the following, we first present our four fault-based prioritization techniques and then present our approaches to ensemble the results produced by the four base TCP techniques.

A. Our Base Fault-Based TCP Techniques

In this subsection, we present our four base TCP techniques: optimized BD-TCP, HDFDC, Exp-TCP, and Bayesian TCP. The outputs generated by these four techniques are used by ensemble techniques to produce combined prioritization lists.

Optimized BD-TCP: We have named our first technique optimized BD-TCP. Optimized BD-TCP is the optimized version of our BD-TCP. We have not considered BD-TCP as a base prioritizer in our ensemble technique because optimized BD-TCP subsumes BD-TCP. We first briefly discuss the working of BD-TCP technique as our optimized BD-TCP technique has been developed based on it.

BD-TCP prioritizes test cases based on the total number of bugs for which a test case fails. The test cases that fail for a larger number of mutants are accorded higher priority. In BD-TCP, when two test cases detect identical number of bugs, their interraking is randomly assigned. This, however, may not always result in the best priority order. As an illustration, consider the following example.

From Table I, it can be observed that TC_0 and TC_2 are detecting the same number of bugs (three each). A run of BD-TCP will generate the priority order: TC_0, TC_1, TC_3. This order gives an APFD value of 0.63. However, if the priority order is: TC_2, TC_1, TC_0, then its APFD value will be 0.7. From this, we can infer that BD-TCP does not always produce the best ordering of the test cases. To improve this situation, we have proposed

an optimized BD-TCP technique. In optimized BD-TCP, a test case gets a higher rank, if it detects more number of scarcely detected bugs. In the example given in Table I, TC_0 detects $Fault0$, $Fault3$, and $Fault4$, each of which is detected by two test cases. TC_3 also detects $Fault3$ and $Fault4$, each of which is detected by two test cases, but it also detects $Fault2$, which is detected by only one test case. In the optimized BD-TCP, TC_3 gets a higher rank than TC_1 because TC_3 is detecting a fault that is not detected by any other test case.

Algorithm 1 depicts an algorithmic representation of optimized BD-TCP. Optimized BD-TCP takes FM as its input. In Step 2 of the algorithm, a list *YetToRank* is created. Each entry of the list *YetToRank* contains two fields: *TestCase*, which is the identifier of the test case and *BugsCount*, which is the number of bugs detected by the corresponding test case. The *YetToRank* contains test cases in the *TestCase* field and the corresponding detected bug counts of the test cases in the *BugsCount* field. In Steps 3–5, *YetToRank* is initialized with all the available test cases in *TestCase* field and with 0 in the *BugsCount* fields. Steps 7–15 update the corresponding *BugsCount* field of each test case of *YetToRank* list with the number of bugs it detects. In the steps from 16 to 24, a function (*Compare()*) is called for each pair of test cases to determine their relative priority. Algorithm 2 represents the *Compare()* function which compares the bug detection capability of two test cases. In Steps 2–24 of Algorithm 2, the number of bugs detected by the two test cases is calculated. In Steps 27–32, the number of bugs being detected by the test cases is compared and either TRUE or FALSE is returned depending on which test case detects more bugs. A test case that detects more number of bugs is accorded higher priority. If both the test cases are detecting the identical number of bugs, then in Steps 33–43 of the Algorithm 2, the two test cases are compared based on the type of bugs they are detecting. A test case that detects more number of scarcely detected bugs gets higher priority. After this, in Step 25 of the Algorithm 1, the index of the first test case present in *YetToRank* is pushed into the *Prioritized List* and in Steps 27–33, all the values in the column of FM corresponding to all the faults that are detected by the prioritized test case k are set to 0. Finally, in Step 36, optimized BD-TCP algorithm returns the *Prioritized List*. The worst case time complexity of our optimized BD-TCP algorithm is $\mathcal{O}(n^3m)$ [32]. Here, n is the number of test cases and m is the number of faults.

HDFDC: We have named our third TCP technique as HDFDC prioritization technique. HDFDC divides the test cases into three different groups. The first group (Group 1) contains the prioritized list of test cases that detect "hard to detect" faults. A fault detected by only one of the test cases is denoted as "hard to detect" fault. The remaining test cases that are not detecting "hard to detect" faults comprise the second group (Group 2). The test cases that detect only the bugs that have already been detected by some test cases of Group 2 comprise the third group (Group 3). Thus, we can view the test cases in Group 3 as the set of redundant test cases. Both Group 2 and Group 3 test cases are prioritized using a FDC metric. Group 1 test cases have the highest priority in the final prioritization list. Test cases in Group 2 have the next priority and Group 3 test cases have the lowest priority. The details of our HDFDC technique can be found in [33].

Algorithm 1: Optimized BD-TCP.

Require: Fault matrix FM of size $N \times M$, N is the total number of test cases, and M is the total number of faults

1: Initialize an empty list, *Prioritized List*. // Prioritized list of test cases
2: Create an empty list, *YetToRank*, each element has *TestCase* and their corresponding *BugsCount* as its fields
3: **for** all Row $i \in$ FM **do**
4: Insert (i,0) in *YetToRank*
5: **end for**
6: **while** *YetToRank* is not empty **do**
7: **for** all $x \in$ *YetToRank* **do**
8: TC $= x$.TestCase
9: x.BugsCount $= 0$
10: **for** all $j \in 1...M$ **do**
11: **if** FM[TC][j] $== 1$ **then**
12: x.BugsCount++;
13: **end if**
14: **end for**
15: **end for**
//Sorting algorithm which uses Algorithm 2
16: **for** all $i \in$ 1...length(*YetToRank*-1) **do**
17: **for** all $j \in i$ + 1... length(*YetToRank*) **do**
18: // *Compare()* represented in Algorithm 2 2
19: flag $=$ *Compare*(*YetToRank*[i].TestCase, *YetToRank*[j].TestCase,FM)
20: **if** flag $==$ **false then**
21: swap(*YetToRank*[i],*YetToRank*[j])
22: **end if**
23: **end for**
24: **end for**
25: $k =$ *YetToRank*[0].TestCase // Test case which detects most number of bugs
26: Insert k in the *Prioritized List*.
27: **for** all $i \in 1...M$ **do**
28: **if** FM[k][i]$==$1 **then**
29: **for** all $j \in 1...N$ **do**
30: FM[j][i] $= 0$
31: **end for**
32: **end if**
33: **end for**
34: Remove the pair *YetToRank*[0] from the list *YetToRank*.
35: **end while**
36: return *Prioritized List*

Exp-TCP: We propose a technique to generate a prioritized list of test cases based on the expected value of bug exposing potential of a test case. We create an $N \times N$ matrix (N being the number of test cases to be ordered) whose rows represent test cases and columns represent ranks. The entry corresponding to ith row and jth column contains the expected number of yet

Algorithm 2: *Compare()* function for sorting.

Require: Fault matrix FM, Test cases TC_a, and TC_b
1: Create an empty list *Final Order* which will contain the final ordering of the two input test cases.
2: Create two empty lists $BugCountList_a$ and $BugCountList_b$.
3: **for** all $i \in 1...$ Total number of bugs in FM **do**
4: **if** FM[TC_a][i] == 1 **then**
5: Counter = 0
6: **for** all $j \in 1...$ Total number of test cases in FM **do**
7: **if** FM[j][i] == 1 **then**
8: Counter++
9: **end if**
10: **end for**
11: Push the value of counter into $BugCountList_a$.
12: **end if**
13: **end for**
14: **for** all $i \in 1...$ Total number of bugs in FM **do**
15: **if** FM[TC_b][i] == 1 **then**
16: Counter = 0
17: **for** all $j \in 1..$ Total Number of test cases in FM **do**
18: **if** FM[j][i] == 1 **then**
19: Counter++
20: **end if**
21: **end for**
22: Push the value of counter into $BugCountList_b$.
23: **end if**
24: **end for**
25: Sort the two lists $BugCountList_a$ and $BugCountList_b$ in ascending order using any sorting algorithm.
26: //$BugCountList_a$ has length strictly greater than length of $BugCountList_b$
27: **if** length($BugCountList_a$) > length($BugCountList_b$) **then**
28: return **true**
29: **else**
30: //$BugCountList_a$ has length strictly less than length of $BugCountList_b$
31: **if** length($BugCountList_a$) < length($BugCountList_b$) **then**
32: return **false**
33: **else**
34: //else part handles the case where length of $BugCountList_a$ is same as length of $BugCountList_b$
35: **for** all $i \in 1...$ length($BugCountList_a$) **do**
36: **if** $BugCountList_b[i] > BugCountList_a[i]$ **then**
37: return **true**
38: **end if**
39: **if** $BugCountList_a[i] > BugCountList_b[i]$ **then**
40: return **false**
41: **end if**
42: **end for**
43: return **true**
44: **end if**
45: **end if**

Algorithm 3: Exp TCP Algorithm.

Require: Fault matrix FM
1: Define FM to be of size $N \times M$ where N is the number of test cases and M is the number of faults.
2: Create a 2-D matrix, *ExpMat* of size $N \times N$ and initialize all of its elements to 0.
3: Create an empty list named *Prioritized List*.
4: Create a list, *NumberofTestCases* of size M.
5: For each fault f_i in FM, calculate the number of test cases that detect f_i and put that value in the ith position of the list *NumberofTestCases*.
6: **for** all *i*: 1 to M, *i* represents the faults **do**
7: temp = *NumberofTestCases*[i]
8: **for** all *x*: Test cases in FM such that FM[*x*][*i*] = 1 **do**
9: **for** all *j* : 1 to N **do**
10: $ExpMat[x][j] = ExpMat[x][j] + \frac{\binom{N-\text{temp}}{j-1}}{\binom{N-1}{j}}$
11: **end for**
12: **end for**
13: **end for**
14: Create an empty list *YetToRank*.
15: **for** all $i \in 1...N$ **do**
16: Insert *i* into the list *YetToRank*.
17: **end for**
18: **for** all $i \in 1...N$ **do**
19: temp = *YetToRank*[0]
20: **for** all j $\in$ *YetToRank* **do**
21: **if** *ExpMat*[j][N - *i* + 1] < *ExpMat*[temp][N - *i* + 1] **then**
22: temp = *j*
23: **end if**
24: **end for**
25: Insert temp into *Prioritized List*
26: Remove temp from *YetTorank*
27: **end for**
28: Reverse the *Prioritized List*
29: return *Prioritized List*

undetected bugs that ith test case detects while being in jth position.

In Algorithm 3, Steps 1–13 create and populate 2-D matrix *ExpMat* of size $N \times N$, each cell in xth row and yth column contains the expected value by which APFD will get affected if the xth test case is ranked at yth position. For a bug (say *B*) being detected by a test case (say *T*), the algorithm calculates the probability of *B* being detected for the first time by the test case *T* for every position of *T* in 1,2,...,N. Using the property of linearity of expectation [34], the algorithm calculates the value for each cell in *ExpMat* by adding the contribution of each bug detected by a test case at every position, i.e., 1 to N. Assuming that all possible rankings of N test cases to be equally probable, if the ith fault is detected by the jth test case when it is present in the kth position, it will decrease the APFD value by $k * \frac{\binom{N-\text{temp}}{k-1}}{\binom{N-1}{k} * k * N * M} = \frac{\binom{N-\text{temp}}{k-1}}{\binom{N-1}{k} * N * M}$, where temp is the number of test cases that detect fault i. In Steps 18–27, the test cases are ranked in reverse order. In the lth iteration of the for loop, the algorithm selects the test case to be ranked in $(N - l + 1)$st position by selecting the test case from $YetToRank$ which decreases the APFD value by the least amount. In Step 28, the algorithm reverses the *Prioritized List* as that was created in reverse order. Finally in Step 29,

TABLE II
FAULT MATRIX TO ILLUSTRATE BAYESIAN TCP

	Fault0	Fault1	Fault2	Fault3	Fault4	Fault5	Fault6	Fault7
TC_0	✓	✓	✓	✓	✓		✓	
TC_1		✓		✓	✓			
TC_2						✓		
TC_3			✓	✓	✓			✓
TC_4		✓		✓	✓			✓
TC_5				✓				✓

Prioritized List is returned. The worst case time complexity of the EXP-TCP algorithm is $\mathcal{O}(n^2m)$ [32]. Here, n is the number of test cases and m is the number of faults.

Bayesian TCP: This technique is based on Bayesian probabilistic calculation. The test cases are prioritized based on the probability of a test case detecting a bug. For a test case TC_i, $P(\text{TC}_i/F)$ is the probability that for a given fault *F*, the test case TC_i will detect it. It can be calculated using the following equation:

$$P(\text{TC}_i/F) = P(\text{TC}_i \cap F)/P(F) \tag{3}$$

where, *P*(*F*) expands to $P(F/\text{TC}_1) * P(\text{TC}_1) + P(F/\text{TC}_2) * P(\text{TC}_2) + P(F/\text{TC}_3) * P(\text{TC}_3) + \cdots + P(F/\text{TC}_n) * P(\text{TC}_n)$ and $P(F/\text{TC}_i)$ is the probability that fault *F* is detected by test case TC_i.

The average probability score of each TC_i for all faults

$$\frac{\sum_{j=1}^{n} P(\text{TC}_i/F_j)}{n}. \tag{4}$$

We have prioritized the test cases based on the probability scores computed by using (4).

We now illustrate our Bayesian TCP technique using an example. Consider the FM of Table II. Using this FM, we can calculate $P(\text{TC}_1/F_3)$ as follows:

$$P(\text{TC}_1/F_3) = P(\text{TC}_1 \cap F_3)/P(F_3)$$

$$\text{where,} \quad P(\text{TC}_1 \cap F_3) = \frac{1}{13 * 8}$$

and $P(F_3) = P(F_3/\text{TC}_0) * P(\text{TC}_0) + P(F_3/\text{TC}_1) * P(\text{TC}_1) + P(F_3/\text{TC}_2) * P(\text{TC}_2) + P(F_3/\text{TC}_3) * P(\text{TC}_3) + \cdots + P(F_3/\text{TC}_5) * P(\text{TC}_5)$ which comes out to be $\frac{5}{13*8}$.

So

$$P(\text{TC}_1/F_3) = P(\text{TC}_1 \cap F_3)/P(F_3) = \frac{\frac{1}{13*8}}{\frac{5}{13*8}} = \frac{1}{5}.$$

Our Bayesian TCP is represented in Algorithm 4. In Steps 3–7, for each test case, the average probability of detecting every fault is calculated and stored in the list *AP*. In Steps 8–17, the test cases are compared based on their average fault detection probability. In each iteration of the outer for loop (in Step 8), the test case with maximum average probability is selected and inserted into the *Prioritized List*. In Step 18, the *Prioritized List* is returned. The worst case time complexity of the Bayesian TCP algorithm is $\mathcal{O}(n(n+m))$ [32]. Here, n is the number of test cases and m is the number of faults.

B. Ensembling

In this subsection, we present three techniques for ensembling the results of the four base prioritizers.

Algorithm 4: Bayesian TCP.

Require: Fault matrix FM
1: Initialize a new list *Average Probability, AP* of size equal to the number of test cases in FM with value 0.
2: Create a new empty list *Prioritized List* to contain final prioritized order of test cases.
3: **for** each test case TC_i **do**
4: **for** each fault F_j **do**
5: $AP[i] = AP[i] + \frac{P(\text{TC}_i/F_j)}{\text{Number of Bugs in FM}}$
6: **end for**
7: **end for**
8: **for** all $i \in 1$...Number of test cases in FM **do**
9: MaxProbabilityTC $= 1$ // Represents the test case with maximum probability
10: **for** all $j \in 1$...Number of test cases in FM **do**
11: **if** $AP[j] > AP[\text{MaxProbabilityTC}]$ **then**
12: MaxProbabilityTC $= j$
13: **end if**
14: **end for**
15: $AP[\text{MaxProbabilityTC}] = 0$
16: Insert MaxProbabilityTC into *Prioritized List*
17: **end for**
18: return *Prioritized List*

TABLE III
ILLUSTRATION OF AVERAGING METHOD

	TC_1	TC_2	TC_3	TC_4	TC_5
$Bayesian-TCP$	2	1	5	3	4
$Exp-TCP$	1	4	2	5	3
$Averaging$	1.5	2.5	3.5	4	3.5

Averaging ensemble: Averaging method of ensembling essentially averages the ranks worked out by the different base prioritizers. The algorithm takes N ($N >= 2$) prioritized lists as its input and gives out an ensemble of these. Table III represents an example of our averaging ensemble method. Here, two prioritizers, Bayesian and Exp-TCP, generate two separate rank lists of test cases. Averaging ensembling method averages the rank lists of Exp-TCP and Bayesian prioritizers and generates a new rank list. In case of a tie in ranking among some test cases, averaging method assigns ranks to these test cases randomly. In Table III, both TC_3 and TC_5 get the same rank after averaging. In this case, any one of them is randomly assigned higher rank.

Averaging ensemble algorithm is presented in Algorithm 5. In Steps 2–6, average ranks of all test cases are computed and stored in the list *Average Rank*. In Steps 11–17, a prioritized order of test cases is created based on the computed average ranks. In Step 18, *Prioritized List* is returned. The worst case time complexity of the averaging ensemble algorithm is $\mathcal{O}(n^2)$ [32], where n is the number of test cases.

Kendall Tau ranking: Kendall Tau ranking metric [19] is computed based on the number of pairwise disagreements between two rank lists. Consider two rank lists: List 1 and List 2. Suppose, in List 1, test case 1 ranks higher than test case 3 but in List 2, test case 1 ranks lower than test case 3. In this case, test cases 1 and 3 are said to form a disagreement pair or conflicting pair. In

Algorithm 5: Averaging Ensemble.

Require: K Prioritized Lists named PL_1 to PL_K from different ranking algorithms, each list of size N

1: Initialize a list, *Average Rank* of size N with all zeros.
2: **for** *i* in 1...K **do**
3: **for** *j* in 1...N **do**
4: *Average Rank*[PL_i[*j*]] += *j*/K
5: **end for**
6: **end for**
7: Initialize a list, *Prioritized List* of size N with zero in each cell.
8: **for** *i* in 1...N **do**
9: *Prioritized List*[*i*] = *i*
10: **end for**
11: **for** *i* in 1...N-1 **do**
12: **for** *j* in 1...N **do**
13: **if** *Average Rank*[*Prioritized List*[*i*]] > *Average Rank*[*Prioritized List*[*j*]] **then**
14: swap(*Prioritized List*[*i*],*Prioritized List*[*j*])
15: **end if**
16: **end for**
17: **end for**
18: return *Prioritized List*

Algorithm 6: Kendall Tau Ensemble.

Require: Prioritized Lists from two ranking algorithms ($PrioritizedList_A$, $PrioritizedList_B$) each of size N, fault matrix FM

1: Create two lists, $TestCaseRank_A$ and $TestCaseRank_B$ each of size N.
2: **for** all *i* in 1...N **do**
3: $TestCaseRank_A[PrioritizedList_A[i]] = i$
4: $TestCaseRank_B[PrioritizedList_B[i]] = i$
5: **end for**
6: Create an empty list of lists, $Conflicts$
7: **for** all *i* in 1...$N-1$ **do**
8: **for** all *j* in *i* + 1... N **do**
9: temp$_a$ = $TestCaseRank_A$[*i*] - $TestCaseRank_A$[*j*]
10: temp$_b$ = $TestCaseRank_B$[*i*] - $TestCaseRank_B$[*j*]
11: **if** temp$_a$ * temp$_b$ < 0 **then**
12: Insert [*i*,*j*] to the list $Conflicts$
13: **end if**
14: **end for**
15: **end for**
16: maxAPFD = 0
17: *Prioritized List* = $PrioritizedList_A$
18: **for** all *i* in 0..$2^{length(Conflicts)} - 1$ **do**
19: tempList = $PrioritizedList_A$
20: bitmask = *i*
21: itr = 0
22: **while** bitmask > 0 **do**
23: **if** (bitmask%2) == 1 **then**
24: conflictingTC$_A$ = *Conflicts*[*i*][1]
25: conflictingTC$_B$ = *Conflicts*[*i*][2] //conflictingTC$_A$ and conflictingTC$_B$ represent the two conflicting test cases corresponding to *Conflicts*[*i*]
26: **for** all *i* in 1...N **do**
27: **if** tempList[*i*] == conflictingTC$_A$ **then**
28: tempList[*i*] = conflictingTC$_A$ + conflictingTC$_B$ - tempList[*i*]
29: Continue
30: **end if**
31: **if** tempList[*i*] == conflictingPair[2] **then**
32: tempList[*i*] = conflictingTC$_A$ + conflictingTC$_B$ - tempList[*i*]
33: Continue
34: **end if**
35: **end for**
36: **end if**
37: bitmask = bitmask/2
38: itr++
39: **end while**
40: //APFD() returns APFD value of a prioritized list
41: **if** APFD(tempList,FM) > maxAPFD **then**
42: maxAPFD = APFD(tempList,FM)
43: *Prioritized List* = tempList
44: **end if**
45: **end for**
46: return *Prioritized List*.

Table IV, the ranks generated by two techniques (Bayesian and EXP-TCP) are shown for five test cases. It can be seen that there are four conflicting pairs: (TC1,TC3), (TC1,TC4), (TC2,TC3), and (TC2, TC4). After finding the set of all conflicting pairs, all possible orderings are determined by first selecting a subset of conflicting pairs and then interchanging the position of test cases in each of the conflicting pair present in the subset. This is repeated for all possible subsets of all the conflicting pairs. Finally, the priority order yielding the largest APFD score is chosen.

Kendall Tau ensembling is presented in Algorithm 6. In Steps 1–15, all the conflicting pairs of test cases are determined and stored in the list *Conflicts*. In Steps 18–46, all possible orderings of test cases is generated by interchanging the ranks of each conflicting pair in every subset of the *Conflicts* list and the APFD value of each ordering is computed. Finally, the ordering that gives the largest APFD among all possible orderings is selected and stored in *Prioritized List*. The worst case time complexity of the Kendall Tau ranking algorithm is $\mathcal{O}(2^{n^2})$ [19], where n is the number of test cases. The reason behind this is that in the worst case, there can be at most O(n^2) conflicting pairs of test cases when the two input rankings are opposite to one another. To reduce the computational complexity and at the same time without unduly affecting the quality of the results, we consider a fixed number of conflicting pairs (c), where $c = \min(\text{total number of conflicting pairs}, 20)$. So, the operational time complexity of Kendall Tau algorithm ensemble is reduced to $\mathcal{O}(2^c)$ performing a maximum of $\mathcal{O}(2^{20})$ operations in the worst case.

Schulze's algorithm for ranking: The Schulze method [20] was originally used as an electoral voting system to select the winner based on the votes cast by the voters who express their preferences. This approach creates an ordered list of candidates. We illustrate this technique using the example given in Table V which shows the number of voters, that is, TCP techniques and their test case preferences. The five considered TCP techniques can show different preference orders for five test cases. A matrix representing the pairwise preferences is shown in Table VI. In

TABLE IV
ILLUSTRATION OF KENDALL TAU DISAGREEMENT

	TC1	TC2	TC3	TC4	TC5
$Bayesian-TCP$	1	2	3	4	5
$Exp-TCP$	3	4	1	2	5

TABLE V
TCP TECHNIQUES AND THEIR ORDER OF PREFERENCE

Number of TCP Techniques	Order of Preferences
2	$TC_0TC_2TC_1TC_4TC_3$
2	$TC_0TC_3TC_4TC_2TC_1$
1	$TC_1TC_4TC_3TC_0TC_2$

TABLE VI
MATRIX OF PAIRWISE PREFERENCES

	$d[*,TC_0]$	$d[*,TC_1]$	$d[*,TC_2]$	$d[*,TC_3]$	$d[*,TC_4]$
$d[TC_0,*]$		4	5	4	4
$d[TC_1,*]$	1		1	3	3
$d[TC_2,*]$	0	4		2	2
$d[TC_3,*]$	1	2	3		2
$d[TC_4,*]$	1	2	3	3	

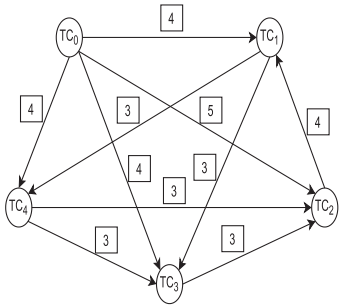

Fig. 2. Directed graph labeled with pairwise preferences.

this matrix, the value present at row *i* and column *j* indicates the number of times TC_i was preferred over TC_j. As an example, please observe that the element at the first row and second column contains the value 4. It implies that TC_0 was preferred over TC_1 by all four TCP techniques.

In the next step of the Schulze algorithm, a directed graph labeled with pairwise preferences is constructed. Fig. 2 shows a directed graph constructed by using the pairwise preferences. The nodes of the graph represent the test cases, and the weights associated with the edges indicate the pairwise preference of test cases. An arrow is drawn from a test case i to test case j when $d[\text{TC}_i, \text{TC}_j] > d[\text{TC}_j, \text{TC}_i]$ holds in the matrix of pairwise preferences. Fig. 3 shows the strongest path from TC_0 by using a dotted edge. The strongest path is one for which the lowest weight of the path is greater than lowest weights of the other paths. The directed path from TC_0 to TC_2 has the lowest weight of 5. We can also reach TC_0 to TC_2 via TC_3 as well as TC_4, both of which have the lowest weight of 3. So, the strongest path from TC_0 to TC_2 is the direct path. The lowest weight of a strongest path is also called *strength* of the strongest path. After finding the *strength* of all the strongest paths, the Schulze algorithm constructs a matrix that represents *strengths* of the strongest paths. Table VII shows the *strengths* of all the strongest paths.

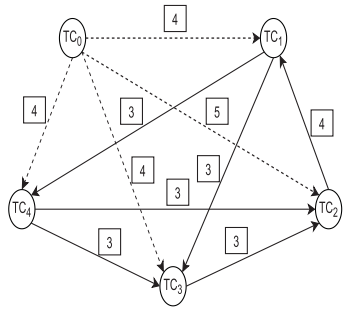

Fig. 3. Directed graph showing strongest paths from "TC_0."

TABLE VII
MATRIX TO SHOW THE STRENGTHS OF THE STRONGEST PATHS

	$d[*,TC_0]$	$d[*,TC_1]$	$d[*,TC_2]$	$d[*,TC_3]$	$d[*,TC_4]$
$d[TC_0,*]$		4	5	4	4
$d[TC_1,*]$	0		3	3	3
$d[TC_2,*]$	0	4		3	3
$d[TC_3,*]$	0	3	3		3
$d[TC_4,*]$	0	3	3	3	

In the table, $d[\text{TC}_0, \text{TC}_1] > d[\text{TC}_1, \text{TC}_0]$ implying thereby that TC_0 gets higher rank than TC_1. Similarly, TC_0 will be assigned the highest rank among all other test cases. After TC_0 is assigned rank 1, TC_0 and its corresponding edges are removed from the directed graph. Again, from the modified graph, the strongest path is determined, and the corresponding test case is assigned the next rank. This process continues until all the test cases have been assigned ranks. The final rank list for our example is: $\text{TC}_0 > \text{TC}_2 > \text{TC}_1 > \text{TC}_4 > \text{TC}_3$.

Algorithm 7 represents the Schulze method for aggregate ranking of test cases present in multiple lists. In Step 2, a set named *remainingTC* having cardinality C is created to contain all the test cases which are yet to be prioritized, C being the number of test cases. In Step 3, a 2-D matrix is created with size $C \times C$, which represents the preference of a test case by different techniques. In Step 4, another 2-D matrix of size $C \times C$ is created to store the *strengths* of the strongest paths between various test cases. Steps 5–28, compute the strengths of the strongest paths using the Floyd–Warshall algorithm and store the result in the matrix *p* of *strengths* of the strongest paths. Steps 29–41 select a winner i from the test cases set by comparing *strengths* of the strongest paths among various test cases. In Step 42, selected test case i is added to the *Prioritized List*. In Step 43, test case i is removed from the set *remainingTC*. Steps 5–43 are repeated until all test cases in *remainingTC* have been considered and added in the *Prioritized List*. Finally, in Step 45, the final *Prioritized List* is returned. The worst case time complexity of the Schulze algorithm for ranking is $\mathcal{O}(n^3)$ [20], where n is the number of test cases.

V. EXPERIMENTAL STUDIES

In this section, we first discuss our experimental setup and the subject programs that we have considered for our experimentation. Subsequently, we present the experimental results.

Algorithm 7: Schulze's Algorithm based Ensemble.

Require: Prioritized Lists from different algorithms
Create an empty list.
Create a set, named *remainingTC* containing values 1,2,...,*C*, where *C* is the total number of test cases to be ranked. // This set contains the yet to be ranked test cases.
Create a 2-D matrix of size $C \times C$ named *d* // $d[i, j]$: The number of TCP techniques that preferred test case i over test case j
Create a 2-D matrix of size $C \times C$ named p // $p[i, j]$: Strength of the strongest path from test case i to j
for all i : values in *remainingTC* **do**
  for all j : values in *remainingTC* **do**
    if $i \neq j$ **then**
      if $d[i,j] > d[j,i]$ **then**
        $p[i,j] = d[i,j]$
      end if
    else
      $p[i,j] = 0$
    end if
  end for
end for
for all i : values in *remainingTC* **do**
  for all j : values in *remainingTC* **do**
    if $i \neq j$ **then**
      for all k : values in *remainingTC* **do**
        if $i \neq k$ **then**
if $j \neq k$ **then**
$p[j,k] = \max\{\ p[j,k], \min\{\ p[j,i], p[i,k]\ \}\}$
end if
          end if
        end for
      end if
  end for
end for
Initialize a new list, winner.
for all i : values in *remainingTC* **do**
  winner[i] = **true**
end for
for all i : values in *remainingTC* **do**
  for all j : values in *remainingTC* **do**
    if $i \neq j$ **then**
      if $p[i,j] > p[j,i]$ **then**
        winner[i] = **false**
      end if
    end if
  end for
end for
Find i such that winner[i] is TRUE and push it into the *Prioritized List*.
Remove i from *remainingTC*
Repeat Steps 5–43 until *remainingTC* is empty.
return *Prioritized List*

We also analyze the results and compare them with those for existing techniques.

A. Experimental Setup

All our experiments were carried out on a 64-b Ubuntu system and a 64-b Windows 10 system with an Intel Core i5-7th Generation @ 2.70 GHz processor, with 8 GB DDR4 RAM. All the projects that we have considered in our study are Java projects which were compiled on JDK version 15. We used muJava [35] mutation system to generate mutants.

The different implementation modules of our fault-based TCP are shown in Fig. 1. For every software project considered in our study, a large number of mutants of the project were generated by automatically seeding bugs into the source code of the project. This was done using the mutator module shown in Fig. 1. This module was implemented using the muJava tool [35]. After generation of a large number of mutants, a bash script that we wrote was used to run the test suite on the mutants and the results were stored in a file. We developed a python script, *Compare.py*, to compare the output file obtained for a mutant with the output file for the original source code to generate the FM. In the next step, the obtained FM is used to prioritize the test cases using four base prioritizers. Among the base prioritizers, three (optimized BD-TCP, HDFDC, and Bayesian TCP) have been implemented in Python using Pandas and NumPy libraries. Exp-TCP has been implemented in C++ using Standard Template Library. The prioritized lists generated by different algorithms in this module are then used by the ensembling module, which generates the final prioritized lists by combining the results of the four base prioritizers. All the three ensemblers in the ensembling module were implemented in Python.

TABLE VIII
PROJECT SUMMARY

S. No.	Project Name	LOC	No. of Classes	No. of Test cases	No. of Mutants
1	Cli	4499	55	38	389
2	Closure	149516	1626	412	474
3	Codec	5834	30	43	318
4	Collections	64908	898	506	250
5	Compress	12816	92	81	390
6	Csv	1674	11	15	312
7	Gson	11560	529	261	360
8	JacksonCore	27908	190	135	376
9	JacksonDatabind	75069	2448	590	468
10	JacksonXml	7500	106	98	256
11	Jsoup	4512	171	41	430
12	JxPath	28657	397	201	330
13	Lang	61289	1298	236	364
14	Math	186609	1540	1310	460
15	Mockito	22654	1406	319	386
16	Time	68712	546	317	276

B. Subject Programs

In order to evaluate the performance of our proposed fault-based TCP techniques, we have considered 16 publicly available Java projects. All the projects were taken from Defects4j [36] GitHub repository. Defects4j is an open-source database of Java programs with realistic faults and provides an experimental infrastructure for software engineering research [36]. A few important characteristics of these java projects have been shown in Table VIII. All the considered java projects are essentially Java libraries. For example, Cli provides an interface to parse command line arguments passed to a program, and $Collections$ provides a framework to store and manipulate various Java

objects. A few other general aspects of these projects are the following.

1) The projects are relatively large with an average size of 35 357 lines of code (LOC).
2) Each project, on an average, has 709 classes.

The projects in Defects4J repository are already seeded with realistic bugs. However, to create more mutants for our experimentation, we have seeded further bugs in these projects using muJava [35] mutation system. The muJava mutation system enables the user to inject different types of mutants at method and class levels. The types of **method-level** mutants that we have seeded are presented in the following.

1) Arithmetic operator:
   a) insertion of arithmetic operator (for example, insertion of $+$ in an expression);
   b) deletion of arithmetic operator (for example, deletion of $+$ in an expression);
   c) replacement of arithmetic operator (for example, replacing $+$ operator by $-$ operator).
2) Conditional operator:
   a) insertion of conditional operator (for example, insertion of $\&\&$ in a condition);
   b) deletion of conditional operator (for example, deletion of $||$ in an expression);
   c) replacement of conditional operator (for example, replacing $||$ by $\&\&$ in a condition).
3) Logical operator:
   a) insertion of logical operator (for example, insertion of ! before a variable);
   b) deletion of logical operator (for example, deletion of ! before a variable);
   c) replacement of logical operator (for example, replacing $|$ by $\&$).
4) Relational operator:
   a) replacement of relational operator (for example, replacing $<=$ by $>=$).
5) Shift operator:
   a) replacement of shift operator (for example, replacing $<<$ by $>>$).
6) Assignment operator:
   a) replacement of assignment operator (for example, replacing $=$ by $+=$).

The types of **class-level** mutants that we have seeded are presented in the following.

1) Hiding variable:
   a) insertion of hiding variable (defining a variable in the child class with the same name of a parent class variable);
   b) deletion of hiding variable (deletion of a variable in the parent class, a variable having the same name of a child class variable).
2) Overridden method:
   a) deletion of an overridden method in the child class;
   b) renaming of an overridden method in the child class.
3) Overloading method:
   a) deletion of an overloading method;
   b) change the contents of overloading method.
4) Variable declaration:
   a) declare the instance variable with parent class type;
   b) declare the parameter variable with child class type.
5) *super* keyword:
   a) insertion and deletion of *super* keyword.
6) *this* keyword:
   a) insertion and deletion of *this* keyword.
7) *static* modifier:
   a) insertion and deletion of *static* modifier.
8) Type cast operator:
   a) insertion and deletion of type cast operator;
   b) change of cast type.

We have seeded the subject programs with mutants corresponding to various simple programming bugs using muJava [35]. We have selected the mutation operators as well as the frequency of their occurrence to roughly match the programming errors typically committed by programmers and the frequency with which they commit these [37]. This is based on a study that we have carried out to determine the important types of bugs that have been reported in four open-source software projects, and the frequency with which these bugs occur [38].

The test suite for each project is available in the Defects4J repository [36]. The number of mutants that we have used and the number of test cases in the test suite for each project are shown in Table VIII.

TABLE IX
APFD SCORES OF DIFFERENT FAULT-BASED TCP TECHNIQUES

S. No.	Programs	Optimized BD-TCP	HDFDC	Exp-TCP	Bayesian	Averaging	Kendall Tau	Schulze
1	Cli	0.981	0.959	0.938	0.970	0.950	0.969	0.975
2	Closure	0.979	0.953	0.937	0.946	0.954	0.970	0.968
3	Codec	0.979	0.97	0.866	0.967	0.957	0.996	0.98
4	Collections	0.956	0.973	0.877	0.932	0.936	0.972	0.958
5	Compress	0.931	0.92	0.853	0.924	0.986	0.930	0.935
6	CSV	0.938	0.930	0.865	0.920	0.894	0.944	0.949
7	Gson	0.96	0.918	0.82	0.915	0.851	0.961	0.963
8	JacksonCore	0.952	0.919	0.910	0.888	0.941	0.940	0.938
9	JacksonDatabind	0.989	0.971	0.934	0.979	0.962	0.985	0.976
10	JacksonXml	0.941	0.94	0.881	0.936	0.913	0.957	0.943
11	Jsoup	0.920	0.919	0.786	0.914	0.910	0.935	0.934
12	JxPath	0.924	0.889	0.888	0.877	0.954	0.962	0.965
13	Lang	0.963	0.958	0.922	0.948	0.932	0.963	0.957
14	Math	0.990	0.970	0.917	0.964	0.949	0.992	0.990
15	Mockito	0.935	0.910	0.882	0.924	0.917	0.933	0.950
16	Time	0.893	0.888	0.80	0.873	0.844	0.882	0.893

Table IX presents the APFD scores of our seven TCP techniques for the 16 Java projects. Among the seven TCP techniques, the first four are base prioritizers, and the last three are those of the three ensemblers. Please observe from Table IX that among the base prioritizers, optimized BD-TCP has achieved the highest APFD score (0.990) for $Math$ projects. On the other hand, the Exp-TCP showed the lowest APFD (0.786) for the $Jsoup$ project. Among our three ensemblers, the Kendall Tau ensembler achieved the highest APFD score (0.996) for the $Codec$ project which is also the highest among all the techniques. Averaging ensembler showed the lowest APFD (0.844) for the $Time$ project. It can be deduced from Table IX that the Kendall Tau ensembler achieved the highest average APFD (0.955) for all the projects among all the techniques. A possible reason for this is that the Kendall Tau ensembler calculates the APFD value of

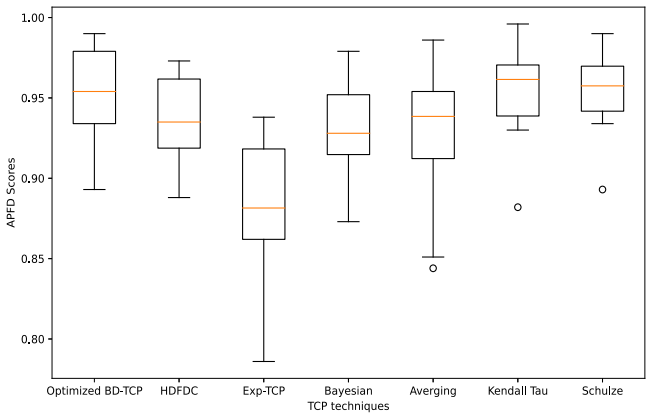

Fig. 4. Box plot representation of different fault-based TCP techniques.

TABLE X
EXECUTION TIME (IN SECONDS) OF DIFFERENT TECHNIQUES OF FAULT-BASED TCP

S. No.	Programs	Optimized BD-TCP (Recurring)	HDFDC (Recurring)	Exp-TCP (Recurring)	Bayesian (Recurring)	Averaging (Recurring)	Kendall Tau (Recurring)	Schulze (Recurring)
1	Cli	0.519	0.364	0.353	0.349	0.214	30.007	2.251
2	Closure	1.966	2.437	0.594	0.429	0.359	34.459	2.713
3	Codec	0.458	0.234	0.561	0.309	0.225	18.594	2.790
4	Collections	1.951	2.324	0.572	0.941	0.252	36.470	3.835
5	Compress	0.408	0.293	0.974	0.408	0.249	49.857	2.084
6	CSV	0.344	0.203	0.341	0.226	0.214	10.107	1.666
7	Gson	0.588	0.303	0.846	0.532	0.237	42.616	2.206
8	JacksonCore	0.524	1.595	0.408	0.651	0.229	46.196	3.110
9	JacksonDatabind	2.960	2.545	3.931	3.399	0.292	49.020	5.143
10	JacksonXml	0.503	0.239	1.112	0.422	0.217	36.864	2.005
11	Jsoup	0.461	0.219	0.827	0.200	0.212	29.067	0.908
12	JxPath	0.607	0.723	2.919	1.016	0.232	37.938	3.675
13	Lang	0.769	1.227	0.436	2.270	0.259	27.686	2.428
14	Math	3.805	3.368	1.943	2.442	0.364	52.677	5.591
15	Mockito	1.874	0.366	0.879	1.152	0.268	29.425	4.182
16	Time	0.607	0.445	1.473	1.793	0.261	36.869	4.414

all possible orderings of conflicting pairs and selects the one that achieves the highest APFD score. During our experimentation, we found that a reasonable number of conflicting pairs to be considered for Kendall Tau ensembling algorithm is 20. With this, the algorithm was time efficient even for projects with large number of test cases and at the same time did not unduly compromise the quality of the results. Among our four base prioritizers, optimized BD-TCP showed the highest average APFD (0.951), and Exp-TCP showed the lowest average APFD (0.879). An arrangement of our TCP techniques in increasing order of average APFD is: Exp-TCP (0.879) $<$ Averaging (0.928) $<$ Bayesian (0.929) $<$ HDFDC (0.936) $<$ Optimized BD-TCP (0.951) $<$ Schulze (0.954) $<$ Kendall Tau (0.955).

It can also be deduced from Table IX the average APFD achieved by our seven TCP techniques considering all the projects. It can be calculated from Table IX that our fault-based TCP techniques achieved the highest average APFD (0.970) for $JacksonDatabind$ project and the lowest average APFD (0.867) for $Time$ project. $JacksonDatabind$ has the highest number of classes (2448) among all the projects, which can be observed from Table VIII. Our techniques showed the second-highest total average APFD (0.967) for $Math$ project. In Table VIII, it can be seen that $Math$ has the highest number of LOC (186 609) and highest number of test cases (1310). It indicates that our techniques can be applied even when the size of the test suite is large.

Fig. 4 depicts the box plot representations of the performance of our seven TCP techniques. The *X*-axis of the box plot represents our seven TCP techniques, and *Y*-axis represents the corresponding APFD scores. It can be observed from Fig. 4 that the Exp-TCP technique showed the lowest median and the lowest maximum value among all the techniques. Among the four base prioritizers, optimized BD-TCP showed highest median and highest maximum values. HDFDC and Bayesian techniques showed very similar median, maximum values, as well as interquartile range (IQR). It means that both the techniques perform very similarly for most of the projects. Among our three ensemblers, averaging ranking showed lowest median and lowest maximum values compared to the other two ensemblers. The Kendall Tau ensembler achieved the highest median and the highest maximum value among all the TCP techniques. This implies that for most of the projects, the Kendall Tau ensembler outperformed other techniques. On the other hand, Schulze ensembler showed the highest minimum value and the shortest IQR among all the techniques, implying that Schulze ensembler performs most consistently among all the prioritizers.

C. Time-Cost Analysis of our Fault-Based TCP Techniques

Time overhead for regression TCP consists of the following two main components:

1) one-time cost;
2) recurring cost incurred after each maintenance cycle.

In our fault-based regression TCP approach (shown in Fig. 1), the FM generator module is run initially only once for a given software, and its execution is a one-time cost. The one-time cost is amortized over a large number of regression testing cycles. Therefore, in a practical scenario, this amortized one-time cost is negligible for any specific maintenance cycle. Recurring cost is incurred due to execution of the base prioritizers (optimized BD-TCP, HDFDC, Exp-TCP, and Bayesian), and execution of the ensemblers (averaging, Kendall Tau, and Schulze). In our approach, when the prioritized list of test cases is executed during each maintenance cycle, the FM gets updated with the new fault information without incurring any significant overhead.

We have conducted experiments to determine the execution times of each of our base prioritizers (optimized BD-TCP, HDFDC, Exp-TCP, and Bayesian), and the ensemblers (averaging, Kendall Tau, and Schulze). The results have been tabulated in Table X, in which all times have been shown in seconds. It can be observed that the average execution times of our fault-based TCP techniques (base prioritizers and the ensemblers) is 43.177 s. It can also be seen in Table X that among our four base prioritizers, Bayesian TCP technique incurs the minimum amount of average execution time (1.033 s) and optimized BD-TCP incurs the maximum amount of average execution time (1.146 s). Among our three ensemblers, Kendall Tau incurs the largest average execution time (35.491 s) because as presented in the theoretical analysis part (in Section IV-B), a

Kendall Tau ensembler always tries to find a combination of test cases among all possible combinations to maximize the fault detection rate for a test suite. Schulze ensembler takes the second-lowest amount of average execution time (3.062 s). Averaging ensembler incurs the minimum amount of average execution time (0.255 s) as it simply averages the ranks produced by the four base prioritizers. Overall, it can be seen that the total recurring TCP cost does not even exceed 1 min of execution time for any of our TCP techniques for the programs considered in our experimental study.

Now let us analyze the time savings that can potentially accrue due to appropriate TCP. The time to carry out regression testing consists of several components [39], [40]. These include test setup time, test data input time, test execution time, analysis of test results, and preparation of test report. The time required for executing a test case is usually of the order of a few seconds. Test case execution time is insignificant compared to the times for other regression testing activities, as those involve manual activities. Usually, it takes a few minutes to a few hours to set up the testing environment for a group of test cases depending upon the characteristics of the system and the specific group of test cases. In manual testing, it usually takes time of the order of several tens of seconds to provide the test inputs for executing a test case and it may take several minutes to analyze the results. It typically takes several minutes to incorporate the results of a test case execution in the test report. It is true that the times taken for various testing activities do depend to a large extent on the proficiency of the tester. However, it is reasonable to assume that, on an average, each test case execution takes several minutes. If the regression test suite for a software comprises a few hundreds of test cases, then regression testing after a maintenance cycle can take up to several man-days. As can be seen, this is considerably higher than the time taken for TCP. A proper TCP can provide the developers hours of lead time in fixing the bugs. Further, when only a subset of regression test cases are to be run due to time constraints, proper TCP can reduce a regression testing cycle by several hours. Over the entire maintenance phase (all regression cycles considered together), the savings achieved due to proper regression TCP can be significant.

D. Comparison With Related Work

We compare our work with statement coverage (SC), branch coverage (BC), TCP-ANT, and a diversity-aware mutation-based TCP technique named HYBrid-weight. SC and BC are two popular TCP techniques [9]. Vescan et al. [23] proposed an ant colony optimization-based TCP-ANT technique and named it as *test case prioritization*-ANT. TCP-ANT considers fault severity of the yet-undetected faults as well as test case execution times to prioritize the test cases. They determined the fault severity score of a fault based on the function criticality. A fault present in a critical function is given a higher fault severity score. Shin et al. [24] proposed a set of diversity-aware mutation-based TCP techniques. Since their HYB-w TCP technique is also mutation based, we compare the performance of our TCP techniques with this technique. HYB-w accords higher priority to a test case that increases the weighted sum of the additional distinguished mutants and additional killed mutants. It may be noted that the weight (w) can vary from 0 to 1. In our study, we have taken "w" to be 0.25 as for different w values with which we experimented, HYB-w showed superior results when "w" was set to 0.25.

As we have already discussed, Shin et al. [24] reported a work that is closely related to ours in that they also used a mutation-based TCP technique. However, there are many important differences between their work and our fault-based TCP techniques. Shin et al. proposed their TCP technique by combining the traditional mutation-based [9] and mutant distinguishment-based [41] prioritization techniques. The traditional mutation-based TCP approach is based on the capability of a regression test case to kill mutants. Test cases are essentially prioritized based on the number of additional mutants killed by a test case that have so far remained alive. Mutant distinguishment-based prioritization technique is based on the capability of a regression test case to distinguish the behavior of one mutant from another. A test case is assigned a priority value based on the number of additional mutants it can distinguish. They combined these two approaches to propose their HYB-w TCP technique. On the other hand, we have proposed four fault-based base prioritizers and three techniques for ensembling the results produced by the four base prioritizers. Our first base prioritizer, optimized BD-TCP, prioritizes test cases based on the total number of seeded bugs detected by a test case. That is, a test case gets higher priority if it detects a higher number of seeded bugs. At the first glance, traditional mutation-based TCP and our optimized BD-TCP may look similar. However, one important difference between the two techniques is that though both the techniques prioritize test cases based on the total number of additional bugs detected, but when there is a tie between a set of test cases that detect equal number of bugs, the technique of Shin et al. randomly assigns higher priority to any of these test cases, whereas our optimized BD-TCP technique assigns higher priority to the test case that detects a greater number of scarcely detected bugs. Our second base prioritization approach, HDFDC, prioritizes the test cases based on a test case's capability to detect bugs that are hard to detect. Our third base prioritizer, Exp-TCP, is based on the expected value of bug exposing potential of the test case. Our fourth base prioritizer, Bayesian TCP, is based on first computing the probability of a test case to detect a bug. Based on the computed probability value, it prioritizes the test cases. Subsequently, the results of the four base prioritizers are ensembled using three ensemble methods to generate three different TCP lists. Further, the diversity-aware mutation-based TCP technique of Shin et al. mainly focuses on the capability of test cases to distinguish the behavior of mutants. On the other hand, our four fault-based base prioritizers are based on a direct measure of the capability of the test cases to expose bugs.

We have experimentally evaluated the performance of the HYB-w TCP technique of Shin et al. [24] with our four fault-based base prioritizers. With the help of Tables IX and XI, we can compare the APFD values achieved by HYB-0.25 TCP with our four base prioritizers. From Tables IX and XI, it can be observed that HYB-0.25 attained the lowest APFD among all

TABLE XI
APFD SCORES OF EXISTING TCP TECHNIQUES AND THREE ENSEMBLE METHODS

S. No.	Programs	SC	BC	TCP-ANT	HYB-0.25	Averaging	Kendall Tau	Schulze
1	Cli	0.774	0.774	0.842	0.850	0.950	0.969	0.975
2	Closure	0.514	0.514	0.828	0.968	0.954	0.970	0.968
3	Codec	0.567	0.639	0.912	0.890	0.957	0.996	0.980
4	Collections	0.574	0.574	0.972	0.944	0.936	0.972	0.958
5	Compress	0.603	0.603	0.859	0.893	0.986	0.930	0.935
6	CSV	0.529	0.536	0.917	0.816	0.894	0.944	0.949
7	Gson	0.547	0.547	0.898	0.942	0.851	0.961	0.963
8	JacksonCore	0.577	0.578	0.875	0.936	0.941	0.940	0.938
9	JacksonDatabind	0.581	0.620	0.930	0.935	0.962	0.985	0.976
10	JacksonXml	0.702	0.702	0.974	0.946	0.913	0.957	0.943
11	Jsoup	0.682	0.682	0.849	0.875	0.910	0.935	0.934
12	JxPath	0.531	0.834	0.860	0.939	0.954	0.962	0.965
13	Lang	0.617	0.929	0.846	0.859	0.932	0.963	0.957
14	Math	0.539	0.779	0.750	0.810	0.949	0.992	0.990
15	Mockito	0.662	0.662	0.871	0.933	0.917	0.933	0.950
16	Time	0.550	0.550	0.856	0.925	0.844	0.882	0.893

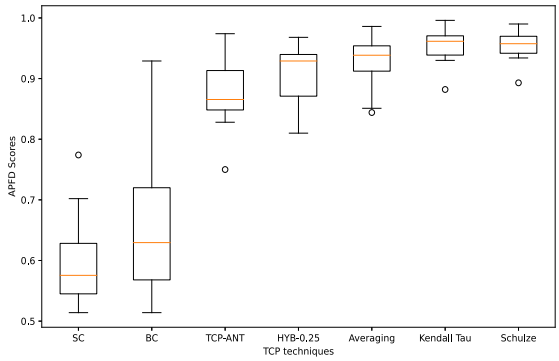

Fig. 5. Box plot representation of the performance of existing TCP techniques and ensemble methods.

the five techniques for Cli, CSV, $Lang$, and $Math$ projects. This can be interpreted to mean that for these projects, test cases are not able to distinguish the behavior of many mutants. For a given project, HYB-w TCP can show improved performance when the test cases are able to distinguish the behavior of a large number of mutants. In line with this observation, HYB-0.25 achieves the highest APFD score among all the five techniques for $JacksonXml$, $JxPath$, and $Time$ projects. It can be observed from Figs. 4 and 5 that among our four base prioritizers, three prioritizers (optimized BD-TCP, HDFDC, and Bayesian) exhibited higher median and larger APFD value than HYB-0.25 and all of our four base prioritizers showed lower IQR than HYB-0.25. Therefore, for most of the considered projects, our fault-based TCP exhibited more consistent performance as compared to the diversity-aware mutation-based TCP technique (HYB-0.25).

Now, we compare the performance of our three ensemblers with four existing TCP techniques. Table XI shows APFD scores obtained by the four existing TCP techniques and our three ensemblers. We have compared the performance of existing techniques with only our three ensemblers and have not considered our base TCP techniques as ensemble techniques yield results that are superior to our four fault-based base prioritizers. It can be observed from Table XI, BC prioritization shows similar or better APFD value than SC prioritization for all

TABLE XII
AVERAGE RELATIVE IMPROVEMENT (ARI) OVER EXISTING TECHNIQUES BY OUR THREE ENSEMBLE METHODS

S. No.	Projects	ARI of Averaging Ensembler	ARI of Kendall Tau Ensembler	ARI of Schulze Ensembler
1	Cli	17.51%	19.86%	20.60%
2	Closure	46.24%	48.69%	48.39%
3	Codec	32.75%	38.16%	35.94%
4	Collections	30.39%	35.41%	33.46%
5	Compress	38.05%	30.21%	30.91%
6	CSV	35.70%	43.29%	44.05%
7	Gson	24.06%	40.10 %	40.39%
8	JacksonCore	33.49%	33.34%	33.06%
9	JacksonDatabind	31.76%	34.91%	33.68%
10	JacksonXml	12.59%	18.01%	16.29%
11	Jsoup	19.51%	22.79%	22.66%
12	JxPath	26.64%	27.70%	28.10%
13	Lang	17.51%	21.41%	20.66 %
14	Math	35.39%	41.53%	41.24 %
15	Mockito	20.15%	22.24%	24.47 %
16	Time	24.18%	29.77%	31.39%

the projects. A possible reason for this is that BC covers all the branches that might get missed in the SC approach. It can also be observed from Table XI that HYB-0.25 and TCP-ANT showed very similar results. However, for most cases, HYB-0.25 achieved marginally better APFD score than TCP-ANT. All the three ensemble methods outperformed existing techniques in all considered projects barring three projects ($Collections$, $JacksonXml$, and $Time$). In these three projects, only HYB-0.25 and TCP-ANT showed APFD values comparable to our ensemble methods.

Fig. 5 presents the box-plot representation of the performance of existing TCP techniques along with that of our proposed ensemble TCP methods. It can be observed from Fig. 5 that among the four existing techniques, TCP-ANT achieved the largest maximum APFD value, but HYB-0.25 achieved the largest median APFD value. On the other hand, SC showed lowest median and BC showed largest IQR. It can be observed from Fig. 5 that all our ensemblers showed higher median, higher maximum, and shorter IQR than existing TCP techniques. This implies that our ensemblers performed better for most cases and achieved more consistent results for all the projects. Among all the TCP techniques, Kendall Tau achieved highest median and Schulze ensembler achieved shortest IQR. Overall, the Kendall Tau ensembler showed superior performance for most of the projects, and the Schulze ensembler showed better performance consistency in terms of the rate of fault detection as compared to all considered techniques.

Table XII represents the average relative improvement achieved by our three ensemblers (averaging, Kendall Tau, and Schulze) over existing techniques. It can be observed from Table XII that all the three ensemble methods showed the highest improvement for the $Closure$ project and the third highest improvement for the $Math$ project over existing techniques. $Closure$ and $Math$ projects have the highest number (474) and the third highest number (460) of seeded mutants among all the projects considered (can be seen in Table VIII). We hypothesize that when there are large number of mutants, all

our ensemble methods show significant improvement in terms of fault detection rate over the existing techniques. On the other hand, all the three ensemblers achieved lowest average relative improvement over existing techniques for the $JacksonXml$ project. It can be observed from the project summary provided in Table VIII, $JacksonXml$ has a significantly lower LOC count, lower number of seeded mutants, and lower number of classes than the other projects. Possibly because of this, all the existing TCP techniques achieved similar APFD scores like as the ensemblers for $JacksonXml$ compared to other projects. Focusing on the individual performance of our ensemble methods, it can be observed from Table XII that averaging, Schulze, and Kendall Tau ensemblers showed 27.87, 31.58, and 31.71% average APFD score improvements respectively over existing TCP techniques for the considered projects. On an average, our proposed ensemble methods achieved at least 30.38% improvement in APFD score over existing TCP techniques.

VI. Threats to Validity

In this section, we discuss an important threat to the validity of our results and also outline how we have addressed this issue. While generating mutants automatically using muJava [35], it is possible that equivalent mutants can get generated. An equivalent mutant is not really an error in the program but is a change introduced into the program (such as replacing one variable by another) for which no test cases can fail. The equivalent mutant problem (EMP) is one of the crucial problems in mutation testing and has been widely studied over last several decades. It is well accepted that automatically detecting equivalent mutants is an undecidable problem [42], [43]. Unless the EMP his handled satisfactorily, the results of a mutation-based TCP can be erroneous. In this light, to handle equivalent mutants that may get generated during our experimentation, we decided to exclude any mutant for which no test case fails. We feel that this is reasonable, since the test suite for all the subject programs are well designed, a mutant for which no test case fails can be taken to be an equivalent mutant and excluded from further considerations.

VII. Conclusion

We first proposed four novel fault-based prioritizers to generate an initial rank lists of regression test cases. These four base prioritizers use four different fault-based methodologies to rank the test cases. We chose three ranking aggregation techniques that aggregate the ranks generated by the four base prioritizers and generate three prioritization lists of the test cases. We experimentally evaluated the performance of our ensemble methods as well as four popular existing TCP techniques on 16 Java projects. Our experimental results showed that averaging, Kendall Tau, and Schulze ensemble methods altogether achieve 60.57%, 47.71%, 8.27%, and 5.16% average relative improvements respectively over SC-based, BC-based, TCP-ANT, and diversity-aware mutation-based HYB-w TCP techniques. The Kendall Tau ensemble technique showed the highest improvement over all the TCP techniques considered in our experimental studies. We observed that the Kendall Tau ensemble technique outperformed all existing TCP techniques even while restricting the conflicting pairs to only 20. The ensemble method achieved on an average 30.38% APFD score improvement over existing techniques. This improvement may be because our TCP techniques consider additional factors such as the total number of bugs detected by a test case, the difficulty level of detecting different types of bugs, and the expected value of failure of a test case due to a bug.

In our article, we did not consider interclass mutation operators. In future, we plan to consider these and also we plan to conduct experiments on larger programs.

References

[1] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, no. 2, pp. 173–210, Apr. 1997.

[2] H. K. N. Leung and L. White, "Insights into regression testing (software testing)," in *Proc. Conf. Softw. Maintenance*, 1989, pp. 60–69.

[3] M. J. Harrold et al., "Regression test selection for java software," *ACM Sigplan Notices*, vol. 36, no. 11, pp. 312–326, Oct. 2001.

[4] R. Gregg, H. Mary Jean, and Dedhia, "Regression test selection for C++ software," Oregon State Univ., Corvallis, OR, USA, Tech. Rep. 99-60-01, 1999.

[5] Rothermel and Harrold, "Selecting regression tests for object-oriented software," in *Proc. Int. Conf. Softw. Maintenance*, 1994, pp. 14–25.

[6] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," in *Proc. Conf. Softw. Maintenance*, 1990, pp. 302–310.

[7] B. Korel, L. H. Tahat, and B. Vaysburg, "Model based regression test reduction using dependence analysis," in *Proc. Int. Conf. Softw. Maintenance*, 2002, pp. 214–223.

[8] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 159–182, Feb. 2002.

[9] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, Oct. 2001.

[10] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, "Selecting a cost-effective test case prioritization technique," *Softw. Qual. J.*, vol. 12, no. 3, pp. 185–210, Sep. 2004.

[11] A. G. Malishevsky, J. R. Ruthruff, G. Rothermel, and S. Elbaum, "Cost-cognizant test case prioritization," Dept. Comput. Sci. Eng., Univ. Nebraska, Lincoln, Tech. Rep. TR-UNL-CSE-2006-0004, 2006.

[12] H. Srikanth, L. Williams, and J. Osborne, "System test case prioritization of new and regression test cases," in *Proc. Int. Symp. Empirical Softw. Eng.*, 2005, pp. 1–10.

[13] R. Krishnamoorthi and S. A. Sahaaya Arul Mary, "Factor oriented requirement coverage based system test case prioritization of new and regression test cases," *Inf. Softw. Technol.*, vol. 51, no. 4, pp. 799–808, Apr. 2009.

[14] B. Korel, L. H. Tahat, and M. Harman, "Test prioritization using system models," in *Proc. 21st IEEE Int. Conf. Softw. Maintenance*, 2005, pp. 559–568.

[15] C. R. Panigrahi and R. Mall, "Model-based regression test case prioritization," in *Information Systems, Technology and Management*, S. K. Prasad, H. M. Vin, S. Sahni, M. P. Jaiswal, and B. Thipakorn, Eds. Berlin, Heidelberg, Germany: Springer, 2010, pp. 380–385, doi: 10.1007/978-3-642-12035-0_39.

[16] C. R. Panigrahi and R. Mall, "An approach to prioritize the regression test cases of object-oriented programs," *CSI Trans. ICT*, vol. 1, no. 2, pp. 159–173, Jun. 2013. [Online]. Available: https://doi.org/10.1007/s40012-013-0011-7

[17] S.-e.-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.

[18] 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.

[19] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar, "Rank aggregation methods for the web," in *Proc. 10th Int. Conf. World Wide Web*, New York, NY, USA, 2001, pp. 613–622.
[20] M. Schulze, "The schulze method of voting," Mar. 15, 2020. [Online]. Available: http://arxiv.org/abs/1804.02973
[21] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proc. 8th Int. Symp. Softw. Rel. Eng.*, 1997, pp. 264–274.
[22] L. Larsen and M. J. Harrold, "Slicing object-oriented software," in *Proc. 18th Int. Conf. Softw. Eng.*, Washington, DC, USA, 1996, pp. 495–505.
[23] A. Vescan, C.-M. Pintea, and P. Pop, "Test case prioritization–ANT algorithm with faults severity," *Log. J. IGPL*, vol. 30, no. 2, pp. 277–288, 2020.
[24] D. Shin, S. Yoo, M. Papadakis, and D.-H. Bae, "Empirical evaluation of mutation-based test prioritization techniques," *Softw. Testing, Verification Rel.*, vol. 29, 2017, Art. no. e1695.
[25] Z. Q. Zhou, C. Liu, T. Y. Chen, T. H. Tse, and W. Susilo, "Beating random test case prioritization," *IEEE Trans. Rel.*, vol. 70, no. 2, pp. 654–675, Jun. 2021.
[26] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The art of test case diversity," *J. Syst. Softw.*, vol. 83, no. 1, pp. 60–66, Jan. 2010. [Online]. Available: https://doi.org/10.1016/j.jss.2009.02.022
[27] R. Krishnamoorthi and S. S. A. Mary, "Regression test suite prioritization using genetic algorithms," *Int. J. Hybrid Inf. Technol.*, vol. 2, no. 3, pp. 35–52, 2009.
[28] N. Gökçe and M. Eminli, "Model-based test case prioritization using neural network classification," *Comput. Sci. Eng.: Int. J.*, vol. 4, pp. 15–25, 2014.
[29] J. Chen et al., "Test case prioritization for object-oriented software: An adaptive random sequence approach based on clustering," *J. Syst. Softw.*, vol. 135, pp. 107–125, Jan. 2017.
[30] S. Mirarab and L. Tahvildari, "A prioritization approach for software test cases based on Bayesian networks," in *Fundamental Approaches to Software Engineering*, M. B. Dwyer and A. Lopes, Eds. Berlin, Heidelberg, Germany: Springer, 2007, pp. 276–290.
[31] X. Zhao, Z. Wang, X. Fan, and Z. Wang, "A clustering-Bayesian network based approach for test case prioritization," in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, 2015, vol. 3, pp. 542–547.
[32] S. Biswas and A. Bansal, "A regression test case prioritization technique using ensemble learning," Dept. Comput. Sci. Eng., Indian Inst. Technol., Kharagpur, WB, India, Tech. Rep., Aug. 2021.
[33] S. Biswas, R. Rathi, A. Dutta, P. Mitra, and R. Mall, "A regression test case prioritization technique targeting 'hard to detect' faults," *Int. J. Syst. Assurance Eng. Manage.*, vol. 13, pp. 1066–1081, 2022.
[34] B. Michael et al., "Linearity of expectation." Apr. 2021. [Online]. Available: https://brilliant.org/wiki/linearity-of-expectation/
[35] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "MuJava: An automated class mutation system: Research articles," *Softw. Testing Verification Rel.*, vol. 15, no. 2, pp. 97–133, Jun. 2005.
[36] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Testing Anal.*, San Jose, CA, USA, 2014, pp. 437–440.
[37] V. Vipindeep and P. Jalote, "List of common bugs and programming practices to avoid them," Dept. Comput. Sci. Eng., IIT Kanpur, WB, India, Tech. Rep., 2005.
[38] J. P. Meher, S. Biswas, and R. Mall, "Bug distribution analysis on open source repository," Dept. Comput. Sci. Eng., Indian Inst. Technol., Kharagpur, WB, India, Tech. Rep., Sep. 2021.
[39] R. Mall, *Fundamentals of Software Engineering*, 4th ed. India: Prentice-Hall of India Pvt. Ltd, 2014.
[40] A. Ngah, M. Munro, and M. Abdallah, "An overview of regression testing," *J. Telecommun., Electron. Comput. Eng.*, vol. 9, pp. 45–49, 2017.
[41] D. Shin, S. Yoo, and D.-H. Bae, "Diversity-aware mutation adequacy criterion for improving fault detection capability," in *Proc. IEEE N9th Int. Conf. Softw. Testing, Verification Validation Workshops*, 2016, pp. 122–131.
[42] M. Papadakis, M. Delamaro, and Y. Le Traon, "Mitigating the effects of equivalent mutants with mutant classification strategies," *Sci. Comput. Program.*, vol. 95, pp. 298–319, 2014.
[43] M. Baer, N. Oster, and M. Philippsen, "Mutantdistiller: Using symbolic execution for automatic detection of equivalent mutants and generation of mutant killing tests," in *Proc. IEEE Int. Conf. Softw. Testing, Verification Validation Workshops*, 2020, pp. 294–303.