Int J Syst Assur Eng Manag (June 2022) 13(3):1066–1081
https://doi.org/10.1007/s13198-021-01385-4

ORIGINAL ARTICLE

# A regression test case prioritization technique targeting 'hard to detect' faults

**Sourav Biswas**[1] · **Raghav Rathi**[2] · **Arpita Dutta**[1] · **Pabitra Mitra**[1] · **Rajib Mall**[1]

Received: 4 April 2021 / Revised: 14 May 2021 / Accepted: 8 September 2021 / Published online: 18 September 2021


**Abstract** We propose a novel regression test case prioritization technique targeting to detect 'hard to detect' faults. A 'hard to detect fault' is a fault that is detected by only one test case. In our technique, we first seed a large number of bugs into a program to create mutants. Each mutant is executed with the test suite and their execution results are recorded in a fault matrix. Using the fault matrix, we first prioritize the test cases based on their 'hard to detect' values. Our technique assigns higher priority to those test cases which reveal bugs in a program-component that are hard to expose. Subsequently, the remaining test cases are prioritized based on their fault detection capability. Our experimental results show that on an average our proposed TCP technique performs 43.82% better than existing TCP techniques.

**Keywords** Regression testing · Test case prioritization · Mutation testing

✉ Sourav Biswas
bsws.sourav@gmail.com

Raghav Rathi
rathi.raghav00@gmail.com

Arpita Dutta
arpitad10j@gmail.com

Pabitra Mitra
pabitra@cse.iitkgp.ac.in

Rajib Mall
rajib@cse.iitkgp.ac.in

[1] Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur, India

[2] Department of Computer Science and Engineering, Delhi Technological University, Delhi, India

## 1 Introduction

Regression testing provides confidence that a set of changes made to a program have not degraded the behavior of the unchanged functionalities of the program Mall (2018). This can simplistically be ensured by rerunning all the available test cases after each change. However, rerunning all test cases after every change to a program is time-consuming and expensive. For this reason, suitable regression test selection (RTS) techniques (Rothermel 1994; Harrold et al. 2001; Gregg et al. 1999) are deployed. RTS techniques select a subset of the test cases to run after a change. Test case prioritization (TCP) techniques (Elbaum et al. 2002; Rothermel et al. 2001; Elbaum et al. 2004; Malishevsky et al. 2006) have been proposed to prioritize the regression test cases. TCP techniques help when the tester is unable to run the full regression test suite due to time or cost considerations. Also, by executing the regression test suite in order of their priority assigned by a TCP technique, the rate of fault detection improves, thereby providing more time to the developers for bug correction.

TCP techniques rank the test cases of a test suite in a priority order. For ranking the test cases, TCP techniques optimize specific criteria such as the maximization of the rate of bug[1] detection, and maximization of the rate of increase in code coverage. In other words, a TCP technique assigns a priority score to each test case based on the number of yet undetected faults detected by a test case or the extent of incremental code coverage achieved. Based on the assigned score, the test cases are ordered in a

[1] In this paper, we have used the terms 'bug' and 'fault' interchangeably.

prioritized list. A tester can choose the top most test cases from the prioritized list on time and cost considerations.

In the literature, several TCP techniques have been proposed by different researchers. Most of the techniques are based on coverage of code Rothermel et al. (2001); Elbaum et al. (2002), requirements Srikanth et al. (2005); Krishnamoorthi and Sahaaya Arul Mary (2009), or various types of model elements Korel et al. (2005); Panigrahi and Mall (2010, 2013). However, test case prioritization techniques based on the capability of test cases to expose 'hard to detect' bugs has not been reported.

In this context, we proposed a novel test case prioritization technique based on the potential of the test cases to expose 'hard to detect' bugs. Our technique prioritizes the test cases in three different groups. First group contains the list of prioritized test cases that detect bugs not detected by other test cases, thereby these test cases are indispensable. The second group contains a priority list of test cases which detect bugs that are also detected by other test cases. Third group contains the prioritized order of the redundant test cases. A higher priority is accorded to a test case that detects bugs detected by a very few test cases.

Rest of the paper is organized as follows. In Sect. 2, we discuss related work. Our proposed TCP technique is presented in Sect. 3. In Sect. 4, we discuss the experimental results. Finally, Sect. 5 concludes this paper.

## 2 Related work

A large number of TCP techniques have been proposed by researchers. These TCP techniques can broadly be classified into: Coverage-based Rothermel et al. (2001); Elbaum et al. (2002), requirement-based Srikanth et al. (2005); Krishnamoorthi and Sahaaya Arul Mary (2009), model-based Korel et al. (2005); Panigrahi and Mall (2010, 2013) techniques. We briefly review these techniques in the following.

Rothermel et al. (2001, 1999) have proposed a number of TCP techniques. They proposed a statement coverage-based approach (Panda et al. 2019; Lou et al. 2015) that orders test cases according to the number of statement exercised. Their branch coverage-based technique (Rothermel et al. 2001, 1999) rank the test cases on the basis of total number of branches exercised by each test case. They also proposed TCP techniques based on fault-exposing-potential (FEP) (Rothermel et al. 2001, 1999) of the test cases. To prioritize the test cases based FEP, they calculated mutation-score for each test case. Mutation score is a ratio of mutants detected by a test case $t$ to the total number of mutants. They used APFD (Rothermel et al. 2001) metric to determine the performance of their proposed techniques. Later, Elbaum et al. (2002) extended the work of Rothermel et al. (2001) by proposing a function coverage approach. In their technique, test cases are prioritized according to the number of functions exercised by each test case. Their experimental studies showed that their proposed technique helps in early fault detection. Jones and Harrold (2003) proposed effective test suite reduction and prioritization technique using a modified condition/decision coverage (MC/DC) based approach (Dutta et al. 2019).

Srikanth et al. (2005) proposed a system-level requirement-based TCP technique. They used factors such as the number of customer requirements, number of changes to requirements, difficulty to implement requirements, and fault susceptibility of requirements. Krishnamoorthi and Sahaaya Arul Mary (2009) proposed a software requirement specification (SRS) document-based TCP prioritization technique. Their technique prioritized test cases using six factors: customer requirements, difficulty in implementing requirements, changes in requirements, faults in requirements, completeness of requirements, and traceability of requirements.

Korel et al. (2005) presented a state model-based TCP technique. They constructed two different Extended Finite State Machine (EFSM) models: one for the original software, and another for the modified software. These two models were compared to identify any differences in transitions. Based on the differences in transitions, test cases were prioritized. Panigrahi and Mall (2010, 2013, 2013) proposed a model-based TCP technique for object-oriented programs. An extended system dependency graph (ESDG) Larsen and Harrold (1996) was constructed to capture the changes to a given program. They computed a slice on the constructed model to identify the affected parts. Test cases were prioritized based on the extent of their execution of the affected nodes. Their experimental study showed a significant improvement in APFD value over code-based techniques (Rothermel et al. 2001; Elbaum et al. 2002).

Panda et al. (2019) proposed a UML diagram based test scenario prioritization for object-oriented software. They have constructed a state machine graph (SMG) from a source code. Then they identified all the nodes which are got affected due to any changes in the source code. They have stored the information of affected node after each modification of the software. Next time when any changes made in the software their technique order the test cases based on the analysis of previously stored information.

Lou et al. (2015) proposed two different test case prioritization techniques with mutation testing. The test cases were prioritized based on their fault detection capabilities. Their first proposed method is statistics-based and the second method is based on a probabilistic approach. Both the approaches follow the following three steps for TCP.

First, they identify the modified part in the original program. Subsequently, they generates the mutants using the mutation operators involved in the modified part. Finally, the test cases are scheduled based on their fault detection capabilities. They have evaluated the performance of both the proposed TCP techniques on fourteen different versions of three open-source Java projects. They compared their techniques with two approaches namely addition and total. These two techniques are widely used as the control approaches in test case prioritization. Also, they applied their methods at two different levels of system design viz., test-method level and test-class level. Based on their experimental results, they reported that the statistics based TCP approach outperforms the probability based TCP as well as the other TCP techniques.

Dhareula and Ganpati (2020) proposed a flower pollination algorithm (FPA) based regression test case prioritization technique. FPA is a nature inspired metaheuristic optimization algorithm that tries to emulate the pollination process of flowering plants. Authors used only the code coverage information of test cases as input to the flower pollination algorithm. FPA does not contain the prior knowledge of the fault detection abilities of test cases. The test cases are prioritized based on the maximum coverage of faults based in the minimum execution time. They have validated their approach using APFD metric on three Java applications. Based on their experimental results, it was observed that FPA based TCP is more effective than reverse order, random order and reverse-random order of test cases.

Vescan et al. proposed an optimized test case prioritization technique using ant colony optimization algorithm. They have named their approach as Test Case Prioritization-ANT (TCP-ANT). The criteria used by the optimization algorithm is the number of faults that are not covered by the test case and the sum of the severity of the uncovered faults. Along with this, the execution time of the test cases are used to compute the pheromone deposited on the edges of the graph. They validated their proposed approach using Grep SIR (2001) data set and found it to be encouraging.

Mirarab and Tahvildari (2008) used Bayesian networks (BNs) to prioritize regression test cases. They used several code coverage metrics, fault proneness of the software components and source code quality metrics to construct a unified model using BNs. They have used APFD metric to compute the effectiveness of their approach. Later, Zhao et al. Zhao et al. (2015) extended their approach by integrating the clustering method based on code coverage and BNs. Their approach helps to break the ties between the test cases. Busjaeger et al. Busjaeger and Xie (2016) introduced the usage of SVMmap Tan et al. (2016) for TCP. They used code coverage, test-failure history, similarity between tests and changes, test case age, and fault history information for test prioritization. Different neural network models have also been used for TCP Gökçe et al. (2006); Spieker et al..

## 3 Proposed technique: HDFDC

In this section, we present our technique for test case prioritization. We have named our technique HDFDC, being an abbreviation of test cases prioritization using '**H**ard to **D**etect' bugs and **F**ault **D**etection **C**apability. We first discuss a few important terminologies that we later use to describe our HDFDC technique.

**Fault Matrix (FM)** A fault matrix (FM) is a matrix of order $n \times m$, where $n$ represents the number of test cases and $m$ represents the total number of faults present in a program. Let $TC_i$ denote the $i^{th}$ test case and $F_j$ represent the $j^{th}$ fault. The bug detection information of a test case is recorded in FM as given by Eq. 1.

$$FM_{ij} = \begin{cases} 1, & \text{if bug}j\text{is detected by test case}i \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

Table 1 represents an example of a Fault Matrix in which the rows represent the test cases and the columns represent the faults.

**Fault Vector (FV):** Fault vector FV=$(FV_0, FV_1, FV_2, ..., FV_{m-1})$ is a vector of length $m$, in which a vector component $FV_j$ represents the number of test cases that detect the $j^{th}$ fault. Variable $m$ represents the total number of faults present. Initially, for each fault $j$, $FV_j$ is *0*. Later, on the execution of each test case, $FV_j$ gets incremented if the $j^{th}$ fault is detected by that test case.

For example, fault vector $FV = (2, 1, 0, ..., 1)$. Here, $FV_0 = 2$ which shows *Fault0* is detected by two test cases. Similarly, $FV_1 = 1$, and $FV_2 = 0$ show that the *Fault1* and *Fault2* are detected by test cases 1 and 0 respectively.

**Fault Detection Capability (FDC) of a test case :** FDC measures the fault detection capability of test cases. A test case gets a higher priority based on the obtained FDC score. A higher FDC score is accorded to a test case that detects bugs which are detected by a very few test cases. FDC metric for each test case $i$ is evaluated using Eq. 2:

$$FDC_i = \sum_{j=0}^{n-1} \frac{1}{FV_j + 1}, \quad where \quad FM_{ij} = 1 \quad (2)$$

where, $j$ indicates the $j^{th}$ fault, $n$ indicates the total number of faults, FM shows the Fault Matrix and $FV$ represents the fault vector.

We illustrate computation of the FDC metric using an example provided in Table 1. Table 1 represents an example of fault matrix where bug detection information of

**Table 1** Illustration of Fault Matrix (Here, F stands for Fault)

| | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | F10 | F11 | F12 | F13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $TC_0$ | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $TC_1$ | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $TC_2$ | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| $TC_3$ | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| $TC_4$ | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| $TC_5$ | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| $TC_6$ | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| $TC_7$ | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |

eight test cases ($TC_0$ to $TC_7$) shown for fourteen faults (*Fault*0 to *Fault*13). Initially the Fault Vector will be assigned with 0 for all bugs. Hence, $FV = (0,0,0,0,0,0,0,0,0,0,0,0,0,0)$. After execution of $TC_0$, the updated FV is (1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0). It can be observed from Table 1 that $TC_1$ detects *Fault*1, *Fault*3, and *Fault*4. Based on the fault detection of $TC_1$, we can calculate the *FDC* metric for $TC_1$ using the updated *FV* by $TC_0$ as follows:

$$\begin{aligned} FDC_1 &= \frac{1}{FV_1+1} + \frac{1}{FV_3+1} + \frac{1}{FV_4+1} \\ &= \frac{1}{2} + \frac{1}{2} + \frac{1}{2} \\ &= 1.5 \end{aligned}$$

### 3.1 An overview of our HDFDC approach

Figure 1 depicts a schematic representation of our proposed technique. For a given Java class, we first generate a large number of mutants. Each mutant contains a single fault. In the Figure, $M_1$, $M_2$, ..., $M_m$ are the mutants generated by a Mutator. In the next step, each mutant is executed with the original test suite and the output is stored for each test case in a separate file. In Fig. 1, it can be seen that mutant $M_1$ is executed using the test cases $TC_1$, $TC_2$, ..., $TC_n$, and the corresponding output files are $O_{1,1}$, $O_{1,2}$, ..., $O_{1,n-1}$, $O_{1,n}$. Similarly, the original program is also executed using the test cases $TC_1$, $TC_2$, ..., $TC_n$ and generated output files are labelled $O_1$, $O_2$, ..., $O_{n-1}$, $O_n$ respectively. Subsequently, we compare the output files corresponding to each mutated version with the output files generated for the original program and generate a Fault Matrix (FM). Using the FM, we prioritize the test cases in three separate groups: *Group 1*, *Group 2*, and *Group 3*. These three groups are defined as follows.

*Group 1* This group contains all the test cases that detect 'hard to detect' bugs. All the test cases of this group are selected compulsory. If we ignore any test case of this group, there can be undetected faults. This group is prioritized based on the total number of 'hard to detect' bugs.

*Group 2* This group contains all the test cases which detect faults that are also detected by other test cases. It contains the test cases in priority order based on the capability of exposing faults which are detected by more than one test cases. To measure the fault exposing potential of a test case, we used our proposed Fault Detection Capability (FDC) metric.

*Group 3* This group contains all the test cases which detect the bugs which are already detected by the test cases of *Group 2*. Therefore this group contains all the redundant

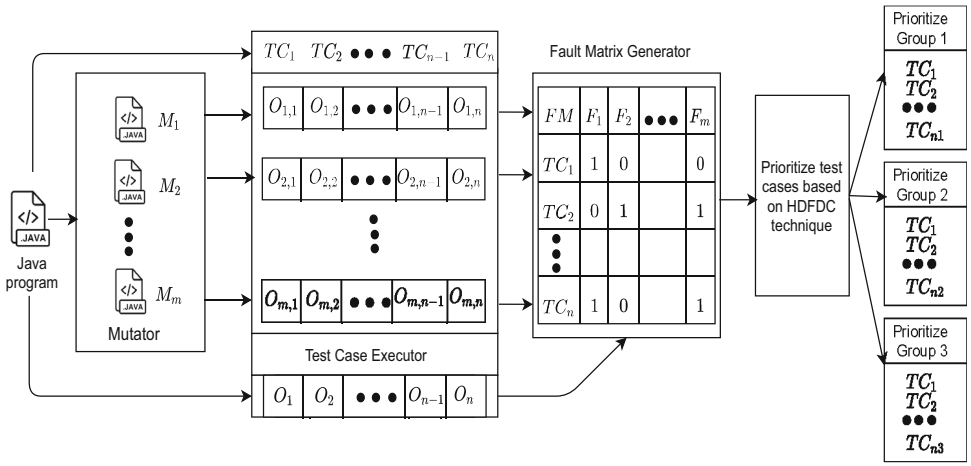


**Fig. 1** A Schematic representation of HDFDC technique

test cases. Even though Group 3 are redundant test cases, however this have slightly inferior potential of detecting bugs on a change. So, if the test budget permits, at least the more promising of these should be executed. Therefore, the redundant test cases are also prioritized based on the FDC metric.

Figure 2 represents the activity diagram model for HDFDC. Various activities that are carried out shown in Fig. 2 are described below:

- *Bugs Seeding* In this step, we select a program and seed a large number of bugs into it and create different mutants for that program. Each mutant contain a single bug.
- *Execute Mutated Program using the Test Suite* In this step, we select a mutant and execute it with all the available test cases and outputs are stored in a separate file. This step is repeated for each mutant.
- *Execute the Original Program using the Test Suite* In this step, the selected original program is executed with all the existing test cases, and the output is written into a file. For each test case, a separate file is created, which contains the result of the original output. For example, if there are $n$ test cases available for a program then $n$ output files are created.
- *Compare the Results of the Original and the Mutated Programs*: In this step, for each test case, we compare the original output of a program with the output of each mutated version of the same program. If the original output and the output of a mutated version are not the same for a test case, then we say that the test case is able to detect the fault introduced in the mutated version. For example, for a test case $TC_i$ and a mutated version $M_j$, the original output is $O_i$, and the output of the mutated version is $O_{j,i}$. If $O_i$ and $O_{j,i}$ are not the same, then test case $TC_i$ is failing for the mutant $M_j$. In other words, the test case $TC_i$ is able to detect the bug which is present in the version $M_j$.
- *Record Mutant Detected* In this step, the mutant detected by a test case is recorded in the 'Fault Matrix'.
- *Prioritize the Test Cases*: In this step, the test cases are prioritized in three different groups: *Group 1*, *Group 2*, and *Group 3*. Test cases which detect at least one 'hard to detect' bugs are prioritized in *Group 1*, and the rest of the test cases are prioritized in *Group 2*, and *Group 3* based on the FDC metric.

### 3.2 Algorithmic formulation of HDFDC

In this subsection, we present the algorithmic description of our proposed HDFDC technique for TCP.

**Algorithm 1** HDFDC

**Require:** Fault Matrix FM and Fault Vector array FV
Initialize: FV[j] = 0, for all bug $j$
Select all the test cases that detect at least one 'hard to detect' bug
Add the selected test cases into *Group 1*
Prioritize the test cases of *Group 1* based on maximum number of 'hard to detect' bugs are detected.
// Update FV array by *Prioritize Group 1* test cases:
**for** each test case i ∈ *Prioritize Group 1* **do**
  **for** each bug j **do**
    **if** (FM[i][j] ==1) **then**
      FV[j]=FV[j]+1;
    **end if**
  **end for**
**end for**
Add the remaining test cases into Group 2 (test case $TC_i \in$ *Group 2*, if $TC_i \notin$ *Group 1*)
// Compute FDC score:
**for** each test case i ∈ *Group 2* **do**
  FDC[i]=0
  **for** each bug j **do**
    **if** (FM[i][j] ==1) **then**
      FDC[i]=FDC[i]+$\frac{1}{FV[j]+1}$
      FV[j]=FV[j]+1;
    **end if**
  **end for**
**end for**
Remove the test cases which detected same bugs from *Group 2* and add the test cases into *Group 3*
Prioritize the test cases of *Group 2* and *Group 3* based on the obtained FDC score
return *Prioritize Group 1*, *Prioritize Group 2*, and *Prioritize Group 3*

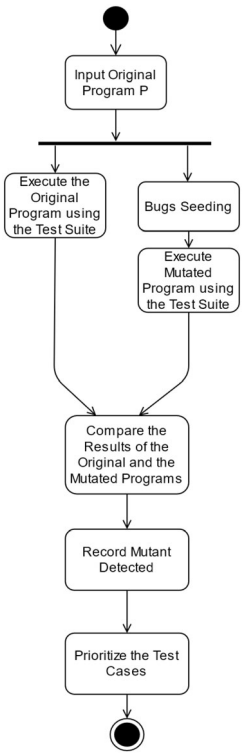


**Fig. 2** Activity diagram of our proposed HDFDC technique

HDFDC algorithm takes Fault Matrix and Fault Vector as input and generates a prioritized list of test cases as its output. Steps 2 and 3 generate *Group 1* test cases. Step 4 prioritizes the test cases present in *Group 1*. Steps 6 to 12 update the FV array after the prioritization of *Group 1* test cases. Steps 13 to 24 generate prioritized test cases of *Group 2* based on the FDC score. In Steps 15 to 23, the FDC score for each test cases of the Group 2 is computed and updated the FV array. In step 24, redundant test cases are selected from *Group 2* and the test cases are added into *Group 3* . Step 25, HDFDC algorithm prioritized the test cases of *Group 2 Group3* based on the *FDC* score and return the *Prioritized Group 1*, *Prioritized Group 2*, and *Prioritized Group 3* in Step 26.

### 3.3 An illustrative example

We illustrate the working of our HDFDC approach for TCP using the Fault Matrix in Table 2. For better visual representation, we have used '×' instead of 1 and have used 'blank; instead of 0 in the Table 2. '×' denotes that the bug is detected by the corresponding test case, and blank indicates that a particular test case does not detect that particular bug.

HDFDC first selects all the test cases which detect 'hard to detect' faults. From Table 2, it can be observed that *Fault*0, *Fault*2, *Fault*5, *Fault*6, *Fault*9, *Fault*10, and *Fault*13 are 'hard to detect' faults and these are detected by $TC_0$, and $TC_2$. So, the selected test cases are stored in *Group 1*= $\{TC_0, TC_2\}$. HDFDC technique orders the selected test cases according to maximum number of 'hard to detect' bugs detection. Among our selected test cases $TC_0$ detects three 'hard to detect' faults and $TC_2$ detects four 'hard to detect' faults. Therefore, $TC_2$ will get a higher priority than $TC_0$. The *Prioritized Group 1* is $\{TC_2, TC_0\}$. Table 3, highlighted the faults that are detected by $TC_2$ and $TC_0$.

After ordering the test cases of *Group 1*, our algorithm updates the Fault Vector FV for each test case of *Prioritized Group 1*. To start with, all the entries of the FV array are initialized to 0. Table 4, shows the initial FV array. Table 5, shows the FV array after the test cases of Group 1 have been prioritized. The remaining test cases would be added in the *Group 2*.

**Table 2** Illustration of fault matrix

| | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | F10 | F11 | F12 | F13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $TC_0$ | × | × | × | × | × | | × | | | | | | | |
| $TC_1$ | | × | | × | × | | | | | | | | | |
| $TC_2$ | | | | | | × | | | | × | × | | | × |
| $TC_3$ | | | | × | × | | | × | × | | | × | × | |
| $TC_4$ | | × | | × | × | | | × | × | | | | | |
| $TC_5$ | | | | × | | | | × | | | | | | |
| $TC_6$ | | × | | × | | | | | × | | | | | |
| $TC_7$ | | | | × | × | | | | × | | | × | × | |
| $TC_8$ | | | | × | × | | | | × | | | × | × | |
| $TC_9$ | | × | | × | | | | | × | | | | | |

**Table 3** Fault Matrix highlighted bug detection by Prioritize Group 1 test cases

| | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | F10 | F11 | F12 | F13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $TC_0$ | × | × | × | × | × | | × | | | | | | | |
| $TC_1$ | | × | | × | × | | | | | | | | | |
| $TC_2$ | | | | | | × | | | | × | × | | | × |
| $TC_3$ | | | | × | × | | | × | × | | | × | × | |
| $TC_4$ | | × | | × | × | | | × | × | | | | | |
| $TC_5$ | | | | × | | | | × | | | | | | |
| $TC_6$ | | × | | × | | | | | × | | | | | |
| $TC_7$ | | | | × | × | | | | × | | | × | × | |
| $TC_8$ | | | | × | × | | | | × | | | × | × | |
| $TC_9$ | | × | | × | | | | | × | | | | | |

**Table 4** Initial *FV* array

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

**Table 5** Fault Vector array *FV* after the *Prioritize Group 1* test cases get prioritized

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

**Table 6** Updated Fault Vector array *FV* after the computation of *FDC* of $TC_1$

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | 2 | 2 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

**Table 7** *FDC* score obtained by the test cases of *Group 2*

| Test Case | FDC Score |
|---|---|
| TC1 | 1.5 |
| TC3 | 4.66 |
| TC4 | 1.83 |
| TC5 | 0.53 |
| TC6 | 0.75 |
| TC7 | 1.59 |
| TC8 | 1.17 |
| TC9 | 0.47 |

$$Group\ 2 = \{\{TS\} - \{Group\ 1\}\}$$

where, TS represents the entire test suite

**Table 8** Prioritized order of the test cases of *Prioritize Group 2*

| Test Case | FDC Score |
|---|---|
| TC3 | 4.66 |
| TC4 | 1.83 |
| TC7 | 1.59 |
| TC1 | 1.5 |
| TC6 | 0.75 |
| TC5 | 0.53 |

$$\begin{aligned} Group\ 2 = \{\{&TC_0, TC_1, TC_2, TC_3, TC_4, TC_5, \\ &TC_6, TC_7, TC_8, TC_9\} \\ &- \{TC_2, TC_0\}\} \\ &= \{TC_1, TC_3, TC_4, TC_5, TC_6, TC_7, TC_8, TC_9\} \end{aligned}$$

HDFDC algorithm computes the FDC score of each test

**Table 9** Prioritized order of the test cases of *Prioritize Group 3*

| Test Case | FDC Score |
|---|---|
| TC8 | 1.17 |
| TC9 | 0.47 |

case present in *Group 2*. The computation of FDC score is carried out in Steps 15 to 23 of our Algorithm 1 HDFDC. The FDC metric for *TC*1 is computed using the Fault Matrix shown in Table 2 and Fault Vector shown in Table 5:

$$FDC_1 = \frac{1}{FV[1]+1} + \frac{1}{FV[3]+1} + \frac{1}{FV[4]+1} = \frac{1}{2} + \frac{1}{2} + \frac{1}{2} = 1.5$$

where, FM[1][1] =1, FM[1][3]=1, and FM[1][4]=1.

Along with the computation of the FDC score for each test case, FV array is also updated. As can be observed, the test case *TC*1 detects the faults *Fault*1, *Fault*3, and *Fault*4. So, FV[1], FV[3], and FV[4] value will be incremented by 1 (FV[j] = FV[j]+1). The updated FV array after the FDC computation of TC1 is shown in Table 6. Similarly, for the rest of the test cases of *Group 2*, HDFDC algorithm will compute the FDC score and simultaneously update the FV array.

The final FDC score for all the test cases of Group 2 is shown in Table 7. From Table 7, it can be observed that *TC*3 obtained highest FDC score. From Table 2, it can be observed that *TC*3 detects the faults *Fault*7, *Fault*8, *Fault*11, and *Fault*12 which are not detected by any other test cases before the execution of *TC*3. From the Fault Matrix in Table 2, it can be observed that *TC*8 and *TC*9 are detecting same bugs which are already detected by *TC*7 and *TC*6 respectively. So, *TC*8 and *TC*9 are added in the *Group 3* as per the algorithmic formulation of HDFDC technique. Based on the computed FDC scores, Group 2 and Group 3 are prioritized next. Prioritized Group 2 is shown in Table 8 8, and prioritized Group 3 is shown in Table 9.

## 4 Experimental studies

In this section, we first discuss the experimental setup and subject programs that we have used. We then present an evaluation metric to compare the effectiveness of different techniques. Subsequently, we analyze the obtained results and discuss the threats to the validity of the results.

**Table 10** Project characteristics

| S. No. | Project Name | LOC | No. of Classes | No. of Test cases | No. of Mutants |
|---|---|---|---|---|---|
| 1 | Book | 80 | 1 | 15 | 176 |
| 2 | Elevator | 580 | 1 | 142 | 800 |
| 3 | JFreeChart | 1481 | 1 | 654 | 1250 |
| 4 | Jtcas | 169 | 1 | 160 | 180 |
| 5 | Tritype | 106 | 1 | 65 | 50 |
| 6 | Area and Perimeter | 916 | 1 | 110 | 50 |
| 7 | Cli | 4499 | 55 | 38 | 39 |
| 8 | Closure | 149516 | 1626 | 412 | 174 |
| 9 | Codec | 5834 | 30 | 43 | 18 |
| 10 | Collections | 64908 | 898 | 506 | 4 |
| 11 | Compress | 12816 | 92 | 81 | 47 |
| 12 | Csv | 1674 | 11 | 15 | 16 |
| 13 | Gson | 11560 | 529 | 261 | 18 |
| 14 | JacksonCore | 27908 | 190 | 135 | 26 |
| 15 | JacksonDatabind | 75069 | 2448 | 590 | 112 |
| 16 | JacksonXml | 7500 | 106 | 98 | 6 |
| 17 | Jsoup | 4512 | 171 | 41 | 93 |
| 18 | JxPath | 28657 | 397 | 201 | 22 |
| 19 | Lang | 61289 | 1298 | 236 | 64 |
| 20 | Math | 186609 | 1540 | 1310 | 106 |
| 21 | Mockito | 22654 | 1406 | 319 | 38 |
| 22 | Time | 68712 | 546 | 317 | 26 |

### 4.1 Experimental setup

All experiments have been conducted on a 64-bit Ubuntu machine, 2.5 GHz Intel Core i5 processor, with 8 GB RAM. All the programs which are used to evaluate the performance of our technique are Java classes. The Java programs are compiled on JDK version 8. We have used the mutation tool muJava muJava (2016) to create the mutants of considered Java programs. All the modules were coded using Python.

### 4.2 Subject programs

To evaluate the effectiveness of our HDFDC technique, we have used twenty two publicly available Java projects. Table 10 shows the important characteristics of these Java projects. We have considered the programs *Book*, *Elevator*, *JFreeChart*, *Jtcas*, *Tritype* etc. *JFreeChart* and *Elevator* are obtained from an open source library that has been used in test coverage of Java programs Gligoric et al. (2015). The code for *Book* and *Area* and *Perimeter* classes are taken from a student assignment program. Rest of the programs are available at the open-source GitHub repository Defects4j Just et al. (2014) and SIR Rothermel et al..

### 4.3 Mutant generation

We have used muJava muJava (2016) tool for mutant generation. The muJava tool supports injection of different type of bugs that are frequently committed by the programmers. We have used class-level and method-level mutation operators. In method-level mutation operators we have used the following:

- *Arithmetic operators* One of the example of this category of bugs is replacing $++$ by $--$
- *Relational operators* One of the example of this category of bugs is replacing $>$ by $<$
- *Conditional operators* One of the example of this category of bugs is replacing&& by ||
- *Logical operators* One of the example of this category of bugs is replacing & by |
- *Assignment operators* One of the example of this category of bugs is replacing $-=$ by $+=$

In class-level category we have used muJava operators for Java-specific features, such as JTI, JTD, JSI, JSD, etc.

- *JTI* The operator inserts the keyword *this*.
- *JTD* The operator deletes uses of the keyword *this*.
- JSI : The operator adds the *static* modifier to change instance variables to class variables.
- *JSD* The operator removes the *static* modifier to change class variables to instance variables.
- *JID* The operator removes the initialization of member variables in the variable declaration so that member variables are initialized to appropriate default values of Java.

In the table 10, for the projects S. No. 1 to S. No 6, we have used muJava tool for mutant generation. Rest of the projects (S. No. 7 to S. No. 22) are taken from Defects4j repository and these projects are already seeded with real faults. The number of mutants for each of the projects are shown in the table 10.

**Table 11** Illustration of APFD value calculation

| TC\Fault | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | F10 |
|---|---|---|---|---|---|---|---|---|---|---|
| $TC_1$ | | | | | | × | × | | | |
| $TC_2$ | | × | | | × | | | | | × |
| $TC_3$ | | | | | | | | | | |
| $TC_4$ | | | | × | | | | | × | |
| $TC_5$ | | | | | | | | | | |
| $TC_6$ | × | | | | × | | | | | |
| $TC_7$ | | | × | | | | | × | | × |
| $TC_8$ | | × | | | | | × | | | |
| $TC_9$ | × | | | × | | | | × | | |
| $TC_{10}$ | | | × | | | | | | × | |

### 4.4 TCP evaluation metric

To evaluate the performance of various considered TCP techniques, we have used the average percentage of faults detected (APFD) metric. Lou et al. (2015) proposed APFD metric. It captures how fast a prioritized test suite is capable of detecting faults. A higher APFD value denotes a higher rate of fault detection. For a test suite T, APFD metric is evaluated using Eq. 3:

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n * m} + \frac{1}{2 * n} \tag{3}$$

where,

- $n$ is the total number of test cases presents in test suite $T$
- $m$ represents total number of faults to be detected
- $TF_i$ represents the position of the first test case in $T$ which detects fault $i$.

Suppose, there are ten faults ($m = 10$) and ten test cases ($n = 10$) for a given program. The obtained fault matrix is shown in Table 11. APFD value for this program is calculated as follows:

Prioritized test case order: $TC_4, TC_2, TC_1, TC_7, TC_6, TC_9, TC_{10},$

$TC_5, TC_8, TC_3$. APFD applied w.r.t prioritized Test Suite:

$$APFD = 1 - \frac{5+2+4+1+2+3+3+4+1+2}{10*10} + \frac{1}{2*10} = 1 - \frac{27}{100} + \frac{1}{20} = 1 - 0.27 + 0.05 = 0.78$$

The APFD value for non-prioritized test suite:

$$APFD = 1 - \frac{6+2+7+4+2+1+1+7+4+2}{10*10} + \frac{1}{2*10} = 1 - \frac{36}{100} + \frac{1}{20} = 1 - 0.36 + 0.05 = 0.69$$

From the above example, it can be observed that APFD value (0.78) obtained for prioritized test suite is higher than the APFD value (0.69) for non-prioritized test suite. Therefore, we can say that a prioritized order is necessary to achieve higher fault detection rate.

Another metric we have used to evaluate the performance of various TCP techniques is $APFD_C$ Elbaum et al. (2001) which is the 'cost-cognizant' version of the $APFD$ metric. $APFD_C$ consider both the test case cost and fault severity to measure the fault detection rate of a test case. For a test suite T, $APFD_C$ metric is evaluated as:

$$APFD_C = \frac{\sum_{i=1}^{m}(f_i * (\sum_{j=TF_i}^{n} t_j - \frac{1}{2} * t_{TF_i}))}{\sum_{i=1}^{n} t_i * \sum_{i=1}^{m} f_i} \quad (4)$$

where,

- $n$ is the total number of test cases presents in test suite $T$
- $m$ represents total number of faults revealed by $T$
- $TF_i$ represents the position of the first test case in $T$ which detects fault $i$.
- $t_j$ is the cost of the $j^{th}$ test case.
- $f_i$ is the severity of the $i^{th}$ fault.

### 4.5 Experimental results

We compare the results of HDFDC with six other techniques: No-Prioritization (NP), random prioritization, Statement Coverage (SC), Branch Coverage (BC), Flower Pollination Algorithm (FPA) based regression test case prioritization technique, and ant colony optimization algorithm based test case prioritization technique (TCP-ANT). No-prioritization technique considers the test cases

**Table 12** Obtained APFD scores

| S. No. | Programs | NP | Random | SC | BC | FPA | TCP-ANT | HDFDC |
|---|---|---|---|---|---|---|---|---|
| 1 | Book | 0.538 | 0.643 | 0.761 | 0.834 | 0.849 | 0.842 | 0.93 |
| 2 | Elevator | 0.461 | 0.486 | 0.691 | 0.7 | 0.743 | 0.865 | 0.927 |
| 3 | JFreeChart | 0.44 | 0.529 | 0.639 | 0.739 | 0.786 | 0.852 | 0.986 |
| 4 | Jtcas | 0.46 | 0.51 | 0.82 | 0.80 | 0.83 | 0.92 | 0.939 |
| 5 | Tritype | 0.49 | 0.542 | 0.564 | 0.581 | 0.688 | 0.795 | 0.814 |
| 6 | Area and Perimeter | 0.398 | 0.421 | 0.437 | 0.437 | 0.468 | 0.69 | 0.7 |
| 7 | Cli | 0.526 | 0.592 | 0.685 | 0.685 | 0.848 | 0.832 | 0.904 |
| 8 | Closure | 0.573 | 0.551 | 0.530 | 0.530 | 0.791 | 0.858 | 0.973 |
| 9 | Codec | 0.516 | 0.674 | 0.549 | 0.618 | 0.915 | 0.919 | 0.953 |
| 10 | Collections | 0.529 | 0.461 | 0.602 | 0.602 | 0.995 | 0.995 | 0.996 |
| 11 | Compress | 0.543 | 0.605 | 0.638 | 0.638 | 0.787 | 0.872 | 0.952 |
| 12 | CSV | 0.469 | 0.325 | 0.512 | 0.528 | 0.9 | 0.9 | 0.9 |
| 13 | Gson | 0.581 | 0.635 | 0.617 | 0.617 | 0.883 | 0.958 | 0.991 |
| 14 | JacksonCore | 0.547 | 0.788 | 0.602 | 0.602 | 0.945 | 0.935 | 0.983 |
| 15 | JacksonDatabind | 0.565 | 0.729 | 0.595 | 0.612 | 0.886 | 0.855 | 0.984 |
| 16 | JacksonXml | 0.698 | 0.908 | 0.705 | 0.705 | 0.961 | 0.998 | 0.943 |
| 17 | Jsoup | 0.453 | 0.764 | 0.706 | 0.706 | 0.794 | 0.879 | 0.936 |
| 18 | JxPath | 0.553 | 0.723 | 0.598 | 0.897 | 0.952 | 0.962 | 0.986 |
| 19 | Lang | 0.571 | 0.573 | 0.626 | 0.937 | 0.880 | 0.836 | 0.963 |
| 20 | Math | 0.566 | 0.415 | 0.541 | 0.795 | 0.913 | 0.777 | 0.985 |
| 21 | Mockito | 0.566 | 0.583 | 0.734 | 0.734 | 0.845 | 0.914 | 0.981 |
| 22 | Time | 0.490 | 0.591 | 0.593 | 0.625 | 0.860 | 0.926 | 0.978 |

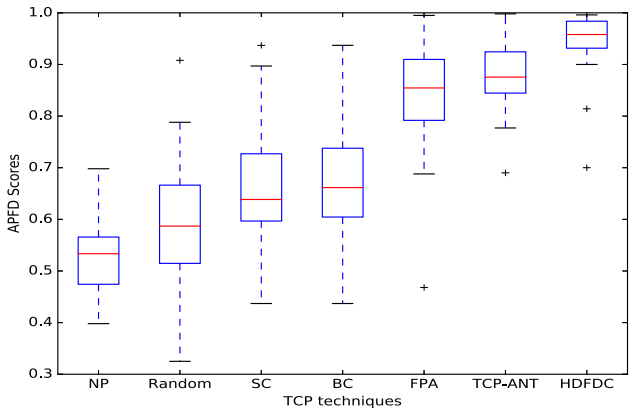


**Fig. 3** Box plot representation of APFD achieved by different TCP techniques

without any prioritization. Random prioritization technique randomizes the test cases. Rothermel et al. (2001, 1999) proposed statement coverage (SC), and branch coverage (BC) based TCP approaches. These are two prominent TCP techniques available in the literature. Dhareula and Ganpati (2020) proposed flower pollination algorithm (FPA) based-Test Prioritization technique. It is a metaheuristic-based approach which prioritize the test cases very effectively. Vescan et al. proposed TCP-ANT technique. TCP-ANT prioritized the test cases based on the severity of the uncovered faults. This technique showed very encouraging results on all the experimented programs. FPA and TCP-ANT are the recent techniques in the literature and considered *APFD* metric to measure the effectiveness of the techniques.

Table 12 shows the APFD values obtained using seven different techniques. We can observe from Table 12 that our HDFDC based TCP technique obtained the maximum APFD value (0.996) for *Collections* project and the lowest APFD value (0.7) for *Area* and *Perimeter* class. It can be observed from project characteristics Table 10, *Collections* are seeded with lowest number of mutants (4) that may lead HDFDC to achieved highest APFD for *Collections*. BC obtained a similar or a better APFD score than SC except for *Jtcas*, where SC scored (0.82) and BC scored (0.80). Similarly, Random prioritization obtained better APFD than NP for most of the projects except for *Collections*, *CSV*, and *Math*, where NP scored better APFD value than Random prioritization. It can be also observed that FPA and TCP-ANT performs better than NP, Random, SC and BC for all the programs. But, our HDFDC technique performs more effectively than FPA and TCP-ANT techniques for all the projects except *CSV* and *JacksonXml* projects. FPA, TCP-ANT, and HDFDC obtained exactly same APFD value (0.9) for *CSV* and for *JacksonXml* HDFDC obtained lower APFD (0.943) than FPA (0.961) and TCP-ANT (0.998). It can be also observed from the Table 12 that among all the techniques, HDFDC showed highest

**Table 13** Relative improvement achieved on APFD

| S.No. | Projects | NP (%) | Random (%) | SC (%) | BC (%) | FPA (%) | TCP-ANT (%) |
|---|---|---|---|---|---|---|---|
| 1 | Book | 72.86 | 44.63 | 22.2 | 11.51 | 9.54 | 10.45 |
| 2 | Elevator | 101.08 | 90.74 | 34.15 | 32.42 | 24.76 | 7.16 |
| 3 | JFreeChart | 124.09 | 86.38 | 54.30 | 33.41 | 25.44 | 15.72 |
| 4 | Jtcas | 104.13 | 84.11 | 14.51 | 17.37 | 13.13 | 2.06 |
| 5 | Tritype | 66.12 | 50.48 | 44.32 | 40.10 | 18.31 | 2.58 |
| 6 | Area and Perimeter | 75.87 | 66.27 | 60.18 | 60.18 | 49.57 | 1.44 |
| 7 | Cli | 71.86 | 52.70 | 31.97 | 31.97 | 6.60 | 8.65 |
| 8 | Closure | 69.8 | 76.5 | 83.5 | 83.5 | 23 | 13.40 |
| 9 | Codec | 84.68 | 41.39 | 73.5 | 54.2 | 4.15 | 3.69 |
| 10 | Collections | 88.27 | 116.05 | 65.44 | 65.44 | 0.10 | 0.10 |
| 11 | Compress | 75.32 | 57.35 | 49.21 | 49.21 | 20.96 | 9.17 |
| 12 | CSV | 91.89 | 176.92 | 75.78 | 70.45 | 0 | 0 |
| 13 | Gson | 70.56 | 56.06 | 60.61 | 60.61 | 12.23 | 3.44 |
| 14 | JacksonCore | 79.7 | 24.74 | 63.28 | 63.28 | 4.08 | 5.13 |
| 15 | JacksonDatabind | 74.15 | 34.97 | 65.37 | 60.78 | 11.06 | 15.08 |
| 16 | JacksonXml | 35.1 | 3.854 | 33.569 | 33.569 | −1.87 | −5.51 |
| 17 | Jsoup | 106.6 | 22.51 | 32.5 | 32.5 | 17.88 | 6.4 |
| 18 | JxPath | 81.23 | 36.37 | 64.88 | 9.92 | 3.57 | 2.49 |
| 19 | Lang | 68.65 | 68.06 | 53.83 | 2.77 | 9.43 | 15.1 |
| 20 | Math | 74.02 | 137.34 | 82.07 | 23.89 | 7.88 | 26.66 |
| 21 | Mockito | 73.32 | 68.26 | 33.65 | 33.65 | 16.09 | 7.33 |
| 22 | Time | 99.59 | 65.48 | 64.92 | 56.48 | 13.72 | 5.61 |

**Table 14** Total average relative improvement on APFD achieved for each of the project

| S.No. | Projects | Total average relative improvement (%) |
|---|---|---|
| 1 | Book | 28.53 |
| 2 | Elevator | 48.53 |
| 3 | JFreeChart | 56.55 |
| 4 | Jtcas | 39.21 |
| 5 | Tritype | 36.95 |
| 6 | Area and Perimeter | 52.25 |
| 7 | Cli | 33.95 |
| 8 | Closure | 58.28 |
| 9 | Codec | 43.60 |
| 10 | Collections | 55.9 |
| 11 | Compress | 43.53 |
| 12 | CSV | 69.17 |
| 13 | Gson | 43.91 |
| 14 | JacksonCore | 40.02 |
| 15 | JacksonDatabind | 43.56 |
| 16 | JacksonXml | 16.44 |
| 17 | Jsoup | 30.98 |
| 18 | JxPath | 33.07 |
| 19 | Lang | 36.30 |
| 20 | Math | 58.64 |
| 21 | Mockito | 38.71 |
| 22 | Time | 50.96 |

average APFD (0.941) and NP showed lowest Average APFD score (0.524). SC and BC prioritization techniques achieved very similar average APFD scores (0.665 & 0.678 respectively). If we arrange them in decreasing order of average APFD scores, the arrangement will be: HDFDC (0.941) > TCP-ANT (0.88) > FPA (0.841) > BC (0.678) > SC (0.665) > Random (0.593) > NP (0.524).

Figure 3 represents the box plot diagram showing the APFD scores achieved on twenty-two java projects by seven different prioritization techniques. As can be seen from Fig. 3, both the outlying and the maximum values of NP and Random are much smaller than corresponding values of other five techniques. TCP-ANT achieved highest maximum value compared to others techniques. Among all the techniques, HDFDC achieved highest median. This implies in most cases HDFDC performs better than other six techniques under consideration. The performance of SC and BC are much similar and they outperform NP and Random with larger median and maximum values. TCP-ANT has shorter interquartile range (IQR) and slightly larger median than FPA that means its performance is more effective and consistent than FPA. HDFDC has the shortest IQR among all the seven techniques. This implies HDFDC showed most consistent performance for all the projects under experimentation.

Table 13 shows relative improvement achieved in APFD score by the HDFDC technique over all the other

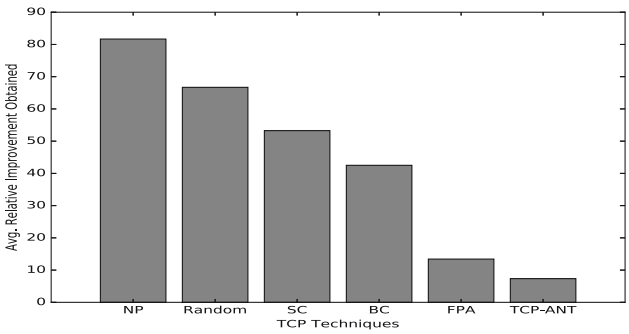


**Fig. 4** Average relative improvement achieved on APFD over existing methods using our HDFDC technique

TCP techniques. We can observe that HDFDC shows a significant relative improvement over NP and Random prioritization for all the projects under test. It shows highest relative improvement (176.92%) over Random prioritization for *CSV* project among all other techniques considering all the projects. HDFDC performs more effectively than SC and BC for all twenty-two Java projects. Among SC and BC, HDFDC shows lowest relative improvement (2.77%) over BC for *Lang* project and it achieves same highest relative improvement (83.5%) over both SC and BC for *Closure* project. Whereas, as compared with FPA and TCP-ANT, HDFDC showed positive improvement in relative APFD value for most of the

**Table 15** Obtained $APFD_C$ scores

| S. No. | Programs | NP | Random | SC | BC | FPA | TCP-ANT | HDFDC |
|---|---|---|---|---|---|---|---|---|
| 1 | Book | 0.435 | 0.564 | 0.489 | 0.546 | 0.722 | 0.798 | 0.811 |
| 2 | Elevator | 0.297 | 0.321 | 0.398 | 0.42 | 0.618 | 0.696 | 0.784 |
| 3 | JFreeChart | 0.3 | 0.296 | 0.487 | 0.287 | 0.656 | 0.793 | 0.963 |
| 4 | Jtcas | 0.226 | 0.274 | 0.43 | 0.43 | 0.805 | 0.876 | 0.898 |
| 5 | Tritype | 0.44 | 0.378 | 0.522 | 0.541 | 0.727 | 0.794 | 0.851 |
| 6 | Area and Perimeter | 0.387 | 0.482 | 0.428 | 0.428 | 0.433 | 0.62 | 0.657 |
| 7 | Cli | 0.412 | 0.559 | 0.655 | 0.655 | 0.828 | 0.815 | 0.891 |
| 8 | Closure | 0.426 | 0.301 | 0.369 | 0.269 | 0.704 | 0.741 | 0.950 |
| 9 | Codec | 0.415 | 0.489 | 0.344 | 0.452 | 0.848 | 0.856 | 0.925 |
| 10 | Collections | 0.348 | 0.289 | 0.459 | 0.359 | 0.986 | 0.986 | 0.984 |
| 11 | Compress | 0.395 | 0.451 | 0.481 | 0.481 | 0.742 | 0.819 | 0.936 |
| 12 | CSV | 0.262 | 0.244 | 0.408 | 0.426 | 0.909 | 0.909 | 0.909 |
| 13 | Gson | 0.272 | 0.406 | 0.396 | 0.344 | 0.857 | 0.917 | 0.975 |
| 14 | JacksonCore | 0.459 | 0.639 | 0.459 | 0.359 | 0.802 | 0.827 | 0.949 |
| 15 | JacksonDatabind | 0.456 | 0.588 | 0.436 | 0.436 | 0.657 | 0.718 | 0.961 |
| 16 | JacksonXml | 0.556 | 0.821 | 0.541 | 0.541 | 0.931 | 0.987 | 0.943 |
| 17 | Jsoup | 0.468 | 0.760 | 0.473 | 0.554 | 0.76 | 0.878 | 0.938 |
| 18 | JxPath | 0.434 | 0.554 | 0.389 | 0.584 | 0.806 | 0.899 | 0.971 |
| 19 | Lang | 0.346 | 0.298 | 0.489 | 0.586 | 0.704 | 0.646 | 0.908 |
| 20 | Math | 0.388 | 0.310 | 0.352 | 0.447 | 0.647 | 0.611 | 0.957 |
| 21 | Mockito | 0.332 | 0.265 | 0.506 | 0.506 | 0.773 | 0.807 | 0.941 |
| 22 | Time | 0.31 | 0.329 | 0.426 | 0.522 | 0.829 | 0.862 | 0.945 |

projects except for *CSV* and *JacksonXml*. HDFDC showed no improvement (0%) over FPA and TCP-ANT for *CSV*. On the other hand, for *JacksonXml* project, as compared with FPA and TCP-ANT, HDFDC showed deterioration in relative APFD value by −1.87% and −5.51% respectively. HDFDC showed less than 1% relative improvement over FPA and TCP-ANT for *Collections*, *CSV*, and *JacksonXml* projects. One of the reasons for this low effectiveness of HDFDC for these specific projects is that it adds most of the test cases in *Group 2*. One of the observations can be drawn from here is that HDFDC performs more effectively when there are more test cases in the *Group 1*. In other words, when there are more numbers of 'hard to detect' faults in the program, HDFDC shows a significant improvement in early fault detection.

Table 14 shows total average relative improvement by the HDFDC technique over all other techniques for each of the project. It can be observed that HDFDC showed the highest total average relative improvement (69.17%) for *CSV* project and the lowest total average relative improvement (16.44%) for *JacksonXML*. It can be observed from Table 13, for *CSV* project, HDFDC showed significant relative improvement over NP, Random, SC, and BC that leads HDFDC to achieved highest total average relative improvement for *CSV* project. On the other hand, for *JacksonXML*, HDFDC showed deterioration in relative improvement over FPA and TCP-ANT. That is the main reason for HDFDC to achieved lowest total average relative improvement for *JacksonXML* project. Another observation can be made from Table 14, for *Math* and *Closure* HDFDC achieved 58.64% and 58.28% total average relative improvement respectively which are the second and third highest total average relative improvement among all the projects. Now, it can be observed from Table 10, *Math* and *Closure* has highest number of lines of code (LOC) compared to other projects. So, one of the observations can be drawn from this is that our HDFDC technique performs better for large projects.

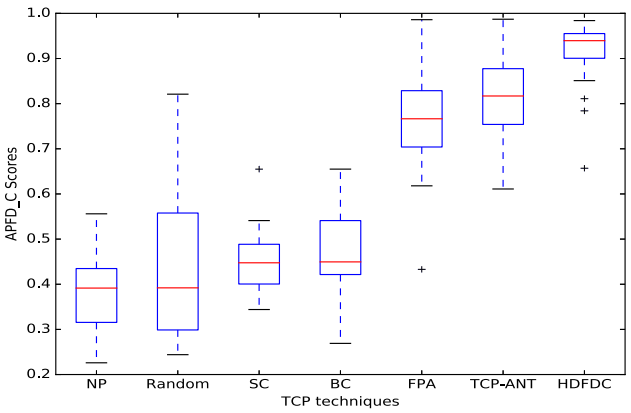


**Fig. 5** Box plot representation of $APFD_C$ achieved by different TCP techniques

Figure 4 represents the average relative improvement achieved by the HDFDC technique over NP, Random, SC, BC, FPA, and TCP-ANT techniques. Among this six techniques, HDFDC showed the highest average relative improvement over NP and the lowest average relative improvement over TCP-ANT. On an average, it shows relative improvement in APFD value by 73.85% than NP and Random. On the other hand, HDFDC shows an average relative increment in APFD value by 28.81% over SC, BC, FPA, and TCP-ANT. On an average HDFDC shows 43.82% average relative improvement over the existing TCP techniques. This improvement may be because our technique considers different types of bugs based on their difficulty of detection. So, it may lead to a situation where the test cases that detect 'hard to detect' bugs have a higher probability of revealing more faults than the test cases that detect normal bugs.

We have also conducted experimental evaluation based on $APFD_C$ achieved by different TCP techniques of our consideration. $APFD_C$ achieved by different techniques shown in Table 15. From the Table 15, we can observed that No Prioritization (NP) showed lowest $APFD_C$ score (0.226) for *Jtcas* followed by Random prioritization which showed second lowest $APFD_C$ score (0.244) for *CSV* Project. NP and Random showed poor performance compared to other techniques in terms of $APFD_C$. Like our previous *APFD* scores analysis, SC and BC perform very similar on $APFD_C$ also. TCP-ANT outperformed FPA for more most of the projects and $TCP-ANT$ achieved highest $APFD_C$ (0.987) for *JacksonXml* project among all the techniques. Our technique *HDFDC* showed lowest $APFD_C$ (0.657) for *Area* and *Perimeter* and highest $APFD_C$ (0.984) for *Collections* project. In our previous analysis on APFD, we found that there also HDFDC achieved lowest and highest *APFD* scores for the same projects *Area* and *Perimeter* and *Collection* respectively. For most of the considered projects, HDFDC obtained higher $APFD_C$ than other techniques. This established that HDFDC helps detect more severe faults earlier in a regression testing cycle.

Figure 5 shows the box plot representation of $APFD_C$ scores achieved by seven different techniques. As can be seen from Fig. 5, Random technique has largest IQR than other techniques that means Random showed more variance in $APFD_C$ scores than all other techniques under consideration. SC showed shorter IQR than BC than means in terms of $APFD_C$ scores, SC showed more consistency than BC. FPA and TCP-ANT has very similar IQR but TCP-ANT showed slightly larger median than FPA. Like box plot representation of APFD scores (Fig. 3), here in $APFD_c$ box plot also HDFDC has shortest IQR and largest median among all the seven techniques which implies HDFDC showed most consistent performance and more effective than other techniques in terms of cost and severe faults detection.

### 4.6 Threats to the validity

- Our experiments have been carried out using a limited set of data. However, to mitigate this risk, we have considered several programs with widely varying LOCs, complexity, and specifications.
- We have injected only basic programming bugs using the mutation tool. We have not injected higher-order bugs such as algorithmic errors. However, it has been reported that basic programming bugs can lead to higher order mutants Nguyen and Madeyski (2014).
- In our experimentation, every subject program contains exactly one seeded fault. On the other hand, in practice programs can have more than one bug. However, since debugging is carried out by considering one test failure at a time, we argue that our approach does not compromise the prioritization of the test cases.
- We have considered the test cases which are available in Defects4j Just et al. (2014) and SIR Rothermel et al. repository for any particular project. Results might differ for other test cases. However, the two considered repositories contain programs that are realistic and have widely differing characteristics.

## 5 Conclusion

We have proposed a TCP technique based on the capability of test cases to detect 'hard to detect' bugs. Our TCP algorithm prioritized the test cases in three different groups based on 'hard to detect' bugs and overlapped bugs. For this, we have proposed a metric called Fault Detection Capability (FDC) to measure the bug detection capability of test cases. We have compared the effectiveness of our proposed HDFDC technique with existing prioritization techniques. Our experimental results show that on an average, our HDFDC technique achieves 73.85% relative improvement over No Prioritization (NP) and Random Prioritization, 52.88% relative improvement of APFD over statement coverage (SC), 42.13% than branch coverage (BC) based techniques, and on an average 10.11% relative improvement over Flower Pollination Algorithm (FPA) based TCP technique, and Test Case Prioritization technique based on ANT colony optimization algorithm (TCP-ANT). Our HDFDC technique also shows a significant improvement on $APFD_C$ over other existing TCP techniques. That implies HDFDC is effective in terms of cost and severe faults detection. Another advantage of our

technique over existing techniques is that our works not required prepossessing steps like code instrumentation, coverage analysis etc.

We have considered class-level and method-level mutation operators for seeding bugs into a program. In future, we plan to extend our approach by injecting other types of bugs such as inter-class mutants. At present, we have considered only single bug per program. But, in practice, a program may contain multiple faults. We are planning to handle programs with multiple faults as our future work.

**Funding** No funding used.

**Declarations**

**Conflict of interest** The authors declare no conflict of interest.

## References


Busjaeger B, Xie T (2016) Learning for test prioritization: an industrial case study, in: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 975–980

Dhareula P, Ganpati A (2020) Flower pollination algorithm for test case prioritization in regression testing. In: ICT Analysis and Applications, Springer 155–167

Dutta A, Kumar S, Godboley S (2019) Enhancing test cases generated by concolic testing. In: Proceedings of the 12th Innovations on Software Engineering Conference (formerly known as India Software Engineering Conference), pp. 1–11

Elbaum S, Malishevsky AG, Rothermel G (2002) Test case prioritization: a family of empirical studies. IEEE Transact Softw Eng 28(2):159–182. https://doi.org/10.1109/32.988497

Elbaum S, Rothermel G, Kanduri S, Malishevsky AG (2004) Selecting a cost-effective test case prioritization technique. Softw Qual J 12(3):185–210. https://doi.org/10.1023/B:SQJO.0000034708.84524.22

Elbaum S, Malishevsky A, Rothermel G (2001) Incorporating varying test costs and fault severities into test case prioritization. In: Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001, pp. 329–338. https://doi.org/10.1109/ICSE.2001.919106

Gligoric M, Groce A, Zhang C, Sharma R, Alipour MA, Marinov D (2015) Guidelines for coverage-based comparisons of non-adequate test suites. ACM Transact Softw Eng Methodol (TOSEM) 24(4):1–33

Gökçe N, Eminov M, Belli F (2006) Coverage-based, prioritized testing using neural network clustering. In: International Symposium on Computer and Information Sciences, Springer, pp. 1060–1071

Gregg R, Mary Jean H, Dedhia, (1999) Regression test selection for c++ software. Tech rep, USA

Harrold MJ, Jones JA, Li T, Liang D, Orso A, Pennings M, Sinha S, Spoon SA, Gujarathi A (2001) Regression test selection for java software. SIGPLAN Not 36(11):312–326. https://doi.org/10.1145/504311.504305

Jones JA, Harrold MJ (2003) Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Trans Software Eng 29(3):195–209

Just R, Jalali D, Ernst MD (2014) Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In: ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis, San Jose, CA, USA, pp. 437–440, tool demo

Korel B, Tahat LH, Harman M (2005) Test prioritization using system models. In: 21st IEEE International Conference on Software Maintenance (ICSM'05), 2005, pp. 559–568. https://doi.org/10.1109/ICSM.2005.87

Krishnamoorthi R, Sahaaya Arul Mary SA (2009) Factor oriented requirement coverage based system test case prioritization of new and regression test cases. Inf Softw Technol 51(4):799–808. https://doi.org/10.1016/j.infsof.2008.08.007

Larsen L, Harrold MJ (1996) Slicing object-oriented software. In: Proceedings of the 18th International Conference on Software Engineering, ICSE '96, IEEE Computer Society, Washington, DC, USA, pp. 495–505. http://dl.acm.org/citation.cfm?id=227726.227837

Lou Y, Hao D, Zhang L (2015) Mutation-based test-case prioritization in software evolution. In: IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). IEEE 2015:46–57

Malishevsky AG, Ruthruff JR, Rothermel G, Elbaum S (2006) Cost-cognizant test case prioritization. Tech rep

Mall R (2018) Fundamentals of software engineering. PHI Learning Pvt. Ltd., Delhi

Mirarab S, Tahvildari L (2008) An empirical study on bayesian network-based approach for test case prioritization. In: 2008 1st International Conference on Software Testing, Verification, and Validation, IEEE, pp. 278–287

muJava (2016) https://cs.gmu.edu/offutt/mujava/

Nguyen Q-V, Madeyski L (2014) Problems of mutation testing and higher order mutation testing. Adv Intell Syst Comput 282:157–172. https://doi.org/10.1007/978-3-319-06569-4_12

Panda N, Acharya AA, Mohapatra D (2019) Test scenario prioritization for object-oriented systems using uml diagram. Int J Syst Assur Eng Manag. https://doi.org/10.1007/s13198-019-00759-z

Panigrahi CR, Mall R (2010) Model-based regression test case prioritization. In: Prasad SK, Vin HM, Sahni S, Jaiswal MP, Thipakorn B (eds) Information systems. Technology and Management, Springer, Berlin Heidelberg, Berlin, Heidelberg, pp 380–385

Panigrahi CR, Mall R (2013) An approach to prioritize the regression test cases of object-oriented programs. CSI Transact ICT 1(2):159–173. https://doi.org/10.1007/s40012-013-0011-7

Panigrahi CR, Mall R (2013) A heuristic-based regression test case prioritization approach for object-oriented programs. Innovations Syst Softw Eng 10:155–163

Rothermel Harrold (1994) Selecting regression tests for object-oriented software. In: Proceedings 1994 International Conference on Software Maintenance, pp. 14–25. https://doi.org/10.1109/ICSM.1994.336793

Rothermel G, Elbaum S, Kinneer A, Do H, Software-artifact infrastructure repository, http://sir.unl.edu/portal

Rothermel G, Untch RH, Chengyun Chu, Harrold MJ (1999) Test case prioritization: an empirical study. In: Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). Software Maintenance for Business Change (Cat. No.99CB36360), pp. 179–188

Rothermel G, Untch RH, C Chengyun, Harrold MJ (2001) Prioritizing test cases for regression testing. IEEE Transact Softw Eng 27(10):929–948. https://doi.org/10.1109/32.962562

SIR (2001)https://sir.csc.ncsu.edu/portal/index.php

Spieker H, Gotlieb A, Marijan D, Mossige M, Reinforcement learning for automatic test case prioritization and selection in continuous integration, arXiv preprint arXiv:1811.04122

Srikanth H, Williams L, Osborne J (2005) System test case prioritization of new and regression test cases. In: 2005 International Symposium on Empirical Software Engineering, 2005., p. 10. https://doi.org/10.1109/ISESE.2005.1541815

Tan P-N, Steinbach M, Kumar V (2016) Introduction to data mining. Pearson Education, India

Vescan A, Pintea C.-M, Pop PC, Test case prioritization–ant algorithm with faults severity, Logic Journal of the IGPL

Zhao X, Wang Z, Fan X, Wang Z (2015) A clustering-bayesian network based approach for test case prioritization. In: 2015 IEEE 39th Annual Computer Software and Applications Conference, Vol. 3, IEEE, pp. 542–547