The Journal of Systems & Software 192 (2022) 111419

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Test case prioritization using partial attention ☆

Quanjun Zhang, Chunrong Fang *, Weisong Sun, Shengcheng Yu, Yutao Xu, Yulei Liu

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, 210093, China

ARTICLE INFO

Article history:
Received 22 March 2022
Received in revised form 24 May 2022
Accepted 21 June 2022
Available online 24 June 2022

Keywords:
Software testing
Regression testing
Test case prioritization
Greedy algorithm

ABSTRACT

Test case prioritization (TCP) aims to reorder the regression test suite with a goal of increasing the fault detection rate. Various TCP techniques have been proposed based on different prioritization strategies. Among them, the *greedy-based* techniques are the most widely-used TCP techniques. However, existing *greedy-based* techniques usually reorder all candidate test cases in prioritization iterations, resulting in both efficiency and effectiveness problems. In this paper, we propose a generic *partial attention* mechanism, which adopts the previous priority values (i.e., the number of additionally-covered code units) to avoid considering all candidate test cases. Incorporating the mechanism with the *additional-greedy* strategy, we implement a novel coverage-based TCP technique based on *partition ordering* (OCP). OCP first groups the candidate test cases into different partitions and updates the partitions on the descending order. We conduct a comprehensive experiment on 19 versions of Java programs and 30 versions of C programs to compare the effectiveness and efficiency of OCP with six state-of-the-art TCP techniques: *total-greedy*, *additional-greedy*, *lexicographical-greedy*, *unify-greedy*, *art-based*, and *search-based*. The experimental results show that OCP achieves a better fault detection rate than the state-of-the-arts. Moreover, the time costs of OCP are found to achieve 85%–99% improvement than most state-of-the-arts.

1. Introduction

During software maintenance and evolution, software engineers usually perform code modification due to the fixing of detected bugs, the adding of new functionalities, or the refactoring of system architecture (Elsner et al., 2021; Lam et al., 2020). Regression testing is conducted to ensure that the code modification does not introduce new bugs. However, regression testing can be very time-consuming because of a large number of reused test cases (Wong et al., 1997; Gligoric et al., 2015a; Zhang, 2018; Cruciani et al., 2019). For example, Rothermel et al. (1999) report that it takes seven weeks to run the entire test suite for an industrial project. Besides, with the practices of rapid release (Mäntylä et al., 2015) and continuous integration (Elbaum et al., 2014), the available time for test execution recently keeps decreasing. For example, Memon et al. (2017) report that Google performs an amount of 800K builds and 150M test runs on more than 13K projects every day, consuming a lot of computing resources.

To address the overhead issues of regression testing, test case prioritization (TCP) has become one of the most extensively investigated techniques (Sadri-Moshkenani et al., 2022; do Prado Lima and Vergilio, 2022). Generally speaking, TCP reschedules the execution sequence of test cases in the entire test suite with the goal of detecting faults as early as possible. Traditional TCP techniques (Wong et al., 1998; Khatibsyarbini et al., 2018; Yoo and Harman, 2012) usually involve an elementary topic, prioritization strategies, which incorporate test adequacy criteria (e.g., code coverage) to represent different behaviors of test cases. In previous work, the most widely-investigated prioritization strategies are *greedy-based* strategies (Rothermel et al., 1999) (i.e., the *total-greedy* and *additional-greedy* strategies), which are generic for different coverage criteria. Given a coverage criterion (e.g., statement or method coverage), the *total-greedy* strategy selects the next test case yielding the highest coverage, whereas the *additional-greedy* strategy selects the next test case covering the maximum code units not covered in previous iterations. The recent empirical results show that although conceptually simple, the *additional-greedy* technique has been widely recognized as one of the most effective TCP techniques on average in terms of fault detection rate (Luo et al., 2016; Lu et al., 2016; Luo et al., 2018; Chi et al., 2018; Chen et al., 2018; Cheng et al., 2021).

Compared with the *total-greedy* strategy, the *additional-greedy* strategy empirically performs outstandingly due to its feedback

☆ Editor: W. Eric Wong.

* Corresponding author.
E-mail addresses: quanjun.zhang@smail.nju.edu.cn (Q. Zhang), fangchunrong@nju.edu.cn (C. Fang), weisongsun@smail.nju.edu.cn (W. Sun), yusc@smail.nju.edu.cn (S. Yu), MF21320170@smail.nju.edu.cn (Y. Xu), 1515999248@qq.com (Y. Liu).

https://doi.org/10.1016/j.jss.2022.111419
0164-1212/

mechanism, where the next test case selection takes into account the effect of already prioritized test cases (Zhang et al., 2013a; Eghbali and Tahvildari, 2016). However, there also exists a shortcoming in the *additional-greedy* strategy. Given a regression test suite T with n test cases, when selecting the ith test case, the remaining $n - i + 1$ candidate test cases need to be updated. Specifically, for each candidate test case, all not-yet-covered code units are examined, of which those covered by the candidate test case are identified. The priority values of candidate test cases need to be measured based on the feedback binary states of each statement (i.e., covered or not covered). As a result, the priority values of the candidate test cases in the previous iterations are lost and need to be recalculated in current iteration.

However, due to considering all candidate test cases in each iteration, the *additional-greedy* strategy may suffer from the efficiency problem. For example, consider 3 candidate test cases, expressed as t_1, t_2 and t_3, in a certain iteration, covering 4, 3 and 2 additional statements, respectively. After the test case t_1 is selected, t_2 and t_3 need to be updated in the next iteration. Ideally, the remaining test cases can cover a maximum of 3 additional statements in the next iteration, and only test case t_2 potentially satisfies the hypothesis. If not, a further hypothesis that both test cases t_2 and t_3 have a maximum of 2 additionally-covered statements is considered, and so on. As a result, the test cases that cover more statements in the previous iteration are more likely to maintain the advantage in the next iteration, as the test cases cannot cover more statements in the next iteration than in the previous iteration. For example, updating the test cases covering no code units in previous iterations is unnecessary until the prioritization process repeats. Thus, the *additional-greedy* strategy, which considers all candidate test cases at once in each iteration, may bring redundant calculation in efficiency.

Besides, there is a high possibility of tie-occurring when considering all candidate test cases, may lead to a decrease performance in the effectiveness. In the above example, a tie may occur if both t_2 and t_3 are considered at once (i.e., t_2 and t_3 has the highest coverage of statements not yet covered). When facing a tie, the *additional-greedy* strategy implicitly assumes that all remaining candidates are equally important, and selects one randomly. However, previous empirical studies (Eghbali and Tahvildari, 2016) have shown that the probability of ties is relatively high in the *additional-greedy* strategy, and the random tie-breaking strategy can be ineffective. It can be observed that due to considering all candidate test cases in each iteration, the *additional-greedy* strategy suffers from the both efficiency and effectiveness problems.

In this paper, to address the issues mentioned above, we propose a generic concept, partial attention mechanism, to avoid considering all candidate test cases with previous priority values (i.e., the number of additionally-covered code units) . We also apply the concept to the *additional-greedy* strategy and implement a novel coverage-based TCP technique based on the notion of partition ordering (OCP). Our technique pays attention to the partial test cases instead of the whole candidate test set with the help of priority values calculated in the previous prioritization iteration. The key idea of our technique is as follows: the priority values of the candidate test cases in the previous iteration can be regarded as a reference in the next iteration, so as to avoid considering all candidates at the same time. To implement this idea, all candidate test cases are classified into different partitions based on their previous priority values. Then among the candidates that have the highest priority value in the previous iteration, the one with the unchanged coverage of not-yet-covered code units is selected. Likewise, if no test case meets the selection criterion, test cases with the second highest priority values are considered, and so on.

We perform an empirical study to compare OCP with six state-of-the-art TCP techniques in terms of testing effectiveness and efficiency on 19 versions of four Java programs, and 30 versions of five real-world Unix utility programs. The empirical results demonstrate that OCP can outperform state-of-the-arts in terms of fault detection rate. OCP is also observed to have much less prioritization time than most state-of-the-arts (except the *total-greedy* strategy, a low bound control TCP technique) and the improvement can reach 85%–99% on average. We view our proposed technique as an initial framework to control the balance of full prioritization and partial prioritization during TCP, and believe more techniques can be derived based on our technique.

In particular, the contributions of this paper are as follows:

- We propose the first notion of the partial attention mechanism that uses previous priority values to avoid considering all candidate test cases in TCP.
- We apply the partial attention mechanism to the *additional-greedy* strategy, leading a novel coverage-based TCP technique based on partition ordering (OCP).
- We conduct an empirical study to investigate the effectiveness and efficiency of the proposed technique compared to six state-of-the-art TCP techniques.
- We release the relevant materials (including source code, subject programs, test suites and mutants) used in the experiments for replication and future research (Anon., 0000a).

The rest of this paper is organized as follows. Section 2 reviews some background information and presents a motivation example. Section 3 introduces the proposed approach. Section 4 presents the research questions, and explains details of the empirical study. Section 5 provides the detailed results of the study and answers the research questions. Section 6 discusses some related work, and Section 7 discusses the threats to validity of our experiments. Section 8 presents the conclusions and discusses future work.

2. Background & motivation

In this section, we provide some background information about test case prioritization and a motivating example.

2.1. Test case prioritization

Test case prioritization (TCP) (Rothermel et al., 1999) aims to reorder the test cases to maximize the value of an objective function (e.g., exposing faults earlier, Elbaum et al., 2000; or reducing the execution time cost, Zhang et al., 2009; Mei et al., 2012). TCP problem is formally defined as follows:

Definition 1 (*Test Case Prioritization*)**.** Given a test suite T, PT is the set of its all possible permutations, and f is an object function defined to map PT to real numbers $\mathbb{R}$. The problem of TCP (Rothermel et al., 1999) is to find $P' \in PT$, such that $\forall P'',\ P'' \in PT(P'' \neq P'),\ f(P') \geq f(P'')$.

However, it is infeasible to obtain the fault detection capability of test cases before test execution. Therefore, some alternative metrics (e.g., structural coverage), which are in some way correlated with the fault detection rate, are adopted to guide the prioritization process instead (Rothermel et al., 1999; Wang et al., 2017). Among all metrics, code coverage is the most widely used one (Luo et al., 2016; Lou et al., 2019). Intuitively, once a criterion is chosen, a specific prioritization strategy is used to order the test cases according to the chosen criterion, such as the *greedy-based* strategies (Elbaum et al., 2000), *search-based* strategies (Li et al., 2007), and *art-based* strategies (Jiang et al., 2009).

ID	Program	Test cases			
		t_1 (1, 0)	t_2 (1, 1)	t_3 (0, 2)	t_4 (2, 1)
s_1	if (a == 0)	√	√	√	√
s_2	return b;	×	×	√	×
s_3	while (b != 0) {	√	√	×	√
s_4	if (a > b)	×	√	×	√
s_5	a = a - b; // b = a-b;	×	×	×	√
s_6	else b = b - a;}	×	√	×	×
s_7	return a;	√	√	×	×
	Result	Pass	Pass	Pass	Fail

Fig. 1. A motivating example.

2.2. A motivating example

To better illustrate the details of OCP, Fig. 1 shows a piece of code with a fault in line s_5, which can be detected by the test case t_4. The code is a method that computes the greatest common divisor using the subtract-based version of Euclid's algorithm (Weimer et al., 2009; Gazzola et al., 2017). The source code is on the left and four test cases with their statement coverage information are on the right.

Before explaining the details of OCP, we first review the steps that the *additional-greedy* strategy takes to prioritize the four test cases. In the first iteration, the *additional-greedy* strategy chooses the test case t_2 with the maximum coverage. To continue, in the second iteration, the *additional-greedy* strategy selects the test case with the maximum coverage of not-yet-covered statements, e.g., s_2 and s_5. The *additional-greedy* strategy updates the coverage states for all remaining test cases and faces a tie where both t_3 and t_4 cover one of the not-yet-covered statements. In such a case, a random one (e.g., t_3 or t_4) will be selected. In the third iteration, the *additional-greedy* strategy searches for the test case, which yields the maximum coverage of statements that the first and the second test case have not covered, and t_4 or t_3 will be selected. In other words, in each iteration, the *additional-greedy* strategy selects the test case that provides the maximum coverage for the not-yet-covered statements. In the fourth iteration, the last test case t_1 is selected and this procedure continues until the ordering is complete. As a result, the test sequence of the *additional-greedy* strategy is $\langle t_2, t_3, t_4, t_1\rangle$ or $\langle t_2, t_4, t_3, t_1\rangle$ with the APFD values ranging from 0.375 to 0.625. However, as discussed in Section 1 the *additional-greedy* strategy needs to consider all candidate test cases at each iteration, which may result in suboptimal performance in effectiveness and efficiency. For example, in the second iteration, we need to update all the remaining test cases and also perform a random tie-breaking.

If OCP is applied to the example, the first selected test case is t_2, which is the same as the *additional-greedy* strategy. However, in the second iteration, when facing a tie, OCP prefers the fault-revealing test case t_4 as it covers more statements than t_3 in the first iteration. In the third iteration, t_3 is updated first as it covers more statements than t_1 in the last iteration and found to cover one not-yet-covered statement s_2. As no statement is covered by t_1 in the second iteration, we can observe t_1 cannot cover more statements in the next iteration. Thus, t_3 is selected without updating t 1, which leads to fewer calculations. As a result, the test sequence of OCP is $\langle t_2, t_4, t_3, t_1\rangle$ with the APFD value reaching 0.625, which may result in a higher fault detection rate with lower prioritization time. In recent years, the sizes of the regression test suites of modern industrial systems grow at a fast pace, and existing TCP techniques (e.g., the *additional-greedy* strategy and its follow-ups) have become inadequate in efficiency (Luo et al., 2016; Zhou et al., 2022). However, there exist little work to improve the *additional-greedy* strategy efficiency while preserving the high effectiveness.

3. Approach

In this section, we introduce the details of test case prioritization by the partial attention mechanism.

3.1. Partial attention mechanism

Although the *additional-greedy* strategy empirically performs outstandingly in terms of fault detection rate, there is a weakness in the feedback mechanism. As discussed in Section 1, considering all candidate test cases in each prioritization iteration may result in redundant calculations and a high probability of tie-occurring. In fact, only the binary states of the code units (i.e., covered or not covered) are fed back to the next iteration, and some valuable information (e.g., the previous priority values) is discarded. In other words, the candidate test cases are independent of each other before each iteration, and the loss of previous priority values may lead to a decrease performance in the effectiveness and efficiency of TCP techniques. As a result, the priority values of all candidate test cases need to be updated based on huge calculations.

Thus, to address the problem of considering all candidate test cases in each iteration, we attempt to adopt the feedback information from another perspective. Specifically, the priority values in previous iterations are adopted to pay attention to partial candidate test cases. The critical insight is that the number of additionally-covered code units is non-monotonically decreasing, as it cannot cover more code units in the next iteration. Then, the priority values of candidate test cases can be stored in a well-designed structure (i.e., partition in Section 3.2). Meanwhile, the structure can be maintained in the next iteration, such that the more important test cases can be given more attention without additional calculations. As a result, we propose a concept of the partial attention mechanism and apply the concept to state-of-the-art, the *additional-greedy* strategy.

Suppose that test case t_i and t_j covers s_{ik} and s_{jk} not-yet-covered statements ($s_{ik} > s_{jk}$) at kth iteration, respectively. Thus, at the next iteration, t_i is first updated and is found to cover $s_{i(k+1)}$ statements. There may exist two possible situations: (1) $s_{i(k+1)} \leq s_{jk}$: t_j needs to be updated and the number of covered statements is $s_{j(k+1)}$. If $s_{i(k+1)}$ equals $s_{j(k+1)}$, t_i is preferred as it covers more statements than t_j in the ith iteration. Otherwise, the test case covering more statements in jth iteration is selected, which is identical to the *additional-greedy* strategy. (2) $s_{i(k+1)} > s_{jk}$: t_i is selected without updating t_j. Suppose the covered statements of selected test case at kth iteration and t_i are $S = \{s_1, s_2, \ldots, s_n\}$ and $S' = \{s'_1, s'_2, \ldots, s'_m\}$. If $S \cup S' = \emptyset$, we have $s_{jk} = s_{jk}$, otherwise $s_{jk} > s_{jk}$. Thus, we observe that as the selection steps iterate, the number of covered statements should be non-monotonically decreasing ($s_{jk} \geq s_{jl}$). In such case, we can conform $s_{i(k+1)} > s_{j(k+1)}$ from $s_{i(k+1)} > s_{jk}$ and $s_{jk} \geq s_{jl}$ without updating t_j.

In conclusion, before covering all code units, if test case t_i covers more code units than another test case t_j, t_i is more likely to have a higher priority value. Thus, instead of considering both t_i and t_j, the more important one t_i should be updated first. If t_i covers more code units than the theoretical best priority value of t_j (i.e, the priority value at last iteration), t_i will be selected. Otherwise, the remaining t_j is updated and compared with t_i.

Algorithm 1 Pseudocode of OCP

Input: T: $\{t_1, t_2, \cdots, t_n\}$ is a set of unordered test cases with size n; U: $\{u_1, u_2, \cdots, u_m\}$ is a set of code units with size m in the program P
Output: S: a set of prioritized test cases

```
1: S ← ∅
2: Candidates ← ∅
3: priority ← m                                   ▷ the highest priority value
4: for each j (1 ≤ j ≤ m) do
5:     UnitCover[j] ← false
6: end for
7: for each i (1 ≤ i ≤ n) do
8:     Candidates ← Candidates ≻ ⟨t_i, priority⟩
9: end for
10: while |Candidates| > 0 do
11:     maximum ← −1
12:     for each ⟨t_i, temp_priority⟩ ∈ Candidates do
13:         if temp_priority == priority then
14:             temp_num ← 0
15:             for each j (1 ≤ j ≤ m) do
16:                 if Cover[i, j] and not UnitCover[j] then
17:                     temp_num ← temp_num + 1
18:                 end if
19:             end for
20:         end if
21:         if temp_num = priority then
22:             temp_Candidates ≻ ⟨t_i, priority⟩
23:         end if
24:     end for
25:     if priority > 0 then
26:         if |temp_Candidates| > 0 then
27:             ⟨t_k, priority⟩ ← TieSelect(temp_Candidates)
28:             S ← S ≻ ⟨t_k⟩
29:             Candidates ← S \ ⟨t_k, priori⟩
30:             for each j (1 ≤ j ≤ m) do
31:                 if Cover[i, j] and not UnitCover[j] then
32:                     UnitCover[j] ← false
33:                 end if
34:             end for
35:         else
36:             priority ← priority − 1
37:         end if
38:     else
39:         for each j (1 ≤ j ≤ m) do
40:             UnitCover[j] ← false
41:         end for
42:     end if
43: end while
44: return S
```

3.2. Partition ordering based prioritization

In our work, we view the partial attention mechanism as a general concept that can be applied to different prioritization strategies using different coverage criteria. For example, in the *lexicographical-greedy* strategy, the test cases covering the code units fewer covered in the previous iteration should be preferred, as they have a higher probability to cover these code units in the next iteration. As *greedy-based* strategies are the most widely-adopted prioritization strategies (Rothermel et al., 1999; Zhang et al., 2013a), and the *additional-greedy* strategy is considered to be one of the most effective TCP techniques in terms of fault detection capacity (Luo et al., 2016; Lu et al., 2016; Cheng et al., 2021; Luo et al., 2019; Li et al., 2021). We apply the partial attention mechanism to the *additional-greedy* strategy and implement a simple greedy strategy to instantiate the function based on partition ordering. Generally speaking, all candidate test cases are grouped into different partitions based on their previous priority values and higher partitions are updated preferentially.

Given a program under test $U = \{u_1, u_2, \ldots, u_m\}$ containing m code units, and a test suite $T = \{t_1, t_2, \ldots, t_n\}$ containing n test cases, Algorithm 1 describes the pseudocode of our proposed method. At each iteration, all the candidate test cases are sorted based on the priority values (i.e., the number of additionally-covered code units) and the ones with the same priority value are adjacent to each other. We then group the candidate test cases according to their priority values into p partitions, indexed from left to right by $1, 2, \ldots, p$, such that all the candidate test cases within a partition have the same priority value. Based on these partitions, we form the vector $v = [v_1, v_2, \ldots, v_p]$ where v_i indicates priority value in the ith partition. In the next iteration, we then update the candidate test cases in partition p and group them into new partitions according to their updated priority values (i.e., the number of additionally-covered code units in the next iteration). If there exists a test case from the partition p that falls into a new partition j ($v_j > v_{p-1}$), the test case is selected. Because the test cases from the partition i ($i \leq p-1$) cannot be updated into a partition with a higher index j ($v_j > v_{p-1}$). Otherwise, we update the partition from right to left according to the value v_i until a test case is selected. In the worst case, we may update all the partitions if no test case is selected, which is identical to the *additional-greedy* strategy.

Specifically, we use a Boolean array $Cover[i, j]$ ($1 \leq i \leq n,\ 1 \leq j \leq m$) to identify whether the test case t_i covers the code unit u_j or not. We use another Boolean array $UnitCover[j]$ ($1 \leq j \leq m$) to denote whether the code unit u_j has been covered by the already selected test cases or not. We set the value of $UnitCover[j]$ ($1 \leq j \leq m$) to be *false*. Similarly, we use a variable *priority* to denote the largest priority value for all candidate test cases. Meanwhile, we set the value of *priority* to be m. Besides, we use a set *Candidates* to denote all remaining test cases and their corresponding priority values. Initially, we add the whole test suite to *Candidates* with the default priority value m (i.e., the number of code units).

In Algorithm 1, lines 1–9 perform initialization, and lines 10–43 prioritize the test cases. In the main loop from line 10 to line 43, each iteration attempts to find a test case with the given priority value and add it to the prioritized test set S. In particular, lines 12–24 calculate the sum of units covered by the test cases with the highest previous priority value and add the ones that maintain the advantage into the candidate test set *temp_Candidates*. Before choosing the next test case, our approach examines whether or not there are any code units that are not covered by the test cases in S. If all code units have been covered, the remaining candidate test cases are prioritized by restarting the previous process (lines 39–41). Otherwise, we select the test case in *temp_Candidate* with highest previous priority values as the next one and update the cover status of all code units (lines 26–37). If no test case is selected, we further update the second partition, and so on (line 36). This process is repeated until all test cases in *Candidates* have been added to S. It is worth noting that although a partial attention mechanism is adopted in our approach, there is also a small possibility that a tie occurs, e.g., more than one test case in *temp_Candidate* with highest previous priority values. In such a case, similar to the *additional-greedy* strategy, our approach performs a random tie-breaking.

4. Experiment

In this section, we present our empirical study in detail, including the research questions, some variables, subject programs and the experimental setup.

4.1. Research questions

The empirical study is conducted to answer the following research questions.

RQ1 How does the effectiveness of OCP compare with state-of-the-art techniques, in terms of fault detection rate?

RQ2 How does the granularity of code coverage impact the comparative effectiveness of OCP?

Table 1
Studied TCP techniques.

Mnemonic	Description	Category	Reference
TCP_{tot}	*total-greedy* test prioritization	Greedy-based	Rothermel et al. (1999)
TCP_{add}	*additional-greedy* test prioritization	Greedy-based	Rothermel et al. (1999)
TCP_{unif}	*unified-greedy* test prioritization	Greedy-based	Zhang et al. (2013a), Hao et al. (2014)
TCP_{lexi}	*lexicographical-greedy* test prioritization	Greedy-based	Eghbali and Tahvildari (2016)
TCP_{art}	*art-based* test prioritization	Similarity-based	Jiang et al. (2009)
TCP_{search}	*search-based* test prioritization	Search-based	Li et al. (2007)
TCP_{ocp}	our proposed technique OCP	Greedy-based	This study

RQ3 How does the granularity of test cases impact the comparative effectiveness of OCP?

RQ4 How does the efficiency of OCP compare with state-of-the-art techniques, in terms of execution time?

4.2. Independent variables

4.2.1. Prioritization techniques

Although the proposed generic strategies can work with any coverage criteria, we implement OCP based on basic structural coverage criteria due to their popularity (Lu et al., 2016; Zhang et al., 2013a; Lou et al., 2019; Jiang et al., 2009; Luo et al., 2019). We select the six state-of-the-art coverage-based TCP techniques that have been widely used in previous TCP studies (Luo et al., 2016; Lu et al., 2016; Luo et al., 2019): *total-greedy* (Rothermel et al., 1999), *additional-greedy* (Rothermel et al., 1999), *unified-greedy* (Zhang et al., 2013a; Hao et al., 2014), *lexicographical-greedy* (Eghbali and Tahvildari, 2016), *art-based* (Jiang et al., 2009), and *search-based* (Li et al., 2007).

The *total-greedy* technique prioritizes test cases based on the descending number of code units covered by those test cases. The *additional-greedy* technique chooses each test case from the candidate test set such that it covers the largest number of code units not yet covered by the previously selected test cases. Similarly, the *unified-greedy* technique selects the test case with the highest sum of the probabilities that units covered by the test case contain undetected faults, while the *lexicographical-greedy* technique selects the test case with the maximum coverage of one-time-covered code units. Likewise, if a tie occurs, code units that are covered twice are considered, and so on. The *art-based* technique selects each test case from a random candidate test set such that it has the greatest maximum distance from the already selected test cases. Finally, the *search-based* technique considers all permutations as candidate solutions, and uses a meta-heuristic search algorithm to guide the search for a better test execution order (Li et al., 2007). Depending on prioritization strategies, these TCP techniques are grouped into three categories and the details are presented in Table 1.

For the *total-greedy*, *additional-greedy*, *art-based* and *search-based* techniques, we directly use the source code released by existing work (Chen et al., 2018; Huang et al., 2020). Meanwhile, the implementation of the *unified-greedy* technique is not publicly available and the *lexicographical-greedy* technique is implemented in other language (i.e., Matlab). Thus, we implement the *unified-greedy* and *lexicographical-greedy* techniques according to their paper carefully. For the *unified-greedy* technique, we select the basic model (i.e., Algorithm 1 in Zhang et al., 2013a) in our work, as the extended model requires multiple coverage of code units by given test cases, which is beyond the scope of our work. We also select the default configuration (i.e., Algorithm 2 in Eghbali and Tahvildari, 2016) for the *lexicographical-greedy* technique, as it achieve a great balance between fault detection rate and prioritization time.

4.2.2. Code coverage granularity

In traditional TCP studies (Luo et al., 2016; Zhang et al., 2013a), the coverage granularity is generally considered to be a constituent part of the prioritization techniques. To enable sufficient evaluations, we attempt to investigate generic prioritization strategies with various structural coverage criteria (i.e., the statement, branch, and method coverage granularities).

4.2.3. Test case granularity

For the subject programs written in Java, we consider the test case granularity as an additional factor in the prioritization techniques. Test case granularity is at either the test-class or the test-method granularity. Specifically, given a Java program, a JUnit test class file refers to a test case at test-class granularity, while each test method in the file refers to a test case at test-method granularity. In other words, a test case at the test-class granularity generally involves a number of test cases at the test-method granularity. For C subject programs, the actual program inputs are the test cases.

4.3. Dependent variables

To evaluate the effectiveness of different TCP techniques, we adopt the widely-used APFD (*average percentage faults detected*) as the evaluation metric for fault detection rate (Rothermel et al., 1999). Given a test suite T, with n test cases, P' is a permutation of T. Then the APFD value for P' is defined by the following formula:

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

where, m denotes the total number of detected faults and TF_i denotes the position of first test case that reveals the fault i.

4.4. Subject programs, test suites and faults

To enable sufficient evaluations, we conduct our study on 19 versions of four Java programs (i.e., eight versions of *ant*, five versions of *jmeter*, three versions of *jtopas*, and three versions of *xmlsec*), which are obtained from the *Software-artifact Infrastructure Repository* (SIR) (Do et al., 2005; Anon., 0000b). Meanwhile, 30 versions of five real-life Unix utility programs written in C language (six versions of *flex*, *grep*, *gzip*, *make* and *sed*) are also adopted, which are downloaded from the *GNU FTP server* (Anon., 0000c). Both the Java and C programs have been widely utilized as benchmarks to evaluate TCP techniques (Zhang et al., 2013a; Eghbali and Tahvildari, 2016; Jiang et al., 2009; Henard et al., 2016).

Table 3 lists all the subject programs and the detailed statistical information. In Table 3, for each program, columns 3 to 6 summarize the version, size, number of branches, number of methods, respectively.

Each version of the Java programs has a JUnit test suite that is developed during the program's development. These test suites have two levels of test-case granularity: the test-class and the test-method. The numbers of JUnit test cases are shown in the **#Test** column: The data is presented as x (y), where x is the

Table 2
Statistics on Mutation Operators.

Language	Operators	Descriptions
Java	CB	Conditionals boundary
	IC	Increments
	IN	Invert negatives
	MA	Math
	NC	Negate conditionals
	VM	Void method calls
	ER	Empty returns
	FR	False returns
	TR	True returns
	NR	Null returns
	PR	Primitive returns
C	SD	Statement deletion
	UI	Unary insertion
	CR	Constant replacement
	AR	Arithmetic operator replacement
	LR	Logical operator replacement
	BR	Bitwise logical operator replacement
	RR	Relational operator replacement

number of test cases at test-method granularity, and y is the number of test cases at test-class granularity. The test suites for the C programs are collected from the SIR (Do et al., 2005; Anon., 0000b). The number of tests cases in each suite is also shown in the **#Test** column of Table 3.

The faults contained in each version of the programs are produced based on mutation analysis (Papadakis et al., 2019; Zhang et al., 2019). Although some seeded faults of programs are available from SIR, previous research has confirmed that the seeded ones are easily detected and small in size. Meanwhile, mutation faults have previously been identified as suitable for simulating real program faults (Andrews et al., 2005; Belli et al., 2006; Do and Rothermel, 2005; Just et al., 2014; Belli et al., 2016) and have been widely applied to various TCP evaluations (Rothermel et al., 1999; Luo et al., 2016; Lu et al., 2016; Zhang et al., 2013a; Elbaum et al., 2000; Luo et al., 2019; Henard et al., 2016). Thus, for both C and Java programs, mutation faults are introduced to evaluate the performance of the different techniques. The details of these operators are presented in Table 2. For C programs, we obtain the mutants from previous TCP studies (Henard et al., 2016; Andrews et al., 2006), which are produced using seven mutation operators. For Java programs, we use eleven mutation operators from the "NEW_DEFAULTS" group of the PIT mutation tool (Coles et al., 2016) to generate mutants. Specifically, we generate mutants (i.e., faulty versions) by seeding all mutation operators into the subject programs automatically. Then we run the available test suite against each mutant. The mutant is killed if there exist any test that produces inconsistent test outcomes between the original and faulty version, otherwise the mutant is lived. We select all killed mutants to evaluate the fault detection rate of TCP techniques.

Meanwhile, according to existing studies (Huang et al., 2020; Henard et al., 2016), the subsuming mutants identification (SMI) technique (Papadakis et al., 2016) is adopted to remove the duplicate and subsuming mutants from all killed mutants. The number of subsuming mutants used in our experiment is presented in the **#Subsuming_Mutant** column. It is worth noting that the subsuming faults are classified as test-class level and test-method level for the Java programs.

4.5. Framework

Fig. 2 presents the overall experimental framework of the proposed technique. (1) We collect the coverage information for the Java program using the `FaultTracer` tool (Zhang et al., 2012, 2013b), which uses on-the-fly bytecode instrumentation without any modification of the target program based on the ASM bytecode manipulation and analysis framework (Anon., 0000d). For C program, there are six versions of each program P: P_{V0}, P_{V1}, P_{V2}, P_{V3}, P_{V4}, and P_{V5}. Version P_{V0} is compiled using `gcc` 5.4.0 (Anon., 0000e), and then the coverage information is obtained using the `gcov` tool (Anon., 0000f). (2) After collecting the code coverage information, we implement all TCP techniques in Java, and apply them to each program version under study. Specifically, OCP first divides test cases into different partitions and updates test cases in the highest partition. If there exist an updated test case satisfies the selection criteria, the test case will be added to the prioritized test sequence. Because the approaches contain randomness, each execution is repeated 1000 times independently. This results in, for each testing scenario, 1000 test sequences for each TCP technique. (3) To evaluate the fault detection rate, we construct the faulty programs by mutation faults. Specifically, we generate mutants by seeding all mutation operators (presented in Table 2) and consider each mutant as a faulty program with only one mutation fault. We then execute all test cases against each faulty program and remove the mutants that any test case cannot kill. (4) Besides, we calculate the APFD values and prioritization time for all test sequences based on the record information (e.g., the mutation detected results and time cost of each TCP technique) (5) To further test whether there is a statistically significant difference between OCP and other TCP techniques, we perform the unpaired two-tailed Wilcoxon–Mann–Whitney test, at a significance level of 5%, following previously reported guidelines for inferential statistical analysis involving randomized algorithms (Arcuri and Briand, 2014; Gligoric et al., 2015b). To identify which technique is better, we also calculate the effect size, measured by the non-parametric Vargha and Delaney effect size measure (Vargha and Delaney, 2000), $\hat{A}_{12}$, where $\hat{A}_{12}(X, Y)$ gives probability that the technique X is better than technique Y. The statistical analyses are performed using R language (Anon., 0000g).

4.6. Experimental setup

The experiments are conducted on a Linux 5.15.0-25-generic cloud server with eight virtual cores of Intel(R) Xeon(R) Silver 4116 CPU (2.10 GHz) and 32 GBs of virtual RAM.

5. Results and analysis

This section presents the experimental results to answer the research questions. We investigate the effectiveness of OCP to answer RQ1, and perform impact analysis to investigate the influences caused by the code coverage granularity to answer RQ2. Besides, we also perform analysis to investigate the influences caused by and test case granularity on OCP to answer RQ3. Finally, we analyze the time cost of OCP to answer RQ4.

To answer RQ1 to RQ3, Figs. 3 to 6 present box plots of the distribution of the APFD values achieved over 1000 independent runs. Each box plot shows the mean (square in the box), median (line in the box), and upper and lower quartiles (25th and 75th percentile) of the APFD values for all the TCP techniques. Statistical analyses are also provided in Tables 4 to 5 for each pairwise APFD comparison between OCP and the other TCP techniques. For ease of illustration, we denote the mentioned TCP techniques as TCP_{tot}, TCP_{add}, TCP_{lexi}, TCP_{unif}, TCP_{art}, TCP_{search} and TCP_{ocp}, respectively. For example, for a comparison between two methods TCP_{ocp} and M, where $M \in \{TCP_{tot}, TCP_{add}, TCP_{lexi}, TCP_{unif}, TCP_{art}, TCP_{search}\}$, the symbol ✔ means that TCP_{ocp} is better (p-value is less than 0.05, and the effect size $\hat{A}_{12}(TCP_{ocp}, M)$ is greater than 0.50); the symbol ✖ means that M is better (the p-value is less than 0.05, and $\hat{A}_{12}(TCP_{ocp}, M)$ is less than 0.50); and the

Table 3
Subject program details.

Language	Program	Version	KLoC	#Branch	#Method	#Class	#Test_Case		#Mutant		#Subsuming_Mutant	
							#T_Class	#T_Method	#All	#Detected	#SM_Class	#SM_Method
Java	*ant_v1*	v1_9	25.80	5240	2511	228	34 (34)	137 (135)	6498	1332	59	32
	ant_v2	1.4	39.70	8797	3836	342	52 (52)	219 (214)	11,027	2677	90	47
	ant_v3	1.4.1	39.80	8831	3845	342	52 (52)	219 (213)	11,142	2661	92	47
	ant_v4	1.5	61.90	11,743	5684	532	102 (100)	521 (503)	14,834	6585	192	88
	ant_v5	1.5.2	63.50	141,76	5802	536	105 (103)	557 (543)	17,826	6230	211	91
	ant_v6	1.5.3	63.60	141,68	5808	536	105 (102)	559 (537)	17,808	6255	92	91
	ant_v7	1.6 beta	80.40	17,164	7520	649	149 (149)	877 (866)	22,171	9094	284	119
	ant_v8	1.6 beta2	80.40	17,746	7524	650	149 (149)	879 (867)	22,138	9068	226	119
	jmeter_v1	v1_7_3	33.70	3815	2919	334	26 (21)	78 (61)	8850	573	38	20
	jmeter_v2	v1_8	33.10	3799	2838	319	29 (24)	80 (74)	8777	867	37	22
	jmeter_v3	v1_8_1	37.30	4351	3445	373	33 (27)	78 (77)	9730	1667	47	25
	jmeter_v4	v1_9_RC1	38.40	4484	3536	380	33 (27)	78 (77)	10,187	1703	47	25
	jmeter_v5	v1_9_RC2	41.10	4888	3613	389	37 (30)	97 (83)	10,459	1651	53	29
	jtopas_v1	0.4	1.89	519	284	19	10 (10)	126 (126)	704	399	29	9
	jtopas_v2	0.5.1	2.03	583	302	21	11 (11)	128 (128)	774	446	34	10
	jtopas_v3	0.6	5.36	1491	748	50	18 (16)	209 (207)	1906	1024	57	16
	xmlsec_v1	v1_0_4	18.30	3534	1627	179	15 (15)	92 (91)	5501	1198	32	12
	xmlsec_v2	v1_0_5D2	19.00	3789	1629	180	15 (15)	94 (94)	5725	1204	33	12
	xmlsec_v3	v1_0_71	16.90	3156	1398	145	13 (13)	84 (84)	3833	1070	27	10
C	*flex_v0*	2.4.3	8.96	2005	138	–		500	–	–		–
	flex_v1	2.4.7	9.47	2011	147	–		500	13,873	6177		32
	flex_v2	2.5.1	12.23	2656	162	–		500	14,822	6396		32
	flex_v3	2.5.2	12.25	2666	162	–		500	775	420		20
	flex_v4	2.5.3	12.38	2678	162	–		500	14,906	6417		33
	flex_v5	2.5.4	12.37	2680	162	–		500	14,922	6418		32
	grep_v0	2.0	8.16	3420	119	–		144	–	–		–
	grep_v1	2.2	11.99	3511	104	–		144	23,896	3229		56
	grep_v2	2.3	12.72	3631	109	–		144	24,518	3319		58
	grep_v3	2.4	12.83	3709	113	–		144	17,656	3156		54
	grep_v4	2.5	20.84	2531	102	–		144	17,738	3445		58
	grep_v5	2.7	58.34	2980	109	–		144	17,108	3492		59
	gzip_v0	1.0.7	4.32	1468	81	–		156	–	–		–
	gzip_v1	1.1.2	4.52	1490	81	–		156	7429	639		8
	gzip_v2	1.2.2	5.05	1752	98	–		156	7599	659		8
	gzip_v3	1.2.3	5.06	1610	93	–		156	7678	547		7
	gzip_v4	1.2.4	5.18	1663	93	–		156	7838	548		7
	gzip_v5	1.3	5.68	1733	97	–		156	8809	210		7
	make_v0	3.75	17.46	4397	181	–		111	–	–		–
	make_v1	3.76.1	18.57	4585	181	–		111	36,262	5800		37
	make_v2	3.77	19.66	4784	190	–		111	38,183	5965		29
	make_v3	3.78.1	20.46	4845	216	–		111	42,281	6244		28
	make_v4	3.79	23.13	5413	239	–		111	48,546	6958		29
	make_v5	3.80	23.40	5032	268	–		111	47,310	7049		28
	sed_v0	3.01	7.79	676	66	–		324	–	–		–
	sed_v1	3.02	7.79	712	65	–		324	2506	1009		16
	sed_v2	4.0.6	18.55	1011	65	–		324	5947	1048		18
	sed_v3	4.0.8	18.69	1017	66	–		324	5970	450		18
	sed_v4	4.1.1	21.74	1141	70	–		324	6578	470		19
	sed_v5	4.2	26.47	1412	98	–		324	7761	628		22

symbol ❍ means that there is no statistically significant difference between them (i.e., the *p*-value is greater than 0.05).

To answer RQ4, Table 6 provides comparisons of the execution times for the different TCP techniques.

5.1. RQ1: Effectiveness of OCP

In this section, we evaluate the effectiveness of different TCP techniques by fault detection rate. We provide the APFD results for OCP with different code coverage criteria and test case granularities. Figs. 3 to 5 show the APFD results for the C programs, the Java programs at the test-method granularity and the test-method granularity, respectively. Each sub-figure in these figures has the seven TCP techniques across the *x*-axis, and corresponding to the APFD values on the *y*-axis. Table 4 presents the corresponding statistical comparisons. Each row denotes the statistical results for the corresponding program under different coverage criteria. Column "C Programs", "Java-M Programs", "Java-C Programs" and"All Programs" are calculated based on all APFD values for C programs, Java programs at the test-method granularity, Java programs at the test-class granularity and all programs.

5.1.1. C subject programs

Based the results on Fig. 3 and Table 4, we make the following observations:

When comparing TCP_{ocp} with the *greedy-based* strategies, our proposed TCP_{ocp} approach has much better performance than TCP_{tot}, TCP_{add}, TCP_{unify} and TCP_{lexi} for all programs and code coverage granularities, except for *make* with branch coverage (for which TCP_{ocp} has very similar, or better performance). The maximum difference in mean and median APFD values between TCP_{ocp} and TCP_{tot} is more than 40%, while between TCP_{ocp} and TCP_{add}, it is about 10%.

Our proposed technique TCP_{ocp} has similar or better APFD performance than TCP_{art} and TCP_{search} for some subject programs

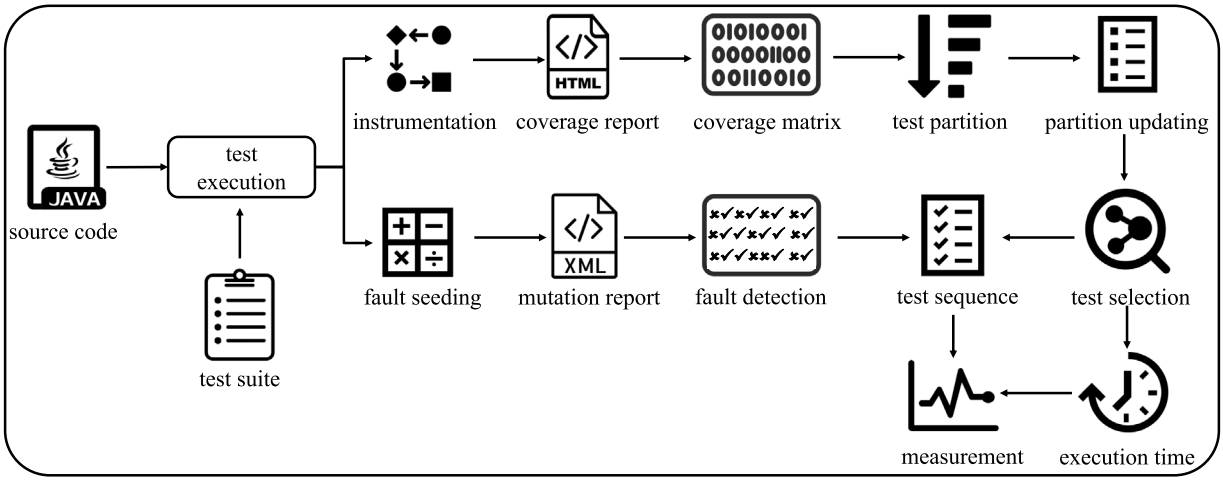

Fig. 2. OCP's framework.

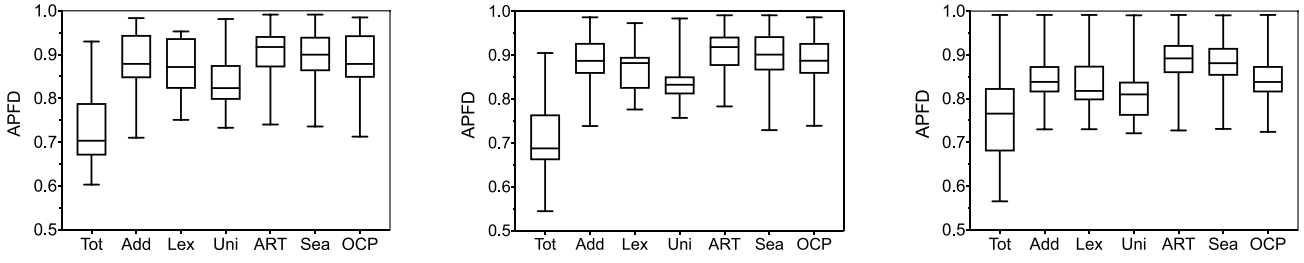

Fig. 3. APFD results for C programs.

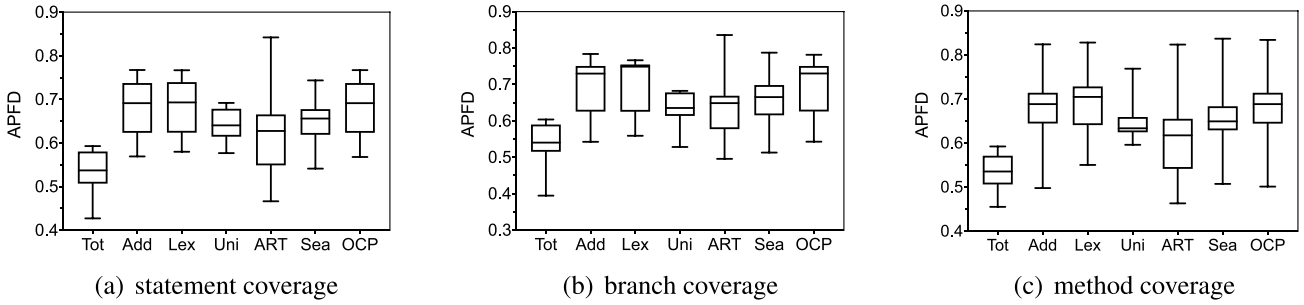

Fig. 4. APFD results for Java programs at test-method granularity.

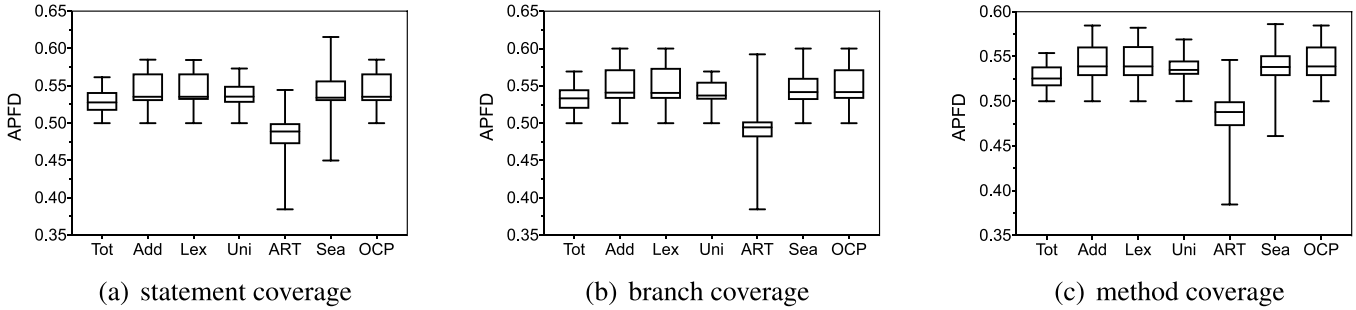

Fig. 5. APFD results for Java programs at test-class granularity.

(e.g., *flex* and *gzip*), with all code coverage granularities, but has slightly worse performance for some others (e.g., *make* and *sed*). However, the difference in mean and median APFD values between TCP_{ocp} and TCP_{art} or TCP_{search} is less than 5%.

Furthermore, the statistical results support the box plot observations. All *p*-values for the comparisons between TCP_{ocp} and the *greedy-based* strategies (i.e., TCP_{tot} or TCP_{add}) are less than 0.05 (except for *make* with branch coverage), indicating that their APFD scores are significantly different. The $\hat{A}_{12}$ values are also much greater than 0.50, ranging from 0.51 to 1.00. However, although all *p*-values between TCP_{ocp} and TCP_{art} or TCP_{search} are also less than 0.05, their $\hat{A}_{12}$ values are much greater than 0.50 in some cases, but much less than 0.50 in others. Nevertheless, considering all the C programs, not only does TCP_{ocp} have significantly different APFD values to the other six TCP techniques, but it also has better performances overall (except for TCP_{art} and TCP_{search}).

5.1.2. Java programs at test-method granularity

Based on Fig. 5 and Table 4, we have the following observations:

Compared with the *greedy-based* strategies, TCP_{ocp} performs much better than TCP_{tot}, regardless of subject program and code coverage granularity, with the maximum mean and median APFD differences reaching about 12%. TCP_{ocp} also has very similar performance to TCP_{add}, with the mean and median APFD differences approximately equal to 1%. However, none of the other two techniques (TCP_{lexi} and TCP_{unify}) is either always better or always worse than TCP_{ocp}, with TCP_{ocp} sometimes performing better for some programs, and sometimes worse.

TCP_{ocp} also performs better than TCP_{art} and TCP_{search} at most cases (except *jtopas* at statement and branch coverage) There is a statistically significant difference between TCP_{ocp} and TCP_{art}, which supports the above observations.

Furthermore, the statistical results support the box plot observations. Considering all Java programs, TCP_{ocp} performs better than TCP_{tot}, TCP_{art} and TCP_{search}, as most *p*-values are less than 0.05, and the relevant effect size $\hat{A}_{12}$ ranges from 0.58 to 0.98. However, OCP has very a similar (or slightly worse) performance to TCP_{add}, with $\hat{A}_{12}$ values of either 0.48 or 0.50.

5.1.3. Java programs at test-class granularity

Based on Fig. 4 and Table 4, we have the following observations:

OCP achieves higher mean and median APFD values than TCP_{tot} for most cases, except *jmeter*. OCP has a very similar performance to TCP_{add}, with their mean and median *APFD* differences at around 1%. OCP has a competitive performance with TCP_{unify} and TCP_{lexi} for all programs with different code coverage granularities. OCP achieves much higher mean and median APFD values than TCP_{art} for most cases, for all programs with all code coverage granularities, with the maximum differences reaching approximately 10%. Other than for a few cases (e.g., *jtopas*), OCP usually has better performance than TCP_{search}.

Furthermore, the statistical analysis supports the above box plots observations. Considering all Java programs together, OCP performs better than TCP_{tot}, TCP_{unify}, TCP_{search}, TCP_{art}, and TCP_{search} on the whole. Most *p*-values are less than 0.05, indicating that their differences are significant; and the effect size $\hat{A}_{12}$ values range up to 1.00, which means that TCP_{ocp} is better than the other five TCP techniques. Finally, while the *p*-values for comparisons between TCP_{ocp} and TCP_{add} are less than 0.05 (which means that the differences are insignificant), the $\hat{A}_{12}$ values range from 0.49 to 0.51, indicating that they are very similar.

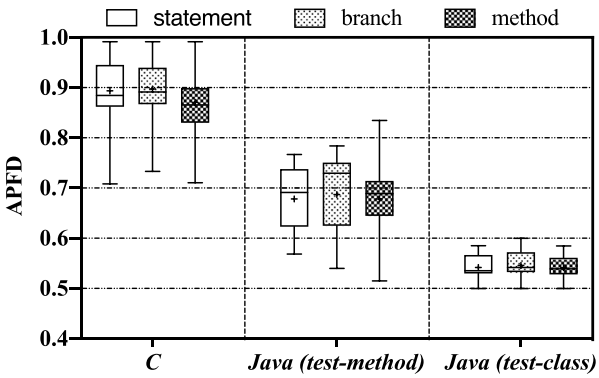

Fig. 6. Effectiveness: APFD results with different code coverage and test case granularities for all programs.

Answer to RQ1: Overall, our analysis on the fault detection effectiveness that (1) For C programs, OCP has significantly better performance than all *greedy-based* strategies and maintaining the comparable performance with TCP_{art} and TCP_{sea}. (2) For Java programs at test-method granularity, OCP has better performance than TCP_{tot}, TCP_{art} and TCP_{sea}, while has similar performance with TCP_{add}, TCP_{unify} and TCP_{lexi}. (3) For Java programs at test-class granularity, OCP has better or similar performance with TCP_{add}, TCP_{art} and TCP_{sea} for all programs, while has comparable performance with TCP_{tot}, TCP_{unify} and TCP_{lexi}.

5.2. RQ2: Impact of code coverage granularity

In our study, three basic structural coverage criteria (i.e., statement, branch and method) are adopted to evaluate the performance of proposed TCP techniques. Previous empirical studies have demonstrated that different code coverage granularities may affect the APFD results (Luo et al., 2016; Huang et al., 2020). Thus, in this section, we examine how the selection of code coverage granularity influences the effectiveness of OCP.

Fig. 6 presents the APFD results of OCP for the three types of code coverage, according to the subject programs' language or test case granularity. The language or test case granularity is shown on the *x*-axis and the APFD scores on the left *y*-axis. It can be observed that for C programs, statement and branch coverage are very considerable, and are more effective than method coverage. However, for Java programs, they have similar performance.

Table 5 presents a comparison of the mean and median APFD values, and also shows the *p*-values/effect size $\hat{A}_{12}$ for the different code coverage granularity comparisons. Column "C", "Java (test-method)", "Java (test -class)" and "All" is calculated based on all APFD values for C programs, Java programs at the test-method granularity, Java programs at the test-class granularity and all programs. It can be observed that the APFD values are similar, with the maximum mean and median value differences being less than 3%, and less than 8%, respectively. According to the statistical comparisons, there is no single best code coverage type for OCP, with each type sometimes achieving the best results.

Table 4
Statistical **effectiveness** comparisons of APFD for **all programs**.

Program name		Statement coverage						Branch coverage						Method coverage					
		TCP_{tot}	TCP_{add}	TCP_{unify}	TCP_{lexi}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{unify}	TCP_{lexi}	TCP_{art}	TCP_{search}	TCP_{tot}	TCP_{add}	TCP_{unify}	TCP_{lexi}	TCP_{art}	TCP_{search}
gnu_flex	–	✔ (1.00)	✔ (0.52)	✔ (0.70)	✔ (0.87)	✔ (0.86)	✔ (0.73)	✔ (1.00)	✔ (0.66)	✖ (0.22)	✔ (0.90)	✔ (0.72)	✔ (0.57)	✔ (1.00)	✔ (0.70)	✔ (0.71)	✔ (0.85)	✖ (0.28)	✖ (0.47)
gnu_make	–	✔ (0.85)	✔ (0.61)	✔ (0.62)	✔ (0.82)	✖ (0.30)	✖ (0.29)	✔ (1.00)	❍ (0.51)	✔ (0.77)	✔ (0.87)	✖ (0.28)	✖ (0.42)	✔ (0.58)	✔ (0.68)	✔ (0.68)	✔ (0.58)	✖ (0.22)	✖ (0.24)
gnu_grep	–	✔ (1.00)	✔ (0.57)	✔ (0.79)	✔ (1.00)	✖ (0.48)	✔ (0.64)	✔ (1.00)	✔ (0.54)	✔ (0.67)	✔ (1.00)	❍ (0.50)	✔ (0.66)	✔ (1.00)	✔ (0.71)	✔ (0.99)	✔ (1.00)	✖ (0.21)	✖ (0.25)
gnu_gzip	–	✔ (0.80)	✔ (0.66)	✔ (0.78)	✔ (0.58)	✖ (0.43)	✖ (0.36)	✔ (0.84)	✔ (0.70)	✔ (0.83)	❍ (0.49)	✖ (0.37)	✖ (0.32)	✔ (0.54)	✔ (0.53)	✔ (0.54)	✔ (0.53)	✖ (0.47)	✖ (0.47)
gnu_sed	–	✔ (1.00)	✔ (0.54)	✔ (0.64)	✔ (1.00)	✖ (0.15)	✖ (0.37)	✔ (1.00)	✔ (0.55)	✔ (0.64)	✔ (1.00)	✖ (0.14)	✖ (0.37)	✔ (1.00)	✔ (0.90)	✔ (0.97)	✔ (0.97)	✖ (0.22)	✖ (0.29)
C Programs		✔ (0.91)	✔ (0.54)	✔ (0.60)	✔ (0.74)	✖ (0.44)	✖ (0.47)	✔ (0.93)	✔ (0.55)	✔ (0.60)	✔ (0.77)	✖ (0.43)	✖ (0.47)	✔ (0.81)	✔ (0.61)	✔ (0.66)	✔ (0.74)	✖ (0.37)	✖ (0.41)
ant	**method**	✔ (1.00)	❍ (0.50)	✖ (0.42)	✔ (1.00)	✔ (1.00)	✔ (1.00)	✔ (1.00)	❍ (0.50)	✖ (0.40)	✔ (1.00)	✔ (1.00)	✔ (1.00)	✔ (1.00)	❍ (0.50)	✖ (0.22)	✔ (0.98)	✔ (0.96)	✔ (0.95)
jtopas	**method**	✔ (1.00)	❍ (0.50)	✖ (0.36)	✖ (0.39)	✖ (0.00)	✖ (0.44)	✔ (1.00)	❍ (0.50)	✖ (0.38)	✔ (1.00)	✖ (0.00)	✖ (0.44)	✔ (1.00)	❍ (0.50)	✖ (0.03)	✔ (0.66)	✔ (0.56)	❍ (0.50)
jmeter	**method**	✔ (1.00)	❍ (0.50)	❍ (0.50)	✖ (0.38)	✔ (1.00)	✔ (0.62)	✔ (1.00)	✖ (0.48)	✔ (0.61)	❍ (0.51)	✔ (1.00)	✔ (0.69)	✔ (0.86)	❍ (0.50)	✖ (0.47)	✖ (0.45)	✔ (0.98)	✔ (0.58)
xmlsec	**method**	✔ (1.00)	❍ (0.50)	✖ (0.39)	✔ (1.00)	✔ (1.00)	❍ (0.49)	✔ (1.00)	❍ (0.49)	✖ (0.13)	✔ (1.00)	✔ (1.00)	✔ (0.77)	✔ (1.00)	❍ (0.50)	✖ (0.36)	✔ (0.99)	✔ (1.00)	✔ (0.55)
Java-M Programs		✔ (0.98)	❍ (0.50)	✖ (0.48)	✔ (0.67)	✔ (0.70)	✔ (0.65)	✔ (0.94)	❍ (0.50)	✖ (0.45)	✔ (0.70)	✔ (0.64)	✔ (0.66)	✔ (0.98)	❍ (0.50)	✖ (0.39)	✔ (0.71)	✔ (0.77)	✔ (0.64)
ant	**class**	✔ (0.83)	❍ (0.50)	✔ (0.52)	✔ (0.70)	✔ (0.99)	✔ (0.60)	✔ (0.89)	❍ (0.50)	✔ (0.51)	✔ (0.69)	✔ (1.00)	✔ (0.54)	✔ (0.94)	❍ (0.50)	✖ (0.48)	✔ (0.71)	✔ (1.00)	✔ (0.57)
jtopas	**class**	❍ (0.50)	❍ (0.50)	❍ (0.50)	❍ (0.50)	✔ (0.93)	❍ (0.50)	❍ (0.50)	❍ (0.50)	❍ (0.50)	❍ (0.50)	✔ (0.91)	❍ (0.50)	✖ (0.33)	❍ (0.50)	❍ (0.50)	✖ (0.33)	✔ (0.94)	❍ (0.50)
jmeter	**class**	✖ (0.36)	❍ (0.50)	✖ (0.46)	✖ (0.36)	✔ (1.00)	❍ (0.50)	✖ (0.40)	✖ (0.47)	✔ (0.52)	✖ (0.40)	✔ (1.00)	✔ (0.52)	✖ (0.31)	❍ (0.50)	✖ (0.48)	✖ (0.32)	✔ (1.00)	❍ (0.49)
xmlsec	**class**	✔ (1.00)	❍ (0.50)	❍ (0.50)	✔ (1.00)	✔ (1.00)	❍ (0.50)	✔ (0.97)	✖ (0.48)	✖ (0.40)	✔ (0.96)	✔ (1.00)	❍ (0.50)	✔ (1.00)	❍ (0.51)	✔ (0.60)	✔ (1.00)	✔ (1.00)	✔ (0.58)
Java-C Programs		✔ (0.63)	❍ (0.50)	❍ (0.50)	✔ (0.55)	✔ (0.98)	✔ (0.53)	✔ (0.65)	❍ (0.50)	❍ (0.50)	✔ (0.56)	✔ (0.97)	✔ (0.51)	✔ (0.67)	❍ (0.50)	❍ (0.50)	✔ (0.58)	✔ (0.98)	✔ (0.52)
All Programs		✔ (0.71)	✔ (0.51)	✔ (0.51)	✔ (0.56)	✔ (0.57)	✔ (0.51)	✔ (0.73)	✔ (0.51)	✔ (0.51)	✔ (0.57)	✔ (0.55)	✔ (0.51)	✔ (0.69)	✔ (0.52)	✔ (0.51)	✔ (0.57)	✔ (0.57)	❍ (0.50)

Table 5
Statistical **effectiveness** comparisons of APFD between different coverage granularities for OCP.

Metric	Language	Mean			Median			Comparison		
		Statement	Branch	Method	Statement	Branch	Method	Statement *vs* Branch	Statement *vs* Method	Branch *vs* Method
APFD	C	0.89	0.90	0.87	0.88	0.89	0.87	1.65E−15/0.48	0/0.62	0/0.65
	Java (test-method)	0.68	0.69	0.68	0.69	0.73	0.69	7.13E−97/0.43	2.04E−3/0.51	1.39E−137/0.58
	Java (test-class)	0.54	0.55	0.54	0.54	0.54	0.54	4.34E−6/0.43	4.34E−06/0.49	1.67E−94/0.56
	All	0.72	0.73	0.71	0.72	0.75	0.70	1.8E−24/0.48	6.71E−34/0.52	2.2E−113/0.54

Nevertheless, branch coverage appears slightly more effective than statement and method coverage for OCP.

Answer to RQ2: Overall, our analysis on the code coverage granularity reveals that the code coverage granularity may only provide a small impact on OCP testing effectiveness, with branch coverage possibly slightly outperforming statement and method coverage.

5.3. RQ3: Impact of test case granularity

In our study, the Java programs have two granularities of test cases (i.e., the test-class and test-method). Following to previous studies (Zhang et al., 2013a; Huang et al., 2020), we also consider the test case granularity as a factor in the evaluation. Thus, in this section, we also investigate how the test case granularity influence the effectiveness of OCP.

The comparisons are presented in Fig. 6. OCP usually has significantly lower average APFD values for prioritizing test cases at the test-class granularity than at the test-method granularity.

Table 5 presents the statistical effectiveness comparisons of APFD between different granularities for OCP. Each cell in the Mean, Median, and Comparison columns represents the mean APFD value, the median value, and the p-values/effect size $\hat{A}_{12}$ for the different code coverage granularity comparisons, respectively. Considering all the Java programs, as can be seen in Table 5, the mean and median APFD values at the test-method granularity are much higher than at the test-class granularity with all code coverage granularities. In fact, the test case at the test-class granularity consists of a number of test cases at the test-method granularity. For example, there exist 1000 test cases at the test-class granularity and more than 5000 test cases at test-method granularity for Java programs at Table 3, resulting in a much larger number of test cases at the test-method granularity. Thus, the permutation space of candidate test cases at the test-method granularity may be greater, which leads to a better fault detection rate (Zhang et al., 2013a).

Answer to RQ3: Overall, our analysis on the test case granularity reveals that OCP has better effectiveness performance when prioritizing test cases at the test-method granularity than at the test-class granularity in terms of fault detection rate.

5.4. RQ4: Efficiency of OCP

In this section, to evaluate the efficiency of OCP, we calculate the execution time for all TCP techniques.

Table 6 presents the statistics about time costs (i.e., the preprocessing time and prioritization time) for all subject programs and studied TCP techniques.

Specifically, the preprocessing time contains the compilation time for executing the program and the instrumentation time for collecting the coverage information. Thus the preprocessing time of the subject programs is the same for different TCP techniques and is not presented. Apart from the first two columns that display the program name and the programming language it belongs to, each cell in the table shows the mean prioritization time over the 1000 independent runs using each TCP technique.

As discussed in Section 4, the Java programs have each version individually adapted to collect the code coverage information, with different versions using different test cases. Thus, the prioritization time is collected for each Java program version. In contrast, each P_{V0} version of the C programs is compiled and instrumented to collect the code coverage information for each test case, and all program versions use the same test cases. Thus, each C program version has the same prioritization time. As a result, we present the time costs for each Java program version and C Program. Furthermore, because all the studied TCP techniques prioritize test cases after the coverage information is collected, they are all deemed to have the preprocessing time.

Based on the time costs, we have the following observations: (1) As expected, the time costs for all TCP techniques (including OCP) are lowest with method coverage, followed by branch, and then statement coverage for all programs. As shown in Table 3, the number of methods is much smaller than the number of branches, which in turn is smaller than the number of statements. Thus, the coverage information representing the related test cases is smaller, requiring less time to compute the priority value. (2) It is also expected that (for the Java programs) prioritization at the test-method granularity would take longer than at the test-class granularity, regardless of code coverage granularities. As shown in Table 3, the number of test cases to be prioritized at the test-method granularity is large than those at the test-class granularity, which requires more prioritization iterations.

(3) OCP requires much less time to prioritize test cases than most studied TCP techniques (e.g., TCP_{add}, TCP_{lexi}, TCP_{uni}, TCP_{art} and TCP_{search}) irrespective of subject program, and code coverage and test case granularities. Meanwhile, OCP can improve the costs of TCP_{add} by 85% on average. Considering that TCP_{add} remains state-of-the-art, the decreases in costs can achieve slightly better or comparable fault detection effectiveness and thus are valuable actually. It should be noted that TCP_{tot} has a much faster prioritization rate than TCP_{ocp}, as it does not use feedback information during the prioritization process. However, TCP_{tot} performs worst among all TCP techniques and is usually considered as a low bound control TCP technique (Zhang et al., 2013a).

Answer to RQ4: Overall, our analysis on the efficiency reveals that except TCP_{tot}, the APFD values of which is much lower than OCP, TCP_{ocp} has much less time to prioritize test cases than TCP_{add}, TCP_{lexi}, TCP_{uni}, TCP_{art} and TCP_{search}.

6. Related work

A considerable amount of research has been conducted to improve regression testing performance on various issues (Lou et al., 2019; Pan et al., 2022; Yu et al., 2021; ai Sun et al., 2022; Mondal and Nasre, 2021; Fang et al., 2014; Haghighatkhah et al., 2018). We focus on the coverage-based TCP techniques and summarize the existing work from the following categories.

Table 6
Efficiency: Comparisons of execution costs in milliseconds for different TCP techniques.

Language	Program	Statement coverage							Branch coverage							Method coverage						
		TCP_{tot}	TCP_{add}	TCP_{lex}	TCP_{uni}	TCP_{art}	TCP_{search}	TCP_{ocp}	TCP_{tot}	TCP_{add}	TCP_{lex}	TCP_{uni}	TCP_{art}	TCP_{search}	TCP_{ocp}	TCP_{tot}	TCP_{add}	TCP_{lex}	TCP_{uni}	TCP_{art}	TCP_{search}	TCP_{ocp}
C	*gnu_flex*	**4.31**	368.5	1028.03	687.98	7937.7	4189.23	**138.69**	**2.66**	274.83	565.48	365.06	7476.21	3846.69	**79.3**	**0.60**	60.31	45.87	31.26	353.5	3001.22	**12.78**
	gnu_grep	**2.5**	164.06	469.2	312.44	3879.79	2834.07	**48.18**	**2.02**	161.33	399.76	236.09	5442.39	2979.37	**41.39**	**0.45**	38.94	27.27	17.54	213.55	2424.7	**12.73**
	gnu_gzip	**0.49**	**8.29**	19.68	22.1	63.49	438.9	8.34	**0.26**	5.71	14.38	13.76	46.42	406.68	**4.93**	**0.06**	2.71	1.81	1.85	6.72	349.84	**2.00**
	gnu_make	**0.82**	**13.07**	40.78	40.36	101.77	556.23	13.25	**0.75**	8.91	27.91	25.95	106.16	442.69	**6.9**	**0.06**	**1.25**	2.34	2.25	5.7	205.53	1.66
	gnu_sed	**0.75**	46.54	117.93	76.77	975.4	1952.64	**12.78**	**0.49**	37.0	71.59	38.71	859.63	1949.83	**7.9**	**0.16**	15.19	12.79	7.25	84.42	1766.16	**4.01**
Java(test-method)	*ant_v*1	**1.75**	40.04	158.52	57.01	6185.81	29490.59	**6.85**	**0.49**	9.17	42.45	12.09	565.6	8852.76	**2.36**	**0.29**	7.12	28.49	8.69	1124.89	9018.78	**2.22**
	*ant_v*2	**4.9**	179.95	618.03	220.52	37120.98	35375.86	**22.17**	**1.34**	44.05	164.34	51.6	1645.86	26382.14	**6.77**	**0.78**	32.07	103.95	32.92	6315.33	21068.79	**8.24**
	*ant_v*3	**4.98**	181.64	616.08	217.66	36575.56	35217.32	**22.03**	**1.21**	45.43	164.08	50.47	1618.44	20623.5	**6.85**	**0.84**	32.36	104.01	32.83	6281.18	15832.8	**8.41**
	*ant_v*4	**30.26**	1937.49	6928.77	2470.94	531511.88	403666.85	**177.07**	**7.79**	533.67	1946.62	666.16	19113.81	104579.16	**61.56**	**4.41**	352.97	1118.36	411.38	111594.23	65434.09	**101.87**
	*ant_v*5	**33.51**	2339.8	8259.94	2980.85	701415.88	462601.72	**210.64**	**8.58**	648.16	2316.6	795.81	22050.04	115193.82	**71.31**	**5.11**	424.71	1329.35	500.8	136199.85	75283.14	**128.75**
	*ant_v*6	**32.98**	2261.22	8056.38	2899.7	675083.49	462799.01	**204.27**	**9.04**	635.54	2271.12	779.82	18858.47	102967.53	**70.19**	**4.98**	417.81	1294.7	488.7	127806.2	73827.57	**125.04**
	*ant_v*7	**75.39**	6447.83	28863.39	10719.12	3582724.65	1243589.12	**608.8**	**21.54**	2449.7	8276.29	3030.26	103270.76	302281.19	**222.36**	**12.84**	1577.69	4793.59	1913.45	697296.49	135253.39	**486.84**
	*ant_v*8	**71.15**	6576.87	29071.21	10645.15	3588748.59	1225224.6	**637.39**	**20.11**	1985.4	8372.64	3052.9	104345.4	256010.28	**225.68**	**12.13**	1637.23	4832.87	1911.08	701421.34	175257.75	**579.66**
	*jtopas_v*1	**2.05**	37.08	157.98	52.08	5590.25	49553.33	**10.79**	**0.66**	13.76	56.64	18.1	2054.62	22940.51	**3.86**	**0.14**	3.34	12.28	3.4	528.61	3118.26	**1.98**
	*jtopas_v*2	**1.97**	39.16	164.99	55.08	6068.39	59249.46	**11.3**	**0.67**	15.49	59.5	19.12	2170.79	47675.36	**3.91**	**0.16**	3.74	13.44	3.9	576.68	8540.78	**2.54**
	*jtopas_v*3	**6.98**	166.64	714.39	228.32	57582.22	65410.86	**36.91**	**2.34**	59.67	257.85	78.34	14679.05	60761.66	**14.23**	**0.43**	16.93	54.2	16.77	3955.14	32994.5	**6.76**
	*jmeter_v*1	**0.44**	4.94	27.79	6.18	346.59	31128.38	**1.58**	**0.18**	1.02	6.16	1.23	25.68	5259.7	**0.44**	**0.14**	0.89	4.88	1.01	68.12	5013.62	**0.44**
	*jmeter_v*2	1.12	10.99	68.28	13.83	1063.24	25942.38	**3.12**	**0.24**	1.73	9.64	2.07	47.21	6825.12	**0.62**	**0.25**	2.09	10.6	2.2	185.66	7374.39	**0.86**
	*jmeter_v*3	**1.39**	14.1	81.6	17.61	1380.78	17542.05	**3.78**	**0.16**	1.93	10.85	2.39	58.86	4833.83	**0.7**	**0.19**	2.94	15.61	3.51	295.78	2938.2	**0.97**
	*jmeter_v*4	**1.12**	16.06	83.42	18.04	1422.88	11761.66	**3.81**	**0.16**	2.1	10.86	2.4	59.07	1481.61	**0.67**	**0.21**	3.3	15.66	3.65	303.56	1934.82	**1.01**
	*jmeter_v*5	**1.24**	18.35	96.51	21.21	1801.14	13119.46	**4.42**	**0.23**	2.47	12.67	2.84	71.5	1347.98	**0.77**	**0.42**	3.69	18.32	4.47	377.77	2001.66	**1.20**
	*xmlsec_v*1	**1.13**	22.06	99.16	22.64	2365.41	10392.07	**4.20**	**0.47**	4.69	24.3	5.52	118.31	1368.55	**1.20**	**0.25**	3.35	14.51	3.06	339.86	1435.36	**1.16**
	*xmlsec_v*2	**1.21**	24.06	108.63	26.1	2703.6	11188.71	**4.97**	**0.54**	6.37	28.58	7.03	150.92	1593.97	**1.54**	**0.18**	3.54	15.26	3.34	375.93	1363.08	**1.30**
	*xmlsec_v*3	**1.21**	18.78	89.32	20.26	2098.82	10128.92	**4.45**	**0.36**	5.12	24.59	5.69	111.92	1314.03	**1.28**	**0.14**	2.7	12.03	2.45	259.3	1050.24	**1.03**
Java(test-class)	*ant_v*1	**0.42**	3.69	19.67	4.33	133.56	6457.16	**1.44**	**0.13**	0.82	5.71	1.01	35.04	1178.75	**0.40**	**0.10**	0.62	3.86	0.73	25.2	922.62	**0.33**
	*ant_v*2	**1.36**	11.57	64.61	14.26	628.79	18689.06	**3.65**	**0.44**	2.93	17.66	3.64	171.69	3686.37	**1.01**	**0.19**	1.78	11.26	2.43	111.20	2376.55	**0.72**
	*ant_v*3	**1.4**	11.37	64.0	14.28	630.27	19582.33	3.85	**0.27**	2.87	17.52	3.58	170.01	4371.02	**1.06**	**0.28**	1.79	10.98	2.29	109.12	2755.49	**0.70**
	*ant_v*4	**5.76**	76.96	420.2	101.54	9003.24	102183.09	**17.85**	**1.57**	22.47	119.32	26.76	1933.59	19413.74	**4.94**	**1.22**	13.56	64.98	15.84	1465.66	13110.27	**2.90**
	*ant_v*5	**6.44**	88.95	460.8	109.61	10078.73	116900.69	**19.14**	**1.72**	24.71	131.13	28.89	2139.8	20867.59	**5.34**	**0.9**	16.01	70.96	16.95	1656.95	13822.1	**3.12**
	*ant_v*6	**6.65**	82.74	451.09	107.43	9705.8	116659.29	**18.64**	**1.84**	22.4	128.29	27.67	2017.54	21966.04	**5.14**	**0.89**	13.27	69.35	16.64	1587.45	15390.26	**3.02**
	*ant_v*7	**12.8**	249.01	1193.98	326.64	131530.77	163839.7	**39.39**	**3.60**	69.75	344.87	90.07	5848.22	54516.25	**11.67**	**2.00**	42.48	196.28	53.68	7277.59	32652.62	**7.48**
	*ant_v*8	**12.68**	247.24	1196.01	325.44	132814.84	156276.32	**39.79**	**3.61**	70.38	346.6	90.6	5934.71	57982.7	**11.84**	**1.97**	41.67	195.86	53.18	7261.99	41400.42	**7.17**
	*jtopas_v*1	**0.15**	0.62	4.66	0.74	6.16	8268.77	**0.50**	**0.06**	0.20	1.73	0.28	2.29	2093.0	**0.19**	**0.02**	0.05	0.42	0.05	0.53	677.81	**0.04**
	*jtopas_v*2	**0.16**	0.61	5.34	0.86	7.67	7132.1	**0.55**	**0.06**	0.22	1.98	0.37	2.95	2625.9	**0.20**	**0.02**	**0.05**	0.48	0.08	0.78	898.9	**0.05**
	*jtopas_v*3	**0.49**	2.12	15.62	2.65	31.87	27084.71	**1.38**	**0.19**	0.72	5.63	0.88	11.73	6509.74	**0.48**	**0.07**	0.16	1.36	0.22	3.02	1514.9	**0.13**
	*jmeter_v*1	**0.2**	0.77	5.99	0.89	16.66	6560.88	**0.44**	**0.06**	0.15	1.33	0.19	3.66	1602.51	**0.10**	**0.04**	0.16	1.19	0.18	3.36	1254.53	**0.09**
	*jmeter_v*2	**0.31**	1.32	12.27	1.81	38.84	10595.78	**0.79**	**0.07**	0.24	1.85	0.28	6.11	1655.58	**0.14**	**0.05**	0.26	2.06	0.34	6.66	1272.91	**0.18**
	*jmeter_v*3	**0.36**	1.97	16.52	2.57	64.97	13330.05	**1.02**	**0.07**	0.3	2.57	0.44	9.54	1464.98	**0.16**	**0.08**	0.44	3.37	0.53	14.07	1915.9	**0.25**
	*jmeter_v*4	**0.36**	1.98	16.94	2.64	67.29	13366.85	**1.05**	**0.06**	0.28	2.59	0.31	9.44	2238.46	**0.17**	**0.09**	0.42	3.44	0.54	14.48	2167.08	**0.25**
	*jmeter_v*5	**0.43**	2.43	20.95	3.28	93.95	10861.39	**1.30**	**0.07**	0.36	3.02	0.51	11.7	2088.49	**0.20**	**0.08**	0.53	4.19	0.66	19.83	2475.02	**0.29**
	*xmlsec_v*1	**0.21**	1.02	8.25	1.21	18.86	6315.35	**0.65**	**0.06**	0.24	2.14	0.3	4.15	576.82	**0.18**	**0.04**	0.16	1.3	0.16	2.79	875.06	**0.10**
	*xmlsec_v*2	**0.2**	1.01	8.53	1.24	19.05	6104.03	**0.64**	**0.09**	0.25	2.47	0.32	4.31	1269.4	**0.18**	**0.06**	0.14	1.22	0.18	2.72	937.54	**0.11**
	*xmlsec_v*3	**0.17**	0.77	6.9	0.98	12.84	4190.4	**0.57**	**0.06**	0.21	2.01	0.26	2.93	521.64	**0.14**	**0.04**	0.11	1.01	0.14	1.8	706.94	**0.08**
Average		**7.77**	505.2	2091.45	764.06	222084.94	116924.45	**54.71**	**2.24**	166.81	611.01	221.85	7517.83	30431.32	**20.47**	**1.24**	111.17	337.81	129.6	42221.74	18223.62	**35.36**

6.1. Prioritization strategies

Despite the large body of research on coverage-based TCP (Fang et al., 2014; Miranda et al., 2018; Epitropakis et al., 2015; Peng et al., 2020), the *total-greedy* and *additional-greedy* greedy strategies remain the most widely investigated prioritization strategies (Rothermel et al., 1999). In addition to the above *greedy-based* strategies, researchers have also investigated other generic strategies (Li et al., 2007; Jiang et al., 2009). For example, Li et al. (2007) transform the TCP problem into a search problem and propose two *search-based* prioritization strategies (i.e., a hill-climbing strategy and a genetic strategy). Furthermore, inspired by the advantages of adaptive random testing (ART) in replacing random testing (RT) (Huang et al., 2019; Jiang et al., 2009) investigate ART to improve random test case prioritization and propose an *art-based* strategy based on the distribution of test cases across the input domain.

Researchers have also proposed some alternative strategies in previous studies to take further advantage of the code coverage information (e.g., the covered times of code units). For example, Eghbali and Tahvildari (2016) propose an enhanced *additional-greedy* strategy for breaking ties using the notion of lexicographical ordering (i.e., the *lexicographical-greedy* strategy), where fewer covered code units should have higher priority value for coverage. Specifically, unlike traditional *greedy-based* strategies (Elbaum et al., 2000), Eghbali et al. do not categorize all code units into two distinct groups, (i.e., covered and not covered). At each iteration, huge calculations are required to calculate the priority values of all code units based on the number of times they are covered, and then each one in the remaining test cases is lexicographically compared against others. Similarly, Zhang et al. (2013a) propose a variant of the *additional-greedy* strategy to unify the *total-greedy* strategy and the *additional-greedy* strategy. Each time a code unit is covered by a test case, the probability that the code unit contains undetected faults is reduced by some ratio between 0% (as in the *total-greedy* strategy) and 100% (as in the *additional-greedy* strategy), which can also be considered as an effective strategy to break tie cases. The above techniques attempt to make use of more accurate coverage information obtained by additional calculations. For example, the *lexicographical-greedy* strategy needs to rank all code units based on the number of times they are covered, while the *unified-greedy* strategy needs to calculate the value of fault detection probability for each code unit. Most recently, Zhou et al. (2022) propose eight parallel test prioritization techniques, which are adapted from four typical sequential test prioritization techniques (including *total-greedy*, the *additional-greedy* strategy, the *search-based* strategy, and the *art-based* strategy). Different from traditional sequential TCP techniques, it aims at generating a set of test sequences, each of which is allocated in an individual computing resource. Thus, we do not include it in this work.

It can be observed that most existing TCP strategies tend to consider the whole candidate set in prioritization iterations. To address this limitation, we pay attention to partial candidate test cases with the aid of previous priority values, resulting in a better performance in both effectiveness and efficiency.

6.2. Coverage criteria

In principle, TCP techniques can use any test adequacy criterion as the underlying coverage criterion (Hao et al., 2016; Fang et al., 2012). Among various criteria, structural coverage has been widely adopted in previous TCP research, such as statement coverage (Rothermel et al., 1999; Di Nucci et al., 2020), branch coverage (Jiang et al., 2009), method coverage (Zhang et al., 2013a; Wang et al., 2017), block coverage (Li et al., 2007) and modified condition/decision coverage (Jones and Harrold, 2003). Elbaum et al. (2000) also propose a fault-exposing-potential (FEP) criterion based on the probability of the test case detecting a fault. Fang et al. (2012) use logic coverage for TCP, where high coverage of logic expressions implies a high probability of detecting faults. Recently, Chi et al. (2018) demonstrate that basic structural coverage may not be enough to predict fault detection capability and propose a dynamic relation-based coverage based on method call sequences. Wang et al. (2017) detect fault-prone source code by existing code inspection techniques and then propose a quality-aware TCP technique (i.e., QTEP) by considering the weighted source code in terms of fault-proneness. However, such techniques require not only coverage information but also other source code information (e.g., the defect prediction results and method call sequences) and thus are not considered in our work. In this work, we investigate how the basic structure coverage criteria influence the performance of TCP techniques.

6.3. Empirical studies

As an effective regression testing technique, TCP has been extensively studied in the literature from both academic and industrial perspectives. Recently, researchers also performed a large number of empirical studies to investigate TCP from different aspects. For example, several studies usually focus on the performance of the traditional dynamic test prioritization regarding some effectiveness and efficiency criteria (e.g., *APFD*, $APFD_C$, and prioritization time) (Rothermel et al., 1999; Elbaum et al., 2000; Do et al., 2006; Do et al., 2010). Meanwhile, Lu et al. (2016) were the first to investigate how real-world software evolution impacts the performance of prioritization strategies: They reported that source code changes have a low impact on the effectiveness of traditional dynamic techniques, but that the opposite was true when considering new tests in the process of evolution. Citing a lack of comprehensive studies comparing static and dynamic test prioritization techniques, Luo et al. (2016), Luo et al. (2019) compared static TCP techniques with dynamic ones. Henard et al. (2016) compared white-box and back-box TCP techniques. In this work, we focus on the coverage-based TCP techniques and conduct an extensive study to evaluate OCP with six state-of-the-art TCP techniques.

7. Threats to validity

To facilitate the replication and verification of our experiments, we have made the relevant materials (including source code, subject programs, test suites, and mutants) available at Anon. (0000a). Despite that, our study may still face some threats to validity.

7.1. Internal validity

The implementation of our experiment may introduce threats to internal validity. First, randomness might affect the reliability of conclusions. To address this, we repeat the prioritization process 1000 times and used statistical analysis to assess the strategies. Second, the data structures used in the prioritization algorithms, and the faults in the source code, may introduce noise when evaluating the effectiveness and efficiency. To minimize these threats, we use data structures that are as similar as possible, and carefully reviewed all source code before conducting the experiment. Third, to assess the effectiveness of TCP techniques, the most widely used metric *APFD* is adopted in our experiment. However, *APFD* only reflects the rate at which faults are detected, ignoring the time and space costs. Our future work will involve additional metrics (e.g., $APFD_C$) that can measure other practical performance aspects of prioritization strategies.

7.2. External validity

The main threat to external validity lies in the selection of the subject programs and faults. First, although 19 Java and 30 C program versions with various sizes are adopted in our experiment, the results may not generalize to programs written in other languages (e.g., C++ and Python). Meanwhile, the relative performances of TCP techniques on the used mutants may not be generalizable to the real faults, despite the fact that mutation testing have argued to be an appropriate approach for assessing TCP performance (Andrews et al., 2005; Do et al., 2005; Just et al., 2014), To mitigate these threats, additional studies will be conducted to investigate the performance of TCP on programs with real faults and other languages in the future.

8. Conclusion

In this paper, we have introduced a generic partial attention mechanism that adopts priority values of previously selected test cases to avoid considering all test cases. We also apply the concept to the *additional-greedy* strategy and implement a novel coverage-based TCP technique, *partition ordering based prioritization* (OCP). Results from our empirical study have demonstrated that OCP can achieve better fault detection rate than six state-of-the-arts (i.e., *total-greedy*, *additional-greedy*, *unified-greedy*, *lexicographical-greedy*, *art-based*, and *search-based* TCP techniques). OCP is also found to have much less prioritization time to prioritize test cases than most state-of-the-arts (except the *total-greedy* strategy) and the improvement can achieve 85%–99% on average.

In the future, we plan to continue refining the generic partial attention mechanism and extend it to other TCP techniques (e.g., the *lexicographical-greedy* strategy). We will also launch an extensive effort on understanding the impact of the proposed technique for other application domains of TCP research (Wang et al., 2021; Chen et al., 2020; Sharif et al., 2021), such as configuration testing (Cheng et al., 2021; Sun et al., 2020) and combinatorial testing (Henard et al., 2014; Wu et al., 2020).

CRediT authorship contribution statement

Quanjun Zhang: Formal analysis, Reviewing and editing, Writing – original draft, Investigation, Software, Methodology, Conceptualization. **Chunrong Fang:** Project administration, Investigation, Resources, Reviewing and editing, Supervision, Funding acquisition. **Weisong Sun:** Reviewing and editing, Validation. **Shengcheng Yu:** Visualization, Formal analysis. **Yutao Xu:** Writing – review & editing. **Yulei Liu:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The authors would like to thank the anonymous reviewers for insightful comments. This research is partially supported by the National Natural Science Foundation of China (No. 61932012, 62141215) and Science, Technology and Innovation Commission of Shenzhen Municipality, China (CJGJZD20200617103001003).

References

Andrews, J.H., Briand, L.C., Labiche, Y., 2005. Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th International Conference on Software Engineering, ICSE'05. pp. 402–411.

Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S., 2006. Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans. Softw. Eng. 32 (8), 608–624.

Anon., 0000. The project website. https://github.com/QuanjunZhang/OCP. (Accessed 10 March 2022).

Anon., 0000. Software-artifact Infrastructure Repository (SIR). https://sir.csc.ncsu.edu/portal/index.php. (Accessed 10 March 2022).

Anon., 0000. GNU FTP Server. http://ftp.gnu.org/. (Accessed 10 March 2022).

Anon., 0000. ASM: An all purpose Java bytecode manipulation and analysis framework. http://asm.ow2.org/. (Accessed 10 March 2022).

Anon., 0000. gcc: The GNU Compiler Collection. https://gcc.gnu.org/. (Accessed 10 March 2022).

Anon., 0000. gcov: A test coverage program. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html. (Accessed 10 March 2022).

Anon., 0000. R: The R project for statistical computing. https://www.r-project.org/. (Accessed 10 March 2022).

Arcuri, A., Briand, L., 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Reliab. 24 (3), 219–250.

Belli, F., Budnik, C.J., Hollmann, A., Tuglular, T., Wong, W.E., 2016. Model-based mutation testing—approach and case studies. Sci. Comput. Program. 120, 25–48.

Belli, F., Budnik, C.J., Wong, W.E., 2006. Basic operations for generating behavioral mutants. In: Second Workshop on Mutation Analysis (Mutation 2006-ISSRE Workshops 2006). p. 9.

Chen, J., Lou, Y., Zhang, L., Zhou, J., Wang, X., Hao, D., Zhang, L., 2018. Optimizing test prioritization via test distribution analysis. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE'18. pp. 656–667.

Chen, J., Wu, Z., Wang, Z., You, H., Zhang, L., Yan, M., 2020. Practical accuracy estimation for efficient deep neural network testing. ACM Trans. Softw. Eng. Methodol. 29 (4), 1–35.

Cheng, R., Zhang, L., Marinov, D., Xu, T., 2021. Test-case prioritization for configuration testing. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'21. pp. 452–465. New York, NY, USA.

Chi, J., Qu, Y., Zheng, Q., Yang, Z., Jin, W., Cui, D., Liu, T., 2018. Test case prioritization based on method call sequences. In: Proceedings of the 42nd IEEEAnnual Computer Software and Applications Conference, COMPSAC'18, vol. 01, pp. 251–256.

Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A., 2016. PIT: A practical mutation testing tool for Java (Demo). In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA'16. pp. 449–452.

Cruciani, E., Miranda, B., Verdecchia, R., Bertolino, A., 2019. Scalable approaches for test suite reduction. In: 2019 IEEE/ACM 41st International Conference on Software Engineering, ICSE'19. pp. 419–429.

Di Nucci, D., Panichella, A., Zaidman, A., De Lucia, A., 2020. A test case prioritization genetic algorithm guided by the hypervolume indicator. IEEE Trans. Softw. Eng. 46 (6), 674–696.

Do, H., Elbaum, S., Rothermel, G., 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empir. Softw. Eng. 10 (4), 405–435.

Do, H., Mirarab, S., Tahvildari, L., Rothermel, G., 2010. The effects of time constraints on test case prioritization: A series of controlled experiments. IEEE Trans. Softw. Eng. 36 (5), 593–617.

Do, H., Rothermel, G., 2005. A controlled experiment assessing test case prioritization techniques via mutation faults. In: Proceedings of the 21st IEEE International Conference on Software Maintenance, ICSM'05. pp. 411–420.

Do, H., Rothermel, G., Kinneer, A., 2006. Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis. Empir. Softw. Eng. 11 (1), 33–70.

Eghbali, S., Tahvildari, L., 2016. Test case prioritization using lexicographical ordering. IEEE Trans. Softw. Eng. 42 (12), 1178–1195.

Elbaum, S., Malishevsky, A.G., Rothermel, G., 2000. Prioritizing test cases for regression testing. In: Proceedings of the 8th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'00. pp. 102–112.

Elbaum, S., Rothermel, G., Penix, J., 2014. Techniques for improving regression testing in continuous integration development environments. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE'14. pp. 235–245.

Elsner, D., Hauer, F., Pretschner, A., Reimer, S., 2021. Empirically evaluating readily available information for regression test optimization in continuous integration. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'21. pp. 491–504.

Epitropakis, M.G., Yoo, S., Harman, M., Burke, E.K., 2015. Empirical evaluation of Pareto efficient multi-objective regression test case prioritisation. In: Proceedings of the 23rd International Symposium on Software Testing and Analysis, ISSTA'15. pp. 234–245.

Fang, C., Chen, Z., Wu, K., Zhao, Z., 2014. Similarity-based test case prioritization using ordered sequences of program entities. Softw. Qual. J. 22 (2), 335–361.

Fang, C., Chen, Z., Xu, B., 2012. Comparing logic coverage criteria on test case prioritization. Sci. China Inf. Sci. 55 (12), 2826–2840.

Gazzola, L., Micucci, D., Mariani, L., 2017. Automatic software repair: A survey. IEEE Trans. Softw. Eng. 45 (1), 34–67.

Gligoric, M., Eloussi, L., Marinov, D., 2015a. Practical regression test selection with dynamic file dependencies. In: Proceedings of the 24th International Symposium on Software Testing and Analysis, ISSTA'15. pp. 211–222.

Gligoric, M., Groce, A., Zhang, C., Sharma, R., Alipour, M.A., Marinov, D., 2015b. Guidelines for coverage-based comparisons of non-adequate test suites. ACM Trans. Softw. Eng. Methodol. 24 (4), 22:1–22:33.

Haghighatkhah, A., Mäntylä, M., Oivo, M., Kuvaja, P., 2018. Test prioritization in continuous integration environments. J. Syst. Softw. 146, 80–98.

Hao, D., Zhang, L., Zang, L., Wang, Y., Wu, X., Xie, T., 2016. To be optimal or not in test-case prioritization. IEEE Trans. Softw. Eng. 42 (5), 490–505.

Hao, D., Zhang, L., Zhang, L., Rothermel, G., Mei, H., 2014. A unified test case prioritization approach. ACM Trans. Softw. Eng. Methodol. 24 (2), 10:1–10:31.

Henard, C., Papadakis, M., Harman, M., Jia, Y., Le Traon, Y., 2016. Comparing white-box and black-box test prioritization. In: Proceedings of the 38th IEEE/ACM International Conference on Software Engineering, ICSE'16. pp. 523–534.

Henard, C., Papadakis, M., Perrouin, G., Klein, J., Heymans, P., Le Traon, Y., 2014. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. IEEE Trans. Softw. Eng. 40 (7), 650–670.

Huang, R., Sun, W., Xu, Y., Chen, H., Towey, D., Xia, X., 2019. A survey on adaptive random testing. IEEE Trans. Softw. Eng. 47 (10), 2052–2083.

Huang, R., Zhang, Q., Towey, D., Sun, W., Chen, J., 2020. Regression test case prioritization by code combinations coverage. J. Syst. Softw. 169, 110712.

Jiang, B., Zhang, Z., Chan, W.K., Tse, T.H., 2009. Adaptive random test case prioritization. In: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering, ASE'09. pp. 233–244.

Jones, J.A., Harrold, M.J., 2003. Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Trans. Softw. Eng. 29 (3), 195–209.

Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G., 2014. Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE'14. pp. 654–665.

Khatibsyarbini, M., Isa, M.A., Jawawi, D.N., Tumeng, R., 2018. Test case prioritization approaches in regression testing: A systematic literature review. Inf. Softw. Technol. 93, 74–93.

Lam, W., Shi, A., Oei, R., Zhang, S., Ernst, M.D., Xie, T., 2020. Dependent-test-aware regression testing techniques. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'20. pp. 298–311.

Li, Z., Harman, M., Hierons, R.M., 2007. Search algorithms for regression test case prioritization. IEEE Trans. Softw. Eng. 33 (4), 225–237.

Li, F., Zhou, J., Li, Y., Hao, D., Zhang, L., 2021. AGA: An accelerated greedy additional algorithm for test case prioritization. IEEE Trans. Softw. Eng..

Lou, Y., Chen, J., Zhang, L., Hao, D., 2019. A survey on regression test-case prioritization. In: Advances in Computers, Vol. 113. Elsevier, pp. 1–46.

Lu, Y., Lou, Y., Cheng, S., Zhang, L., Hao, D., Zhou, Y., Zhang, L., 2016. How does regression test prioritization perform in real-world software evolution? In: Proceedings of the 38th International Conference on Software Engineering, ICSE'16. pp. 535–546.

Luo, Q., Moran, K., Poshyvanyk, D., 2016. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE'16. pp. 559–570.

Luo, Q., Moran, K., Poshyvanyk, D., Di Penta, M., 2018. Assessing test case prioritization on real faults and mutants. In: Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution, ICSME'18. pp. 240–251.

Luo, Q., Moran, K., Zhang, L., Poshyvanyk, D., 2019. How do static and dynamic test case prioritization techniques perform on modern software systems? An extensive study on GitHub projects. IEEE Trans. Softw. Eng. 45 (11), 1054–1080.

Mäntylä, M.V., Adams, B., Khomh, F., Engström, E., Petersen, K., 2015. On rapid releases and software testing: A case study and a semi-systematic literature review. Empir. Softw. Eng. 20 (5), 1384–1425.

Mei, H., Hao, D., Zhang, L., Zhang, L., Zhou, J., Rothermel, G., 2012. A static approach to prioritizing junit test cases. IEEE Trans. Softw. Eng. 38 (6), 1258–1275.

Memon, A., Gao, Z., Nguyen, B., Dhanda, S., Nickell, E., Siemborski, R., Micco, J., 2017. Taming Google-scale continuous testing. In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP'17. pp. 233–242.

Miranda, B., Cruciani, E., Verdecchia, R., Bertolino, A., 2018. Fast approaches to scalable similarity-based test case prioritization. In: Proceedings of the 40th International Conference on Software Engineering, ICSE'18. pp. 222–232.

Mondal, S., Nasre, R., 2021. Hansie: Hybrid and consensus regression test prioritization. J. Syst. Softw. 172, 110850.

Pan, R., Bagherzadeh, M., Ghaleb, T.A., Briand, L., 2022. Test case selection and prioritization using machine learning: A systematic literature review. Empir. Softw. Eng. 27 (2), 1–43.

Papadakis, M., Henard, C., Harman, M., Jia, Y., Le Traon, Y., 2016. Threats to the validity of mutation-based test assessment. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA'16. pp. 355–365.

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M., 2019. Mutation testing advances: An analysis and survey. In: Advances in Computers, Vol. 112. Elsevier, pp. 275–378.

Peng, Q., Shi, A., Zhang, L., 2020. Empirically revisiting and enhancing IR-based test-case prioritization. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'20. pp. 324–336.

do Prado Lima, J.A., Vergilio, S.R., 2022. A multi-armed bandit approach for test case prioritization in continuous integration environments. IEEE Trans. Softw. Eng. 48 (2), 453–465.

Rothermel, G., Untch, R.H., Harrold, M.J., 1999. Test case prioritization: An empirical study. In: Proceedings of the 15th IEEE International Conference on Software Maintenance, ICSM'99. pp. 179–188.

Sadri-Moshkenani, Z., Bradley, J., Rothermel, G., 2022. Survey on test case generation, selection and prioritization for cyber-physical systems. Softw. Test. Verif. Reliab. 32 (1), e1794.

Sharif, A., Marijan, D., Liaaen, M., 2021. DeepOrder: Deep learning for test case prioritization in continuous integration testing. In: 2021 IEEE International Conference on Software Maintenance and Evolution, ICSME'21. IEEE, pp. 525–534.

Sun, X., Cheng, R., Chen, J., Ang, E., Legunsen, O., Xu, T., 2020. Testing configuration changes in context to prevent production failures. In: 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI'20. pp. 735–751.

ai Sun, C., Liu, B., Fu, A., Liu, Y., Liu, H., 2022. Path-directed source test case generation and prioritization in metamorphic testing. J. Syst. Softw. (ISSN: 0164-1212) 183, 111091.

Vargha, A., Delaney, H.D., 2000. A critique and improvment of the CL common language effect size statistics of mcgraw and wong. J. Educ. Behav. Stat. 25 (2), 101–132.

Wang, S., Nam, J., Tan, L., 2017. QTEP: Quality-aware test case prioritization. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE'17. pp. 523–534.

Wang, Z., You, H., Chen, J., Zhang, Y., Dong, X., Zhang, W., 2021. Prioritizing test inputs for deep neural networks via mutation analysis. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering, ICSE'21. IEEE, pp. 397–409.

Weimer, W., Nguyen, T., Le Goues, C., Forrest, S., 2009. Automatically finding patches using genetic programming. In: Proceedings of the 31st International Conference on Software Engineering, ICSE'09. pp. 364–374.

Wong, W.E., Horgan, J.R., London, S., Agrawal, H., 1997. A study of effective regression testing in practice. In: Proceedings of the 8th International Symposium on Software Reliability Engineering. pp. 264–274.

Wong, W.E., Horgan, J.R., London, S., Mathur, A.P., 1998. Effect of test set minimization on fault detection effectiveness. Softw. - Pract. Exp. 28 (4), 347–369.

Wu, H., Nie, C., Petke, J., Jia, Y., Harman, M., 2020. An empirical comparison of combinatorial testing, random testing and adaptive random testing. IEEE Trans. Softw. Eng. 46 (3), 302–320.

Yoo, S., Harman, M., 2012. Regression testing minimization, selection and prioritization: A survey. Softw. Test. Verif. Reliab. 22 (2), 67–120.

Yu, S., Fang, C., Cao, Z., Wang, X., Li, T., Chen, Z., 2021. Prioritize crowdsourced test reports via deep screenshot understanding. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering, ICSE'21. IEEE, pp. 946–956.

Zhang, L., 2018. Hybrid regression test selection. In: Proceedings of the 40th International Conference on Software Engineering, ICSE'18. pp. 199–209.

Zhang, L., Hao, D., Zhang, L., Rothermel, G., Mei, H., 2013a. Bridging the gap between the total and additional test-case prioritization strategies. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE'13. pp. 192–201.

Zhang, L., Kim, M., Khurshid, S., 2012. FaultTracer: A change impact and regression fault analysis tool for evolving Java programs. In: Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'12. pp. 40.

Zhang, L., Kim, M., Khurshid, S., 2013b. FaultTracer: A spectrum-based approach to localizing failure-inducing program edits. J. Softw. Evol. Process 25 (12), 1357–1383.

Zhang, J., Zhang, L., Harman, M., Hao, D., Jia, Y., Zhang, L., 2019. Predictive mutation testing. IEEE Trans. Softw. Eng. 45 (9), 898–918.

Zhang, L., Zhou, J., Hao, D., Zhang, L., Mei, H., 2009. Prioritizing JUnit test cases in absence of coverage information. In: Proceedings of the 25th IEEE International Conference on Software Maintenance, ICSM'09. pp. 19–28.

Zhou, J., Chen, J., Hao, D., 2022. Parallel test prioritization. ACM Trans. Softw. Eng. Methodol. 31 (1), 1–50.

Quanjun Zhang is currently working toward the Ph.D. degree in Software Institute at Nanjing University. He received the B.E. and M.Eng. degree in computer science and technology from Jiangsu University, Zhenjiang, China. His current research interests include software testing and program repair.

Chunrong Fang received the B.E. and Ph.D. degrees in software engineering from Software Institute, Nanjing University, China. He is currently a research assistant at the Software Institute at Nanjing University. He is the director of Nanjing University-Shudui joint research center. His research interests lie in intelligent software engineering, e.g. BigCode and AITesting.

Weisong Sun is a Ph.D Candidate in the Software Institute at Nanjing University, China. His research interests include Artificial Intelligence for Software Engineering and Software Engineering for Artificial Intelligence. He received a B.E. in software engineering from Yangzhou University, China. Contact him at weisongsun@smail.nju.edu.cn.

Shengcheng Yu is currently a Ph.D. student in Nanjing University. He got his bachelor degree in Nanjing University in 2020. His research interests include software testing, mobile application testing, crowdsourced testing, image understanding, multi-modal information semantic analysis and deep learning applications.

Yutao Xu is currently a master student in Nanjing University. He got his bachelor degree in Beijing University of Posts and Telecommunications in 2021. His research interests include software testing.

Yulei Liu is currently an undergraduate student in Nanjing University. His research interests include mobile app testing and reinforcement learning.