2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW) | 979-8-3503-1956-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISSREW60843.2023.00072

# Black-Box Test Case Prioritization Using Log Analysis and Test Case Diversity

Xiaolei Yu, Kai Jia, Wenhua Hu, Jing Tian, Jianwen Xiang*
*School of Computer Science and Artificial Intelligence*
*Wuhan University of Technology, Wuhan, China*
{yuxiaolei, 290308, whu10, jtian, jwxiang}@whut.edu.cn

***Abstract*—Regression testing is a software testing type that examines whether updates made in the software impact the existing functionality of the application. Depressingly, the long testing time and high testing costs make regression testing very expensive. Test case prioritization (TCP) stands out as one of the extensively researched regression testing techniques. It prioritizes test cases to optimize their execution order, aiming to maximize the prioritization goals and reveal faults earlier to provide feedback to testers. The TCP technique based on log analysis (LogTCP) designs the prioritization strategy using logs generated during test case execution. However, LogTCP's performance is limited by its inability to incorporate the diversity of test cases for sorting. To overcome these concerns, we propose a method to implement TCP using k-means clustering and log analysis(called KL-TCP), that takes into account both log information and test case diversity. We examine the effectiveness of this strategy in ten open source Java projects on GitHub. The experimental results show that our proposed method outperforms LogTCP method by detecting a higher average percentage of faults. The best average performance of the ten project experiments reached 0.77(APFD).**

***Index Terms*—test case prioritization, test case diversity, log parsing**

## I. Introduction

Software testing is an effective method to ensure software quality during the software development life cycle. Regression testing is a form of software testing that examines whether changes, updates, or improvements made in the software impact the existing functionality of the application. However, regression testing is an expensive process that can incur significant testing costs. To reduce the overall time for testing, Test Case Prioritization (TCP) [1] techniques are proposed to optimize the test case suite. It prioritizes test cases to optimize their execution order by setting specific prioritization criteria (e.g., execution time, code coverage, etc.), aiming to maximize the prioritization goals and reveal faults earlier to provide feedback to testers [1].

Depending on the source of information used, TCP techniques can be categorized into white-box TCP (WTCP) and black-box TCP (BTCP) [2]. The former investigates coverage information [3], source code changes [4], test case execution history [5] and fallibility of code units [6], [7] to prioritize test cases. In contrast to WTCP, BTCP does not require source code analysis and only utilizes data generated by the current software or by the test cases themselves, including test-case text information [8], [9], test log [10], execution history [11], test requirements [12]. There have been many studies of WTCP that have achieved significant efforts in effectiveness [13]–[15]. However, it has limited use in closed-source project or outsourced testing scenarios. It is also costly to gather coverage information from different versions of code. In contrast, BTCP offers the advantage of not relying on software code information. However, it possesses less available information and tends to demonstrate inferior performance compared to WTCP. Thus, both WTCP and BTCP exhibit practical limitations.

To explore more efficient BTCP technologies and bridge the performance gap between BTCP and WTCP, Chen et al. [10] proposed a log-based TCP framework (LogTCP) by mining test logs generated during test case execution.In their work, the final test sequence outperforms the state-of-the-art BTCP, even competes with the WTCP technique in average fault detection. However, the strategy of prioritizing test cases through log analysis still presents challenges. For example, if the test cases are ranked based only on the count of coverage they have for log events, the test cases that cover more log events will have a higher priority, despite being similar and potentially redundant. It has been hypothesised that test cases sharing similar attributes may also detect same faults [16], [17]. Based on this conjecture, some approaches [7], [18], [19] are proposed to consider diversity when prioritizing test cases improves the failure detection rate of TCP.

In this paper, we attempt to explore the effectiveness of BTCP by combining log information with test case diversity. Specially, based on count, ordering and semantic features of the logs generated by the test cases, our approach further classifies similar test cases using the $k$-means clustering. To prioritize test cases within and between clusters, we use a combination of three popular ranking strategies, including total strategy [1], additional strategy [1], and adaptive random prioritization (ARP) strategy [20]. By combining different log features with inner cluster and inter-cluster ranking methods, our approach is able to account for both log information and test case diversity. We conducted experiments based on 10 Java projects from the Github to study the effectiveness of KL-TCP by comparing with advanced LogTCP techniques.

This paper makes the following contributions:

- Considering both log information and test case diversity, a BTCP approach named KL-TCP is proposed in this paper. We have designed a BTCP framework that implements the BTCP technique by combining a series of log

979-8-3503-1956-9/23/$31.00 ©2023 IEEE
DOI 10.1109/ISSREW60843.2023.00072

representation strategies with a clustering algorithm.
- We conduct an extensive study to compare the effectiveness of our proposed KL-TCP approach with LogTCP, utilizing 10 open-source Java projects. The experimental results show that our approach detects faults faster than the LogTCP method and achieves a higher fault detection rate.

## II. Related Work

### A. Test Case Prioritization

TCP is categorized as WTCP or BTCP depending on whether or not source code information is used. WTCP has a strong dependency on source code information and is difficult to function without it. For example, in large systems, the expense of collecting and keeping source code coverage information is prohibitively expensive, and the coverage information requires updating as the source code evolves. Unlike WTCP, the BTCP technique simply requires information on the current execution outcomes or the test cases themselves. Furthermore, because the BTCP approach supports the sequencing of freshly created test cases, it is more extensively applicable. Here, we discuss BTCP techniques. In our study, BTCP techniques are used for comparison with KL-TCP.

Ledru et al. [9] treated the test cases as a string, calculated the differences in textual information between the test cases, and then used an adaptive random prioritization to prioritize the test cases. This approach works even worse than random prioritization when it encounters some inputs which are unusually long or short, or even empty. To address this problem, chen et al. [21] utilized test case similarity and the K-medoids approach to further categorize the test cases, reducing the impact of extreme inputs. Thomas et al. [22] applied topic modeling(a text analysis technique) to test case linguistic data by abstracting test cases into topics and then using the topics to calculate the similarity between pairs of test cases (measured by Manhattan distance), prioritizing the test cases using an adaptive random strategy. This method assumes that if two test cases cover the same topic, they have functional similarity and will detect the same faults. Miranda et al. [23] introduced the FAST family of TCP strategies, which address the problem of a high number of test cases by utilizing algorithms typically used in the Big Data sector to locate comparable things. To find a balance between economy and precision, the FAST approach employs a function. It is capable of prioritizing one million test cases in 20 minutes while maintaining its efficacy.

### B. Log Parsing

The primary objective of log parsing is to parse unstructured log statements into structured information, i.e., to segregate the invariant and variable parts of the log statement part. Mainstream log parsing is categorized into frequent pattern mining, machine learning and heuristic rules. Typical log parsing methods that rely on frequent pattern mining are LogCluster [24], which has relatively high parsing efficiency. However, it is not sensitive to the change of word position, and faces challenges in meeting the demand for real-time parsing in modern systems. LogMine [25] utilizes clustering algorithms in machine learning for log parsing. Drain3 [26] uses a tunable tree structure for matching static templates of logs for online log parsing, which is based on fast matching of heuristic rules for efficient online log parsing.

### C. Diversity based TCP

Test cases exhibiting similar attributes tend to possess comparable fault detection capabilities. The core idea of the diversity based TCP approach is to minimize the similarity among test cases by calculating the distance between them based on a distance metric. The distance function include Hamming distance, Jaccard Index, Euclidean distance, and Edit distance. Several researchers have utilized clustering-based methods in TCP [7], [21] to increase the diversity of test case sequences. Clustering-based methods divide test cases into groups or clusters based on a similarity function, so that test cases in the same group exhibit a high degree of similarity.

## III. Proposed Approach

The framework diagram of our approach is shown in Fig. 1 and consists of three main stages. In the first stages, the logs generated by the test cases are analyzed and the test cases are represented as vectors. In the second stages, similar test cases are clustered into one group using $k$-means clustering technique. In the third stages, the items within each cluster are internally sorted. Next, test cases are sequentially selected from each cluster and aggregated into the result set based on the inter-cluster sorting method.

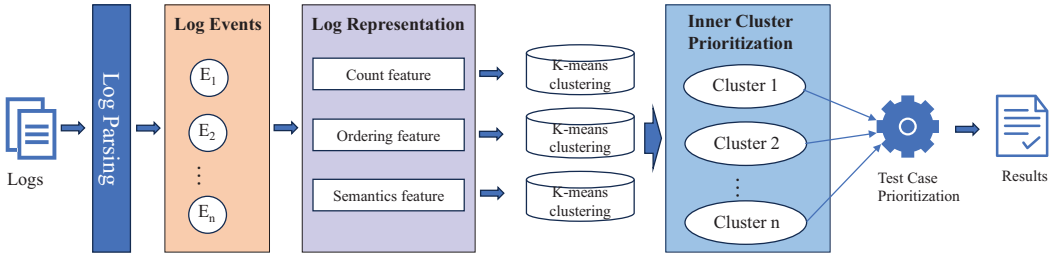


Fig. 1. Overview of our KL-TCP framework

### A. Log Analysis

*1) Log Parsing:* The series of log messages generated by test case execution is unstructured data that contains log parameters such as IP addresses, which often hinders log analysis. We use the Drain3 [26] log parsing tool to extract structured log templates from log messages with logging level *ALL*, which has been shown to work better for TCP [10].

*2) Log Representation:* Depending on the different features of the log templates, test cases can be represented as three different vectors. For a sequence $L$ containing $n$ log templates $E = \{e_1, e_2, ..., e_n\}$:

Depending on the count feature of log templates, the sequence of logs corresponding to the test cases can be turned into an $n$-dimensional vector $C = \{c_1, c_2, ..., c_n\}$, $c_i$ is the number of times the $i$-th log template occurs in the sequence. Instead of using individual log template, the ordering feature for log templates extracts sub-sequences from $L$ via a sliding window $N$, then counts the number of occurrences of the sub-sequences. This method transforms the log sequence into an $m$-dimensional vector, denoted as $O = \{o_1, o_2, ..., o_m\}$, where $m$ refers to the total number of distinct sub-sequences, and $o_i$ refers to the number of times the $i$-th sub-sequences occurs in $L$. Depending on the semantic features of log templates, the sequence of logs corresponding to the test cases can be turned into an vectors $S = \{s_1, s_2, ..., s_n\}$. The FastText algorithm obtains word vectors for each word in the log template, then aggregates the word vectors into log template vectors, and finally aggregates them into $S$.

### B. Proposed Clustering Method

Based on the fact that test cases with similar attributes may detect same faults. We hypothesize that test cases with similar log information will also have similar fault detection capabilities. To group similar test cases we use $k$-means clustering algorithm. $K$-means is essentially a data partitioning method based on the Euclidean distance metric, which has the advantages of efficient scalability, with a near-linear computational complexity.

As shown in Fig. 1, $k$-means clustering receives the set of test cases, each of test case is given a feature vector as input, and returns multiple subsets after clustering them separately. We use the log vectors generated by the three log representations in the previous section as the feature vectors of the test cases, which results in three clustering results.

### C. Proposed Test Case Prioritization Method

*1) Inner cluster Prioritization:* First, the test cases within the clusters are prioritized. According to previous studies [10], the additional and ARP policies will have better performance compared to the total policy, so we choose the additional and ARP strategies for inner cluster sorting. Specifically, we use the ARP strategy to internally prioritize the clustering results of the three log features vectors, and the additional strategy is only applied to the semantic feature vectors, since this feature has no coverage information.

*2) Inter cluster prioritization:* After the internal prioritization of test cases for each cluster is complete, we take turns selecting a test case from each cluster according to its internal priority. The selected test cases are then sorted using the total prioritization strategy, because the coverage of test cases selected from each cluster is already different. Similar to internal prioritization, for semantic feature vectors of test cases, only ARP prioritization strategy can be used.

In this approach, the test cases at the beginning of the sequence are made to belong to various clusters, which are mostly used to detect different faults. We illustrate the test case prioritization process with an example shown in Fig. 2. Given a set $T$ of test cases to be sequenced, it contains 9 test cases. The test cases are first clustered using $k$-means, assuming $k$=3, which returns 3 clusters. After the inner cluster sorting is done, the test cases are iteratively selected from each cluster. For example, $tc_9$, $tc_3$, and $tc_5$ are prioritized first in all three clusters, and these three test cases are prioritized and placed in the result sequence. Repeat this process until that all the test cases have been added to the total prioritized sequence.

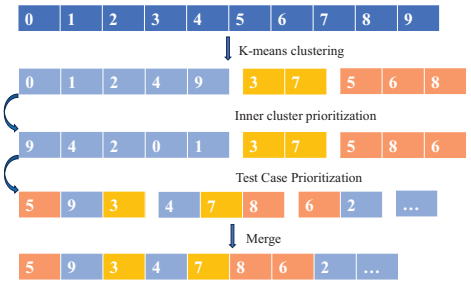


Fig. 2. Example of prioritize test cases

## IV. Empirical Study

This section describes the details of the experimental setup, including experimental subjects, experimental procedure, evaluation metrics, and parameter configuration.

### A. Experimental Subjects

To be able to implement TCP using log information, we use the log dataset implemented by chen et al. [10] as experimental subjects, which contains 10 Java projects and generates logs based on the Log4j or Logback packages (the log level is *ALL*). TABLE I presents the details of these projects, including the number of test cases, the number of log event categories generated during test case execution. It should be noted that there are some test cases in this dataset that do not generate logs, which we also list in the table, while these test cases are not considered during the experiment.

Existing studies [27] have shown that mutation faults are applicable to TCP, and we use mutation faults to evaluate the effectiveness of KL-TCP. For each project, all mutation operators provided by the PIT[1] tool were used to generate

[1]http://pitest.org

TABLE I
EXPERIMENTAL SUBJECTS INFORMATION

| Project | Module | Test classes | With log | Log events |
|---|---|---|---|---|
| Activemq | Activemq-amqp | 85 | 61 | 2634 |
| Airavata | Registry-core | 32 | 31 | 236 |
| Blueflood | Blueflood-http | 25 | 14 | 240 |
| Dubbo | Dubbo-config-spring | 22 | 19 | 264 |
| Flume | Flume-ng-core | 31 | 16 | 76 |
| Kylin | Core-metadata | 54 | 28 | 46 |
| ORCID-Source | Orcid-core | 240 | 72 | 982 |
| Shiro | Core | 64 | 34 | 126 |
| Webdrivermanager | Single-module | 83 | 19 | 562 |
| Wicket | Wicket-core | 455 | 395 | 298 |

mutation faults. After filtering out duplicate and unkillable mutation faults, 500 mutation faults are randomly selected from the remaining set, with every 5 mutation faults constituting a faulty version of the project, totaling 100. For projects having less than 500 mutation faults, the number of faulty versions will also be lower.

### B. Experimental Procedure

We compared KL-TCP with the 7 BTCP techniques in LogTCP. The seven technologies are LogTCP$^{total}_{count}$, LogTCP$^{additional}_{count}$, LogTCP$^{arp}_{count}$, LogTCP$^{total}_{ordering}$, LogTCP$^{additional}_{ordering}$, LogTCP$^{arp}_{ordering}$ and LogTCP$^{arp}_{semantics}$. The subscript is the log template feature for the test cases, and the superscript is their ordering strategy.

As presented in Section III, we constructed 5 BTCP techniques based on log and test case diversity by combining different log template features with $k$-means clustering, as shown in TABLE II. The four columns in the table are the abbreviation of the KL-TCP technique, the feature for the log templates, the inner cluster prioritization strategy, and the prioritization strategy for the iterative selection of test cases.

TABLE II
FIVE KL-TCP TECHNIQUES

| Technique | Log feature | Inner cluster prioritization | Prioritization |
|---|---|---|---|
| KL-TCP$_1$ | count | additional | total |
| KL-TCP$_2$ | count | ARP | total |
| KL-TCP$_3$ | ordering | additional | total |
| KL-TCP$_4$ | ordering | ARP | total |
| KL-TCP$_5$ | semantics | ARP | ARP |

### C. Evaluation Metrics

We use the APFD and the location of the first failure to measure the effectiveness of the KL-TCP techniques.

**APFD** Average Percentage of Faults Detected (APFD) [2], [6], [13] is the percentage of defects detected during the execution of the test suite. It is commonly applied in evaluating TCP techniques and is calculated as:

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{mn} + \frac{1}{2n} \quad (1)$$

Where $TF_i$ denotes the position in the prioritized sequence of the first test case that reveals the $i$-th fault, $m$ is the total number of detected faults; $n$ represents the number of test cases. The more test cases that detect different faults are ranked first, the larger the APFD value is, which indicates that the speed of detecting faults in the prioritized sequence is faster and the corresponding TCP technique is more effective.

**First-Fail** means that the first test case that finds a fault in the execution sequence [7], [21]. The purpose of TCP is to detect faults at the earliest possible stage, so that if the developer receives timely feedback that there is a bug in the project's tests (depending on the location of the first-fail test case), the code can be thoroughly reviewed and modified at a much earlier stage. Lower first-fail value, indicating better prioritization effectiveness.

### D. Implementation and Configurations

The parameters in LogTCP remain the same as in the original work. When extracting ordering features, we set the length of the sub-sequence to 5, and when extracting semantic features, we set the dimension of the word vector to 50. The parameter $k$ in KL-TCP represents the number of clusters, which affects the performance and efficiency of our algorithm. After experimental analysis, we found that appropriate results were achieved with $k$=5, which was set to a fixed value. The experimental scripts for KL-TCP are implemented in Python 3. Each TCP technique is repeated 10 times and the final results are averaged. Our experiments are conducted on the computer with the configuration of AMD Ryzen 7-5800H CPU(3.20GHz), 16G memory, and Ubuntu 18.04 operating system.

## V. RESULT AND ANALYSIS

In this section we present and analyze the experimental results. The Comparison results of each prioritization technique in APFD are shown in TABLE III. The best APFD results for each subject are shown in bold, and the last column shows the average result for all subjects in term of APFD. From TABLE III, it can be seen that KL-TCP gives the best results on almost all projects. Although LogTCP method based on the Ordering representation would be better on Activemq, Airavata and Kylin projects, the maximum difference is only 0.0162. For APFD, KL-TCP$_1$ achieved the best results on 6 subjects, KL-TCP$_2$ achieved the best results on 4 subjects, KL-TCP$_3$ achieved the best results on 2 subjects, KL-TCP$_4$ achieved the best results on 5 subjects, and KL-TCP$_5$ had the highest APFD on all subjects. In terms of the average APFD for the 10 subjects, the KL-TCP technique reaches 0.7671 to 0.7709, while the LogTCP technique reaches 0.6794 to 0.7171. This means that KL-TCP is able to take advantage of the diversity of test cases to achieve better permance.

TABLE III
COMPARISON RESULTS BETWEEN KL-TCP AND LOGTCP IN APFD

| Representation | Approach | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Average |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Count** | $\text{LogTCP}^{total}_{count}$ | 0.7206 | 0.7298 | 0.6802 | 0.7253 | 0.6774 | 0.7049 | 0.5576 | 0.7959 | 0.7008 | 0.8786 | 0.7171 |
| | $\text{LogTCP}^{additional}_{count}$ | 0.6954 | 0.7187 | 0.7373 | 0.6684 | 0.6226 | 0.7476 | 0.5120 | 0.7743 | 0.7083 | 0.8848 | 0.7069 |
| | $\text{LogTCP}^{arp}_{count}$ | 0.6630 | 0.7232 | 0.6881 | 0.6705 | 0.5668 | 0.6566 | 0.5414 | 0.7311 | 0.6421 | 0.9015 | 0.6794 |
| | $\text{KL-TCP}_1$ | 0.7269 | 0.7289 | **0.8008** | **0.8130** | **0.6793** | **0.7566** | 0.6598 | **0.8503** | **0.7233** | 0.9327 | 0.7671 |
| | $\text{KL-TCP}_2$ | **0.7351** | **0.7307** | 0.7976 | 0.8095 | 0.6764 | 0.7362 | **0.6841** | 0.8467 | 0.7203 | **0.9388** | **0.7675** |
| **Ordering** | $\text{LogTCP}^{total}_{ordering}$ | **0.7407** | 0.6988 | 0.4738 | 0.6249 | 0.5505 | 0.7320 | 0.6513 | 0.7712 | 0.7000 | 0.9020 | 0.6845 |
| | $\text{LogTCP}^{additional}_{ordering}$ | 0.7196 | **0.7439** | 0.6024 | 0.6881 | 0.5207 | **0.7675** | 0.5683 | 0.7958 | 0.7143 | 0.8924 | 0.7013 |
| | $\text{LogTCP}^{arp}_{ordering}$ | 0.7145 | 0.7151 | 0.5516 | 0.7042 | 0.6582 | 0.6963 | 0.5870 | 0.8043 | 0.6609 | 0.9024 | 0.6994 |
| | $\text{KL-TCP}_3$ | 0.7245 | 0.7356 | **0.7992** | 0.8053 | **0.6793** | 0.7500 | 0.6866 | 0.8563 | 0.7263 | 0.9368 | 0.7699 |
| | $\text{KL-TCP}_4$ | 0.7150 | 0.7301 | 0.7944 | **0.8081** | 0.6793 | 0.7418 | **0.7140** | **0.8613** | **0.7248** | **0.9374** | **0.7706** |
| **Semantics** | $\text{LogTCP}^{arp}_{semantics}$ | 0.7440 | 0.7237 | 0.6484 | 0.7218 | 0.5938 | 0.7088 | 0.5506 | 0.7474 | 0.6203 | 0.8954 | 0.6954 |
| | $\text{KL-TCP}_5$ | **0.7468** | **0.7356** | **0.7500** | **0.8389** | **0.6774** | **0.7415** | **0.6984** | **0.8510** | **0.6947** | **0.9203** | **0.7654** |

TABLE IV
COMPARISON RESULTS BETWEEN KL-TCP AND LOGTCP IN FIRST-FAIL

| Representation | Approach | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Average |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Count** | $\text{LogTCP}^{total}_{count}$ | 3.25 | **1.14** | 1.44 | **1.07** | 3.46 | 3.00 | 7.91 | 2.85 | 1.14 | 1.94 | 2.72 |
| | $\text{LogTCP}^{additional}_{count}$ | 3.50 | 1.19 | 1.89 | 1.13 | 1.62 | 1.98 | 12.55 | 3.11 | 1.93 | 3.00 | 3.19 |
| | $\text{LogTCP}^{arp}_{count}$ | 4.00 | 1.95 | 1.89 | 2.20 | 3.46 | 2.88 | 12.60 | 1.72 | 2.07 | 4.11 | 3.69 |
| | $\text{KL-TCP}_1$ | 3.63 | 1.71 | **1.00** | 1.47 | **1.54** | **1.62** | **4.67** | **1.13** | **1.07** | **1.87** | **1.97** |
| | $\text{KL-TCP}_2$ | **3.13** | 1.62 | **1.00** | 1.47 | 1.54 | 1.79 | 5.10 | **1.13** | **1.07** | 2.05 | 1.99 |
| **Ordering** | $\text{LogTCP}^{total}_{ordering}$ | **2.19** | 1.86 | 2.33 | 1.73 | 2.77 | 2.07 | 20.02 | 3.17 | 2.00 | 2.58 | 4.07 |
| | $\text{LogTCP}^{additional}_{ordering}$ | 5.06 | **1.38** | 2.11 | 2.47 | 1.92 | 2.71 | 9.55 | 3.89 | 1.71 | 2.90 | 3.37 |
| | $\text{LogTCP}^{arp}_{ordering}$ | 5.03 | 1.43 | 1.44 | **1.07** | 4.23 | 2.55 | 12.47 | 1.76 | 2.07 | 1.54 | 3.36 |
| | $\text{KL-TCP}_3$ | 2.97 | 1.62 | **1.00** | 1.47 | **1.46** | **1.62** | **4.07** | **1.13** | **1.07** | 2.16 | **1.86** |
| | $\text{KL-TCP}_4$ | 2.84 | 1.57 | **1.00** | 1.47 | **1.46** | 1.64 | 5.17 | 1.15 | **1.07** | **2.06** | 1.94 |
| **Semantics** | $\text{LogTCP}^{arp}_{semantics}$ | 3.88 | 2.57 | 1.33 | 1.13 | 3.38 | 2.26 | 18.40 | 1.26 | 1.29 | 1.94 | 3.74 |
| | $\text{KL-TCP}_5$ | **2.22** | **1.95** | **1.00** | 1.00 | **1.62** | **2.00** | **4.41** | **1.15** | **1.21** | **1.05** | **1.76** |

Among the three features of log templates, KL-TCP possesses better APFD on both semantic and count features. The instability in the ordering features' effect might stem from their dimensional complexity. In the ordering feature vector of test case, its dimension can reach 10,000, significantly impacting the clustering effect. The execution of a test case often involves a series of method calls, semantic features can effectively capture this execution information for good performance, and the count features just calculate the number of each log template in the sequence.

We also compared with LogTCP on the first-fail metric, and the results are shown in TABLE IV, with bolded values being optimal. As can be seen from TABLE IV, KL-TCP has smaller first-fail values on the rest of the subjects except on Activemq, Airavata and Dubbo projects, especially on the ORCID-Source subjects, where the gap between KL-TCP and LogTCP technology is significant. In the average first-fail value of 10 subjects, KL-TCP can reach the fastest value of 1.76, while LogTCP can only reach 2.72. This suggests that KL-TCP is effective in advancing the ordering position of the first test case that reveals a fault, allowing developers to find the fault earlier.

## VI. THREATS TO VALIDITY

Internal validity threats are mostly caused by probable mistakes in algorithm implementation. In order to address this issue, this work directly employs the LogTCP implementation. KL-TCP builds the model with the well-known open-source machine learning toolkit Scikit-Learn and runs several tests to limit the influence of the algorithm's unpredictability. For a fair comparison, the log characteristics utilized for log analysis are likewise aligned with LogTCP.

The 10 subjects and faults employed pose the greatest danger to external validity. Despite the fact that various subjects serve distinct purposes, the number of test instances is limited. To further reduce this hazard, we will repeat our research on a larger dataset with a greater number of test cases in the future. In terms of fault types, we utilized mutation faults for our evaluation, and in the future, we will use genuine faults in software development to assess the efficiency of the TCP approach in reducing such risks.

## VII. Conclusion

This paper proposes an approach that addresses the challenges of BTCP by integrating the concepts of test case diversity and log analysis. Specifically, we extract log features generated during test case execution using log parsing and representation. Three strategies are applied to generate log vectors: count feature, ordering feature, and semantic feature. Subsequently, $k$-means clustering is employed to diversify test cases. We conduct an empirical study on 10 open-source Java projects from the GitHub, employing mutation faults to evaluate the performance of the proposed KL-TCP and LogTCP techniques. The evaluation results reveal that the proposed KL-TCP technique outperforms the existing LogTCP method in terms of average fault detection rate and fault detection speed. In the future, we will further investigate the impact of KL-TCP in larger projects with more test cases, and collect real failures in regression testing for evaluation. Moreover, we can enhance the effectiveness of TCP by integrating various types of information, such as history records and test case descriptions from regression testing.

## Acknowledgment

This work was partially supported by the Key Research and Development Program of Hubei Province (Grant No. 2022BAA050, 2020AAA001), and the Natural Science Foundation of Chongqing (Grant No. cstc2021jcyj-msxmX1146).

## References

[1] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[2] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. Le Traon, "Comparing white-box and black-box test prioritization," in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 523–534.

[3] J. Zhou and D. Hao, "Impact of static and dynamic coverage on test-case prioritization: An empirical study," in *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2017, pp. 392–394.

[4] F. Altiero, A. Corazza, S. Di Martino, A. Peron, and L. L. L. Starace, "Inspecting code churns to prioritize test cases," in *IFIP International Conference on Testing Software and Systems*. Springer, 2020, pp. 272–285.

[5] M. A. Rahman, M. A. Hasan, and M. S. Siddik, "Prioritizing dissimilar test cases in regression testing using historical failure data," *Int. Journal of Computer Applications*, vol. 180, no. 14, pp. 1–8, 2018.

[6] M. Mahdieh, S.-H. Mirian-Hosseinabadi, K. Etemadi, A. Nosrati, and S. Jalali, "Incorporating fault-proneness estimations into coverage-based test case prioritization methods," *Information and Software Technology*, vol. 121, p. 106269, 2020.

[7] M. Mahdieh, S.-H. Mirian-Hosseinabadi, and M. Mahdieh, "Test case prioritization using test case diversification and fault-proneness estimations," *Automated Software Engineering*, vol. 29, no. 2, p. 50, 2022.

[8] J. Chen, "Learning to accelerate compiler testing," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 472–475.

[9] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran, "Prioritizing test cases with string distances," *Automated Software Engineering*, vol. 19, pp. 65–95, 2012.

[10] Z. Chen, J. Chen, W. Wang, J. Zhou, M. Wang, X. Chen, S. Zhou, and J. Wang, "Exploring better black-box test case prioritization via log analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 3, pp. 1–32, 2023.

[11] Z. Yu, F. Fahid, T. Menzies, G. Rothermel, K. Patrick, and S. Cherian, "Terminator: Better automated ui test case prioritization," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 883–894.

[12] M. J. Arafeen and H. Do, "Test case prioritization using requirements-based clustering," in *2013 IEEE sixth international conference on software testing, verification and validation*. IEEE, 2013, pp. 312–321.

[13] J. Chen, Y. Lou, L. Zhang, J. Zhou, X. Wang, D. Hao, and L. Zhang, "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*, 2018, pp. 656–667.

[14] A. S. Yaraghi, M. Bagherzadeh, N. Kahani, and L. C. Briand, "Scalable and accurate test case prioritization in continuous integration contexts," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1615–1639, 2022.

[15] A. Ramírez, R. Feldt, and J. R. Romero, "A taxonomy of information attributes for test case prioritisation: Applicability, machine learning," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 1, pp. 1–42, 2023.

[16] D. Leon and A. Podgurski, "A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases," in *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.* IEEE, 2003, pp. 442–453.

[17] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 201–212.

[18] R. Carlson, H. Do, and A. Denton, "A clustering approach to improving test case prioritization: An industrial case study," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 382–391.

[19] C. Fang, Z. Chen, K. Wu, and Z. Zhao, "Similarity-based test case prioritization using ordered sequences of program entities," *Software Quality Journal*, vol. 22, pp. 335–361, 2014.

[20] B. Jiang, Z. Zhang, W. K. Chan, and T. Tse, "Adaptive random test case prioritization," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 233–244.

[21] J. Chen, Y. Gu, S. Cai, H. Chen, and J. Chen, "A novel test case prioritization approach for black-box testing based on k-medoids clustering," *Journal of Software: Evolution and Process*, p. e2565, 2023.

[22] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, vol. 19, pp. 182–212, 2014.

[23] B. Miranda, E. Cruciani, R. Verdecchia, and A. Bertolino, "Fast approaches to scalable similarity-based test case prioritization," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 222–232.

[24] R. Vaarandi and M. Pihelgas, "Logcluster-a data clustering and pattern mining algorithm for event logs," in *2015 11th International conference on network and service management (CNSM)*. IEEE, 2015, pp. 1–7.

[25] H. Hamooni, B. Debnath, J. Xu, H. Zhang, G. Jiang, and A. Mueen, "Logmine: Fast pattern recognition for log analytics," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, 2016, pp. 1573–1582.

[26] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An online log parsing approach with fixed depth tree," in *2017 IEEE international conference on web services (ICWS)*. IEEE, 2017, pp. 33–40.

[27] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang, "How does regression test prioritization perform in real-world software evolution?" in *Proceedings of the 38th International Conference on Software Engineering*, 2016, pp. 535–546.