Received: 8 September 2022 | Revised: 21 December 2022 | Accepted: 18 February 2023

DOI: 10.1002/smr.2565

RESEARCH ARTICLE - EMPIRICAL

Journal of Software: Evolution and Process WILEY

A novel test case prioritization approach for black-box testing based on K-medoids clustering

Jinfu Chen[1,2] | Yuechao Gu[1,2] | Saihua Cai[1,2] | Haibo Chen[1,2] | Jingyi Chen[1,2]

[1]Jiangsu University, Zhenjiang, China

[2]Jiangsu Key Laboratory of Security Technology for Industrial Cyberspace, Zhenjiang, China

Correspondence
Saihua Cai, School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang 212013, China.
Email: caisaih@ujs.edu.cn

Funding information
National Key R&D Program of China, Grant/Award Number: 2020YFB1005500; National Natural Science Foundation of China (NSFC), Grant/Award Numbers: 62172194, 62202206, U1836116; Leading-edge Technology Program of Jiangsu Natural Science Foundation, Grant/Award Number: BK20202001; Natural Science Foundation of Jiangsu Province, Grant/Award Number: BK20220515; China Postdoctoral Science Foundation, Grant/Award Number: 2021M691310; Postdoctoral Science Foundation of Jiangsu Province, Grant/Award Number: 2021K636C; Qinglan Project of Jiangsu Province

Abstract

Regression testing is an essential and expensive process in software testing. However, there may be insufficient resources for the execution of all test cases during regression testing. Test case prioritization (TCP) techniques improve the efficiency of regression testing by adjusting the test case execution sequence. Traditional TCP techniques usually rely on the historical execution information of the software under test for more efficient results. String distance-based TCP (SD-TCP) avoids these limitations; it uses only the textual difference information of the test cases themselves for prioritization. However, the time overhead on the sorting process of this method is not ideal, and the extreme test case inputs have an impact on the stability of the method. To address these problems, we propose a novel test case prioritization strategy, it first classifies the test cases more finely using the K-medoids algorithm and then transforms the set into subsequences and improves the early diversity by greedy sorting within clusters. Finally, the test cases are selected through a polling strategy to compose the execution sequence. Extensive experimental results demonstrate that the proposed approach outperforms SD-TCP in better time efficiency on test case prioritization; it also has a higher average percentage of fault detected (APFD) value than random prioritization (RP) and SD-TCP.

KEYWORDS
cluster analysis, greedy algorithm, regression testing, test case prioritization

1 | INTRODUCTION

In software development, regression testing(RT)[1] is often used to ensure the safety and reliability of the software. It is one of the essential components of software engineering and is very critical for software security. RT retests the software after an update iteration to ensure that no new bugs would be introduced in the latest software version. However, frequent regression testing entails high testing costs. To reduce the overall testing time overhead, test suite reduction (TSR)[2] and test case prioritization (TCP)[3] are commonly used in regression testing. The TSR technique aims to discard the execution of some unimportant test cases after careful selection. The TCP technique aims to find the best sequence of test case execution and then prioritizes the execution of important test cases to maximize the tester's effort.

Current test case prioritization methods can be divided into black-box test case prioritization and white-box test case prioritization based on the necessary information.[4] Among these two test case prioritization methods, white-box test case prioritization is more suitable for regression testing and has better results than black-box methods. But it is usually more complicated, and there may be some technical difficulties in the practical application. In addition, this technology requires some information from previous versions (such as source code,[5–7] coverage information,[8,9] and fault severity[10,11]). These white-box techniques usually do not avoid the question: Is the historical execution information of test cases

available? Collecting this information is also usually costly, and the information is updated as each version of the software iterates. If the software changes too much, this information as a guide for the latest version may not be more useful and even may result in inaccurate results.[12] Compared with white-box methods, static black-box methods[13–15] do not rely on the historical execution information or knowledge of the software but only need current or self-existing information.[16,17]

Therefore, many related techniques have been proposed for the advantages of black-box TCP. For example, Yves Ledru et al[18] proposed a string distance-based test case prioritization approach (SD-TCP) by exploiting the textual information differences in test cases. In SD-TCP, each test case is considered a string, and then a greedy strategy is used to improve the early diversity of the execution sequence, where the test cases are selected iteratively and one test case at a time with the greatest difference from the execution sequence is chosen as a candidate test case. With these strategies, the final execution sequence outperforms the random ordering and the original sequence in detecting the defects.

Although black-box methods are easy to use and only require little information about the software to be tested, they also have some drawbacks. For example, SD-TCP does not perform the sequence well in some cases and is even a little weaker than random priority (RP). One possible reason is that SD-TCP is usually given a higher priority when faced with some particularly long or very short, or even empty inputs. In a real test scenario, these test cases, which are essential for software testing, may be rejected by the initial input judgment of the software. However, the SD-TCP would place almost all extreme inputs at the top of the execution sequence, which can potentially degrade the testing effectiveness. In addition, the computation of character differences of the test cases in SD-TCP needs for multiple iterations, which leads to the size of the test case set having a significant impact on the testing efficiency.

In this paper, based on black-box test case prioritization, the similarity of test cases combined with the K-medoids method is used to classify the test cases further. And then, the greedy algorithm is used to filter and merge the classified sets to ensure the diversity of test cases as early as possible in the execution sequence. The proposed method can reduce the time overhead further while ensuring the sorting effect. In addition, it also reduces the impact of extreme inputs on the execution sequence, thus improving the testing efficiency.

The main contributions of this paper are as follows:

- We apply a new K-medoids strategy on the test cases to effectively classify them.
- We propose a new black-box test case prioritization technology based on the improved K-medoids and greedy strategy to improve the efficiency of test case prioritization.
- We conduct extensive experiments on five classical datasets to compare the effectiveness and efficiency of our proposed KS-TCP method with two black-box TCP methods and RP, and the experimental results show that the KS-TCP method can find more errors faster.

The rest of the paper is organized as follows: Section 2 presents some background knowledge and related work. Section 3 presents the methodology. Section 4 describes the experimental setup. Section 5 shows the experimental results and evaluates the validity of the proposed method. Section 6 gives the conclusion.

2 | BACKGROUND AND RELATED WORK

This section describes some background information about test case prioritization and related work.

2.1 | Regression testing and test case prioritization

Regression testing[19] is an essential part of a complete test process in the software lifecycle, which is used to ensure that previously developed and tested software can continue to function properly after changes such as bug fixes, configuration changes, and software updates. In the practice, the number of test cases will be increasing, especially for the software products that are maintained and updated. For example, under the time constraint of software product launch, there may not be enough time to conduct the total amount of regression testing,[20] which prompts the non-full regression testing strategies. In recent years, some non-full regression testing strategies are adopted under time cost constraints: (1) TSR methods, they expect to recognize tedious examinations and wipe out test cases from a test suite execution with a particular objective, while the reduction is also known as "test suite minimization."[21] (2) Test case selection (TCS) methods, they likewise reduce the number of test cases and generally focus on the modified part of the software under test, which usually requires source code analysis.[22] (3) TCP methods, they intend to rank the test cases to accomplish an early improvement dependent on favored properties as well as enable a way to deal with executing profoundly critical test cases initially as per some measure and produce the ideal result.[23]

Both TSR and TCS techniques require a reduction of the overall number of test cases and are an utterly non-full testing strategy. The risk of missing tests is obvious with this strategy.[24] On the other hand, the TCP technique does not discard test cases but adjusts the execution order of

test cases. Therefore, full regression testing can still be performed when time conditions allow. In other words, TCP technology will be safer compared to TSR and TCS technology,[25] which causes the TCP techniques to be more popular in current regression testing research.[26]

The intent of test case prioritization technique is to rank the test cases to be tested once. All test cases are prioritized according to the needs of the tester, thus allowing certain important test cases to be tested as early as possible. This minimizes the loss of test cases that cannot be executed exhaustedly due to the sudden reduction in testing costs. Ideally, TCP can perfectly prioritize all test cases and generate an optimal execution sequence with the test results that meet the expectations of testers. Therefore, TCP's critical sequencing approach becomes very critical, and a wrong execution sequence can even reduce the tester's effort. The test case prioritization problem was first proposed by Rothermel,[27] and it was defined as follows:

Definition 1. Given a test set T, m is the set of total permutations of the sequence of test cases in the test set T. f is a fitness function, and the purpose of TCP is to find an execution sequence S, which is shown in Equation (1).

$$\forall S'(S' \in m)(S' \neq S)[f(S) \geq f(S')] \tag{1}$$

Based on the required information, the dynamic or static execution condition, and the presence or absence of test cases for execution, current test case prioritization technology can be divided into white-box TCP and black-box TCP. White-box TCP has a firm reliance on execution information, it is difficult to play a role in the situations where execution information is not available. For example, the cost of collecting and maintaining execution information is too high for an extensive system, and the execution information needs to be updated as the source code changes. In contrast to white-box TCP, black-box TCP technology requires only current or self-contained information and does not rely on the historical execution information or software knowledge. Moreover, black-box static TCP technique also can sort the newly generated test cases; thus, it has a broader scope of application.

2.2 | Cluster analysis and related approaches

Clustering analysis[28] is a data analysis technique, and it is used in many fields, such as machine learning and data mining. The clustering technique is also a kind of unsupervised learning. It is usually used in the preprocessing process of data to find the intrinsic connection among the data through comparing the similarity between them and thus reducing the high complexity of the data, which allows subsequent steps to process the data better. Current clustering algorithms are classified as follows[29]: (1) Division-based clustering methods; (2) hierarchy-based clustering methods; (3) density-based clustering methods; (4) grid-based clustering methods; and (5) model-based clustering methods. In test case prioritization techniques, the execution data of test cases can usually be grouped using cluster analysis, and then the grouped data are further used to guide the subsequent prioritization process.

Currently, there have been many test case prioritization techniques applying clustering algorithms. Chen et al[30] have combined two clustering algorithms to optimize the adaptive random sequence (ARS) method[29] for object-oriented software (OOS).[31] The test cases were classified using K-means and K-medoids techniques based on the number of objects of the software to be tested and the similarity of the method call sequences, respectively. This method effectively improves the application of test case prioritization techniques on OOS. However, the experimental objects and test cases are very specific, which causes less applicability to the actual software. Yoo et al[32] proposed a test case prioritization method based on hierarchical analysis; it mainly utilizes the analytic hierarchy process (AHP) method for decision making, but it requires a large number of pair-wise comparisons by testers. To solve this problem, Yoo combined the cohesive hierarchical clustering method; it categories the test cases using information from dynamic runs as criteria, which greatly reduces the high time overhead of pair-wise comparisons, and the experiments also confirm that this method is more effective than some coverage-based test case ranking methods. To improve the error detection rates of test case prioritization, Carlson et al[33] used Euclidean distance and agglomerated hierarchical clustering to classify test cases, and then the code coverage, code complexity, and historical error information were considered in the ranking process, while the historical error information and code complexity information were fused as a new metric. After sorting each cluster after clustering, the final execution sequence is merged by selecting the first element of each cluster. The empirical study of this method shows the effectiveness of the clustering algorithm for test case prioritization techniques. However, its experimental subjects need detailed data, including error messages from real users' feedback, which may have some limitations.

2.3 | String-distance test case prioritization

Yves Ledru[18] proposed a string-distance-based test case prioritization method, where the execution sequence is generated based on the textual differences between test cases. This method mainly exploits a greedy strategy[34]; that is, one test case at a time with the largest textual difference

from the sequence is selected and put into the sequence at a time. Thus, it improves the diversity of the early sequences. In this study, several distance measures were selected, such as Manhattan distance, Euclidean distance, Hamming distance, and Edit distance. Among them, the use of Manhattan distance could achieve relatively good results.

2.4 | Firefly algorithm

Many search-based techniques[35] have been proposed in recent years. Among them, some methods are applicable for black-box testing. Muhammad[36] applied the firefly algorithm (FA) to optimize the test case priority and proposed a TCP method based on the FA, where the fitness function was defined based on the similarity model. FA uses the distance of "term frequency inverse document frequency" (TF-IDF) to calculate the distance and weight. They use the FA method to sort the test cases using black-box information to obtain better APFD results in the experiments. However, this method has specific requirements on the content of test cases, and the description of the importance in original text is inadequate.

3 | PROPOSED METHOD

In this section, we present the details of black-box test case prioritization technique (called KS-TCP) based on K-medoids and similarity.

3.1 | Overall framework

In black-box static test case prioritization techniques, most of them use textual information differences as a metric between test cases, and the extreme inputs will receive higher scores in the calculation of textual information differences. The proposed KS-TCP method uses the clustering technique to group test cases, thus reducing the impact of extreme inputs on the test case prioritization strategy. And then, KS-TCP selects appropriate test cases from each group and uses the greedy strategy to add these test cases to execution sequence, which leads to the high priority characteristics of some extreme inputs being reduced. In addition, the proposed KS-TCP method replaces the overall calculation to group processing to solve the problem of high time overhead in SD-TCP method caused by repeatedly calculating the distance of individual elements from the overall set. Similar to SD-TCP, KS-TCP also uses a greedy algorithm to improve the early diversity of test cases.

The overall framework of KS-TCP is depicted in Figure 1. which consists of four steps: (1) First is the initialization phase, where the provided test case set is used as input, and the similarity between test cases is calculated using Manhattan distance to obtain the distance matrix. (2) Based

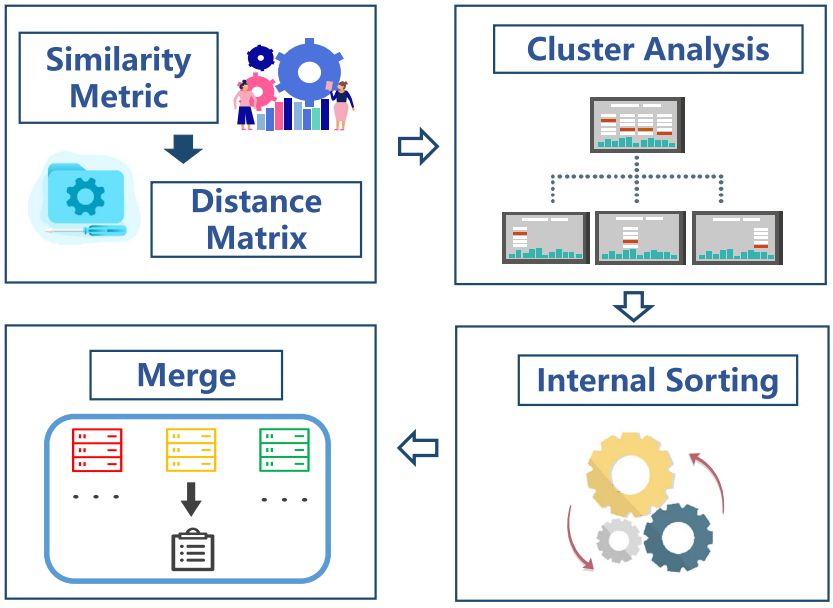

FIGURE 1 The framework of KS-TCP approach.

on the obtained distance matrix, the test case set is grouped using the modified K-medoids method. (3) Sort the subsets after step 2 using the min-max greedy strategy. (4) Finally, the first element of each subset is selected by polling to merge into a subsequence, and then they are added to the final execution sequence.

Algorithm 1 shows the pseudo-code of the proposed KS-TCP method, where the input is the original set of test cases, and the output is the final sorted execution sequence.

First, we initialize the original set of test cases and compute the similarity metric between test cases using Manhattan distance to obtain the distance matrix *distMatrix* (line 4). And then, the test cases are classified using a modified K-medoids method (line 5) to get the set of clusters, and each subset of clusters is cyclically sorted using a greedy strategy (line 6). Before the final execution sequence is completed, the first element *tc* is obtained from each sorted cluster and then added to the execution sequence *S* (lines 9–10), where the *tc* from the original set is removed (line 11). Finally, when all subset clusters are empty, that is, the complete execution sequence *S* is obtained, *S* is output as the final result. The *K_medoids* stage and *Greedy_Sort* stage are described later.

Algorithm 1 KS-TCP

Input: Original Test cases T
Output: Execution sequence S
1: Set $S = \phi$
2: Set $Clusters = \phi$
3: $K = version\_num$
4: $distMatrix = Distance\_calculation(T)$
5: $Clusters = K\_medoids(k, T, distMatrix)$
6: $Greedy\_Sort(Clusters, distMatrix)$
7: **while** $Len(S) \neq len(T)$ **do**
8: **for** clu in $Clusters$ **do**
9: Get first test case tc from clu
10: add tc to S
11: delete tc from clu
12: **end for**
13: **end while**
14: **return** S

3.2 | Similarity metric

Before ranking the test cases, the similarity metric between test cases needs to be calculated, where only the textual differences between test cases in the black-box environment are considered; that is, they are computed as strings. In the calculation process, we consider four commonly used distance metrics: Manhattan distance, Euclidean distance, Hamming distance, and edit distance.

Theoretically, edit distance may be most suitable for measuring the similarity of string distances. However, the time overhead on computing the edit distance is much longer than several other distances. When the set of test cases is very large, the computation time of edit distance can even exceed the execution time of all test cases. Therefore, the edit distance loses its meaning in ranking. In the SD-TCP study, some comparisons have been made to compare other three different distances. Eventually, the Manhattan distance may be more suitable for the comparison of test case distances. For this reason, we also use the Manhattan distance as the final similarity metric, and some experimental comparison also conducted in the end to verify the advantage of Manhattan distance.

The Manhattan distance indicates the sum of the absolute axis distances of two points on a standard coordinate system. The formula is shown in Equation (2).

$$\text{dist}_{man}(x,y) = \sum_{i=1}^{n} |x_i - y_i| \tag{2}$$

ASCII codes are usually used instead of characters for substitution in calculating the string distance. For example, the distance between *abc* and *abd* is 1 because the ASCII of *c* and *d* distance is 1.

3.3 | K-medoids

In this step, we use the K-medoids method to cluster the test cases into different clusters to make the variability of objects in other groups as large as possible, which aims to classify the test cases in a more granular way, therefore providing a basis for subsequent operations.

K-medoid is the classical division method in clustering methods, where medoid denotes the closest point in a set to the mean of that set. This method revolves around the medoid and assigns elements to the nearest medoid, and then iteratively updates the medoid values until the results converge.

The classical K-medoids method can only be used for numerical data, and the median is chosen as the medoid at each time. However, the test cases are not necessarily the numerical inputs, and the selection of medoids will affect the effect of clustering. If the test cases are abstracted as numbers, and the text features are extracted, much information will be lost. Therefore, the following strategy is proposed for the conditions possessed by the static test case. We choose the point with the smallest sum of distance with all other points as the medoid. The definition is similar to the Fermat point in geometry. We use an example shown in Figure 2 to illustrate this process. Suppose there is a test case set $T = \{tc1, tc2, tc3, tc4, tc5\}$, and the length of each edge indicates the similarity between them. The point with the smallest sum of distances is generally at the relative center in the two-dimensional condition, which is closest to the Fermat point. Therefore, in the example, *tc*2 will be selected as the medoid by the modified K-medoids method.

Algorithm 2 shows the pseudo-code of the K-medoids method after modifying the update point policy. This algorithm needs three inputs: The set of test cases *T*, the number of clusters *K*, and the distance matrix distMatix. And finally the finished clusters are processed as the output. In this algorithm, medoids are first initialized, and *K* initial test cases are randomly selected from *T* as medoids (line 2). And then each *tc* is assigned to the cluster represented by the nearest medoid (line 9). The medoids are updated for the next iteration, and in each cluster, the sum of the distances from each *tc* to the other *tc* in the current cluster is calculated (line 17). Save the minimum distance as *min_distance*(line 20), where the minimum value is continuously updated. Then the *tc* represented by the final value will be used as a candidate medoid(line21). After that, repeat the steps in lines 6–25 until the medoids no longer change (line 5). Finally, the clusters with completed clustering are output (line 27).

Algorithm 2 $K\_medoids(K, T, distMatrix)$

```
1: Set medoids = φ
2: Randomly select K initial tc as medoids
3: Set Clusters = [φ] * K
4: Set temp_medoids = φ
5: while temp_medoids ≠ medoids do
6:     temp_medoids = medoids
7:     medoids = φ
8:     for tc in T do
9:         Assign tc to the cluster defined by the closest medoid
10:    end for
11:    for clu in Clusters do
12:        min_distance = MAX_NUM
13:        new_medoid = NULL
14:        for tc_i in clu do
15:            distance_sum = 0
16:            for tc_j in clu do
17:                distance_sum += distMatrix[tc_i, tc_j]
18:            end for
19:            if distance_sum < min_distance then
20:                min_distance = distance_sum
21:                new_medoid = tc_i
22:            end if
23:            add new_medoid to medoids
24:        end for
25:    end for
26: end while
27: return Clusters
```

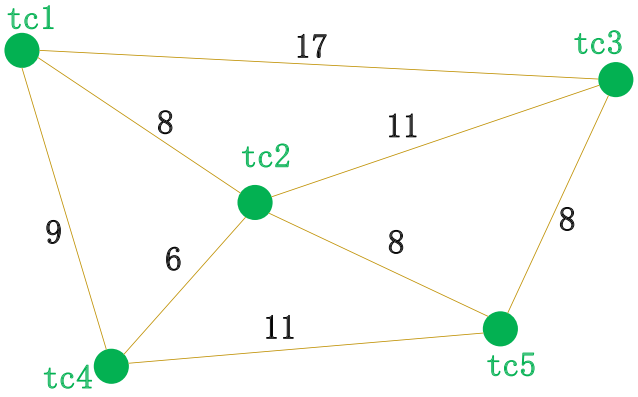

FIGURE 2 Two-dimensional representation of test case distances.

3.4 | Greedy algorithm

In this step, the main task is to sort the elements of the clusters for each cluster after classification.

The primary purpose of intracluster sorting is to make each cluster as a subsequence and no longer as a set, which can improve the early diversity of each subsequence. This process is a critical step in the whole sorting process and reflects our main idea of sorting; that is, we believe that the more dissimilar test cases are more different. Therefore, we want the test cases in front of the execution sequence to be as different as possible so that the test cases executed early in the test are as diverse as possible. This strategy has also been applied in a related study of adaptive random testing (ART), called the min-max strategy.

The main process is to select one test case at a time from the set to be ranked so that the test case has the lowest similarity with the final sequence, and then pick the next test case. In this case, the distance between a single test case and the final sequence is defined in Equation (3):

$$dd(t,T) = \min\{d(t,t_i) | t_i \in \mathrm{T} \cap t_i \neq t\} \tag{3}$$

where t denotes a single test case and T represents a test case set; that is, the distance between all the test cases in T and t is calculated, and the shortest distance among them is taken as the distance from t to T. That is, min-max.

Algorithm 3 shows the pseudo-code of greedy strategy sorting, where the input of this algorithm is the Clusters outputted by the K-medoids and the initially computed distance matrix distMatrix. In greedy strategy sorting, each cluster in Clusters needs to be sorted. Firstly, an initial *tc_r* is randomly selected in the cluster (line 1) and put into *temp_cluster* (lines 3-4) and the *tc_r* is removed from the original cluster (line5). And then, seek for a *tc* in the cluster with the maximum distance to *temp_cluster*, which is defined in Equation (3). Find a test case in the *temp_cluster* with the smallest distance from *tc* and use this distance as the distance from *temp_cluster* to *tc*. After that, move *tc* into *temp_cluster* and remove it from the cluster (lines 8 and 9). When all elements in a cluster are deleted, update *temp_cluster* to the cluster (line11). Finally, sort all the clusters and output the final clusters.

Algorithm 3 $Sort(Clusters, distMatrix)$

1: **for** clu in $Clusters$ **do**
2: $temp\_clu = \phi$
3: Randomly select tc_r in clu
4: add tc_r to $temp\_clu$
5: delete tc_r from clu
6: **while** clu is not empty **do**
7: Find $tc \in clu$ with the min distance from $temp\_clu$ depend on $distMatrix$
8: add tc to $temp\_clu$
9: delete tc from clu
10: **end while**
11: $clu = temp\_cluster$
12: **end for**
13: **return** $Clusters$

3.5 | An example

We use the following example shown in Figure 3 to demonstrate the main process of the proposed KS-TCP approach more intuitively. Given an original test case set *T*, it contains nine test cases. We first use the Manhattan distance metric to calculate the text similarity between the test cases and obtain the distance matrix *distMatrix* to support the later process. Assume that *k* is set to three, which means that the test cases should be divided into three classes.

Next, three test cases are randomly selected as medoids for the K-medoids process. According to the *Matrix*, the nine test cases are evenly divided into three groups in the ideal case. As shown in Figure 3, t_7, t_9, and t_5 are finally divided into group c_1; it means they are very similar, but the test cases in c1 are relatively different from those in c_2 or c_3.

And then, the test cases in each cluster are sorted using the greedy algorithm, and the order in each cluster is rearranged. In c_1, the final order is t_9, t_7, and t_5, which means that the difference between t_9 and $t7$ is greater than the difference between t_9 and t_5. The similarity between t_5 and t_9 is very high and the detection ability is similar, so the priority of one of them is lower in the greedy sorting process.

Finally, the first test case in each cluster is taken out and merged into a subset of test cases. When all groups are empty, all subsets are combined into a complete set of test cases. The original and variant versions of the program under test are executed using the execution sequence of the final output, and when the output does not match, a test case is considered to have found a bug in the variant version.

4 | EXPERIMENT SETUP

In this phase, we introduce the datasets, evaluation metrics, experimental flow, and experimental results.

4.1 | Dataset description

The used dataset in this experiment is the Siemens test suite,[37] which is one of the sets available in Software-artifact Infrastructure Repository (SIR). The Siemens test suite is one of the classic benchmark datasets in TCP research; it is consisted of several C programs, and each program is

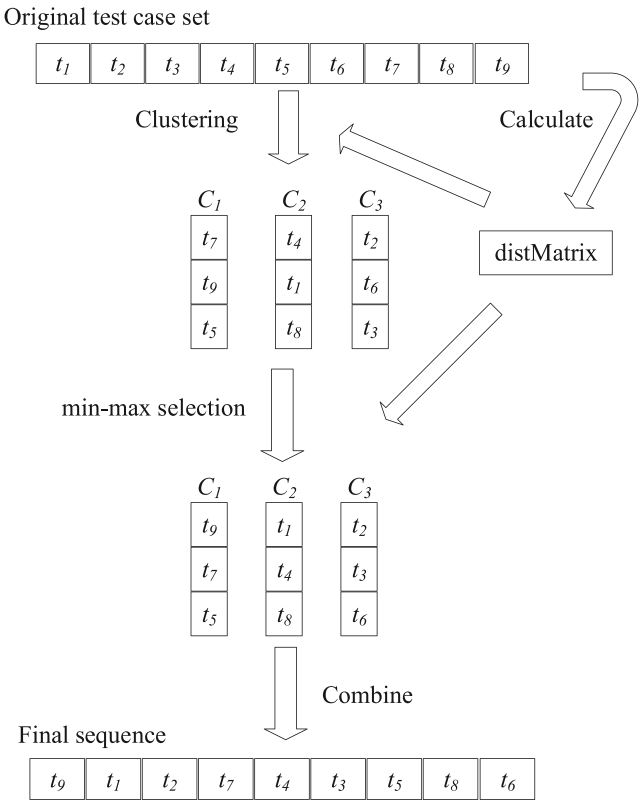

FIGURE 3 Example.

independent and small in program size, with subroutines of only 173–565 lines. Each C program contains dozens of variant versions and has a separate set of test cases. The specific of the Siemens data set is shown in Table 1.

4.2 | Experiment environment

The experiments are conducted using python3 running on Ubuntu 20.04. For the Siemens program, we used the GCC compiler of version 9.2.0. Each experiment is performed for more than 200 repetitions, and the average result is calculated as the final result.

Siemens programs are a classic benchmark dataset for various testing techniques, and each program has a correct original version and multiple manually implanted error versions. We take the input of test case in the correct version as the right result, and if the test case yields a different result in one of the incorrect versions, we then consider that the test case found an error in that version. This study uses the entire test pool as a test set to ensure that all bugs can be found in all error versions. Among them, because we failed to detect one of the variant versions using the given test suite in Schedule2, therefore we discard this version in our experiment.

For computing test case similarity, the test set can be extracted in the script that executes all test cases in the Siemens dataset, where each line is a test case. For example, in the Tcas program, the test cases are as follows.

TABLE 1 Siemens test suite.

Program	Description	No. of lines	No. of cases	No. of version	No. of functions
Replace	Pattern matching	18048	5542	32	21
Schedule	Priority schedulers	3708	2650	9	18
Schedule2	Priority schedulers	3740	2710	10	16
Tcas	Aircraft collision avoidance system	7093	1608	41	9
Totinfo	Computes statistics	12995	1052	23	7

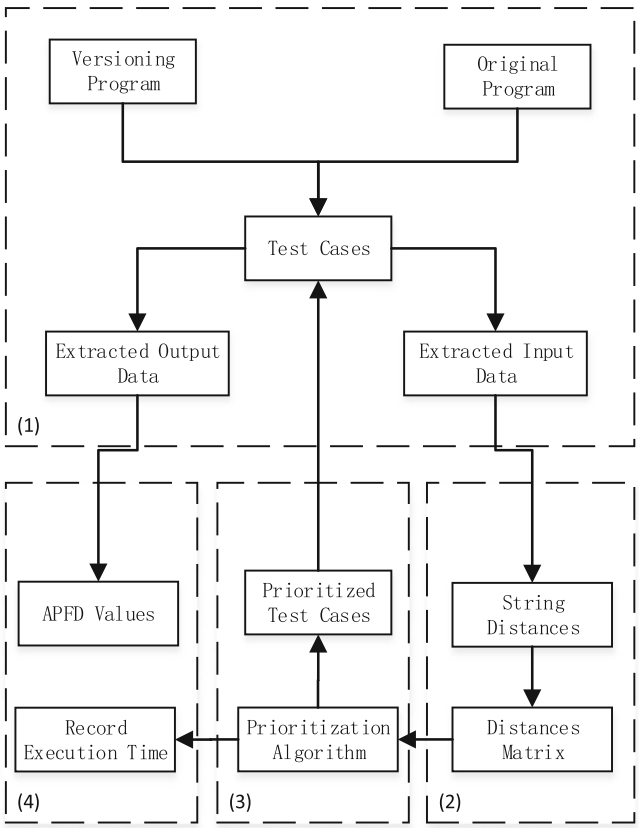

FIGURE 4 Experiment flow.

- 549 1 1 4398 133 1445 1 641 639 0 0 1
- 576 0 1 3469 183 381 2 641 501 1 0 1

Such test cases are easily converted to strings, but each test case may contain an input file in several other programs. The test cases in the execution script are shown as the path to that input file. In our experiments, we stitch the actual contents of the input file with the parameters into a single string, thereby calculating the similarity between test cases.

4.3 | Experiment flow

Figure 4 shows the key steps in the experimental flow in detail. It mainly includes four phases: (1) Test case extraction phase; (2) similarity calculation stage; (3) test case prioritization stage; (4) result analysis stage.

In Phase 1, all inputs are extracted from the program repository, and the inputs are collected. Each input is placed in a text file to facilitate the computation process in the next phase. And then, each test case is calculated using the Manhattan distance metric, and the results are saved in a distance matrix for the subsequent prioritization process.

In Phase 2, based on the distance matrix obtained in Phase 1, the original test case set is prioritized using the proposed test case prioritization method and various comparison methods, respectively, to get the execution sequence for each technique, and the execution sequence is saved into a text file for subsequent program execution.

In Phase 3, the execution sequences obtained above are executed in the original and variant programs, and their output results are saved. The output of original program is compared with that of the variant program, and the sequence number of test case that contains found fault and the order of found fault are recorded for the subsequent result evaluation process.

In Phase 4, the APFD of the test cases is calculated from the program execution records obtained above, and the execution sequences generated by each test case prioritization method are evaluated. Furthermore, the time obtained in the first phase will be recorded to introduce the evaluation of the execution efficiency of the test case prioritization methods.

4.4 | Evaluation metrics

APFD[38] is widely used as one of the primary metrics in this experiment to evaluate the effectiveness of TCP methods, which was first proposed by Rothermel et al.[39] APFD can represent the speed of defect discovery in an execution sequence, and the formula of APFD is shown in Equation (4). Given test set T containing n inputs, the test set T can find m defects of the software under test. For an arbitrary execution sequence S, the first test case that finds defect i will record its position in S and be noted as TF_i.

$$APFD = 1 - \frac{TF_i + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n} \tag{4}$$

It can be observed from Equation (4) that the APFD value ranges from 0 to 1 when the number of n is large. And the closer to 1, the defects are detected faster, and the more effective method is. On the contrary, the test case prioritization technique may have problems or the test case set cannot detect all software defects.

To demonstrate the calculation of APFD, the following example is used to explain the calculation process. As shown in Table 2, there are five errors in a program, and the execution sequence contains five test cases, each of which can detect one or more errors. During the test, all errors can be found when executing to TC3. Then the actual formula of APFD is 0.62, which is shown in Equation (5).

$$APFD = 1 - \frac{2 + 1 + 3 + 1 + 5}{5 \times 5} + \frac{1}{2 \times 5} \tag{5}$$

5 | RESULTS AND ANALYSIS

In the experiment, we compare the proposed KS-TCP method with the SD-TCP,[18] FA-TCP,[36] and random prioritization (RP) in effectiveness and efficiency. For the FA method, we used a custom method for the experiments because the original text lacked a description of TFIDF. The original

TABLE 2 Siemens test suite.

Test Case	Fault1	Fault2	Fault3	Fault4	Fault5
TC1		√		√	
TC2	√				
TC3			√		
TC4	√	√		√	
TC5		√		√	√

text used TFIDF for the test input, while the input of the Siemens program set may not be available for use, so we only used the string metric for its similarity analysis. The APFD results and execution times for each benchmark program are illustrated graphically, and the overall APFD for each benchmark program is presented in tabular form. In addition, we additionally recorded the location of the test case in the execution sequence where the error was first found and the result is presented in a table.

5.1 | The value of K

Since the *K* value in the K-medoids method will affect the final results of the proposed method, and too large or too small *K* value may lead to a decrease in the final performance and effectiveness, therefore, we further tested the impact of different *K* values in our experiments. We conducted experiments with the percentages of total test case set, but it is challenging to achieve optimal results with different rates simultaneously. We adopted an alternative approach that uses the number of program versions to be tested as the *K*-value. For this purpose, ideally, our algorithm would divide the test cases into groups of the same size, and the size of the group would be the *K*-value. In accurately, we want the group that is sorted first to represent the entire set of test cases, that is, each group is assigned different kinds of test cases. In this way, it is possible to test the errors in all program versions to be tested. At the same time, the number of versions is usually not so large to affect the stability of the algorithm. To verify our idea, the value of *K* is chosen as the number of erroneous versions of the benchmark program in the experiments.

5.2 | The APFD value

Figure 5A shows the APFD results for the Replace benchmark program, and the overall APFD value is shown in Table 3. Comparing the experimental results, it can be concluded that the proposed KS-TCP algorithm shows better APFD results in the Replace test procedure. Specifically, the results show that the KS-TCP algorithm is slightly better than other algorithms, and it has a higher average value than several other methods. However, due to the nature of the algorithm, SD-TCP and firefly have much lower standard deviation values, but KS-TCP algorithm is more stable than the RP. All other values have shown that KS-TCP is able to produce consistently higher APFDs, which indicates its applicability in the context of the Replace benchmark procedure. For the FA method, in our experimental setup without the influence of weights, the strategy of choosing unique shortest path of firefly flight is almost opposite to the idea of SD-TCP. Under our experimental setup, the result of FA is lower than RP, which is also found in the following experimental results.

Figure 5B shows the APFD results for the Schedule benchmark program, while Table 4 shows its overall APFD evaluation. Comparing the experiments shown in the table and graph, it can be concluded that KS-TCP shows better APFD results in the Schedule test procedure. The results show that the average APFD of KS-TCP is higher compared to other prioritization algorithms, which shows its applicability in the context of Schedule benchmark procedure.

Figure 5C shows the APFD results for the Schedule2 benchmark program, while Table 5 shows its overall APFD evaluation. Again comparing the tables and graphs, it can be known from the table and figure that the proposed KS-TCP also performs well on the Schedule2 benchmark program. Schedule2 has the same parameter input format as Schedule, so the situation on Schedule2 is similar to Schedule.

Tcas is the only numeric program in the Siemens datasets, and the test case consists of twelve numeric parameters. During the test case distance extraction, Tcas can calculate the Manhattan distance based on the difference of numerical values instead of calculating it as a string. Figure 5D shows the APFD results for the Tcas benchmark procedure, while Table 6 shows its overall APFD evaluation. The results show that KS-TCP provides better APFD results in Tcas applications than other prioritization algorithms, it indicates its applicability in the Tcas benchmark program.

The input of Totinfo is more specific and is done in a form of file, which results its extracted distances are not better applied to the prioritization method compared with other programs. Figure 5E shows the APFD results of Totinfo, while Table 7 shows its overall APFD evaluation. It can be found that KS-TCP still achieves better APFD results; it demonstrates its applicability in the context of the Totinfo benchmark program.

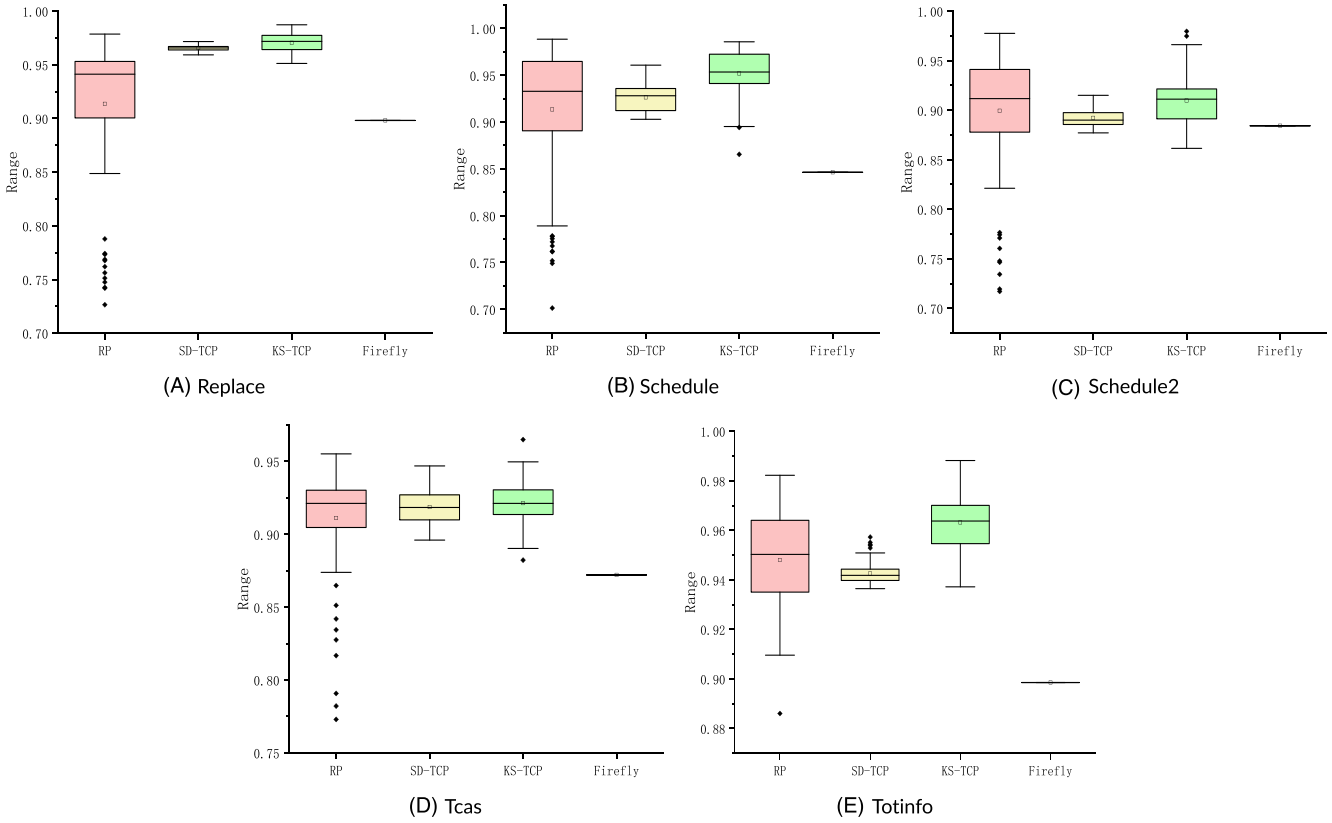

FIGURE 5 The APFD of Siemens suite.

TABLE 3 Overall APFD value on program Replace.

	RP	SD-TCP	KS-TCP	FA-TCP
Mean	0.913527476	0.965476537	0.970527588	0.898077438
Std-dev	0.064204737	0.002586399	0.008819330	0
p value	2.96e−12	6.86e−30	-	6.40e−234

TABLE 4 Overall APFD value on program Schedule.

	RP	SD-TCP	KS-TCP	FA-TCP
Mean	0.913393291	0.926386583	0.951910356	0.846268344
Std-dev	0.069051579	0.014183613	0.023559926	0
p value	3.25e−6	4.16e−20	-	6.10e−106

Figure 5 shows the results of several test case prioritization algorithms on the Siemens program set. The APFD results show that the proposed KS-TCP method is better than SD-TCP, FA, and RP, and it has a greater advantage over most programs. For the SD-TCP method, it can be seen from the figure that it is efficient and stable for some special input programs, such as the Replace program. However, it performs poorly on several other programs, which is probably caused by the extreme inputs affecting the sorting process more or the used sorting strategy is not well suited for these programs.

The RP method performs better than expected on the test set. Through analyzing the test cases of the Siemens program, we find that many programs have a situation that some single test input triggers multiple versions of the error. This condition leads to the fact that the experimental results are all at a high level. The result in graph also shows that excluding some outliers, the APFD values for almost all methods are higher than 0.85. Therefore, in some programs, such as the Totinfo program, the differences in the sorting methods are not significant.

TABLE 5 Overall APFD value on program Schedule2.

	RP	SD-TCP	KS-TCP	FA-TCP
Mean	0.899415703	0.892269783	0.909666667	0.884276343
Std-dev	0.059961031	0.008539822	0.022001138	0
p value	5.67e−3	5.67e−12	-	1.93e−28

TABLE 6 Overall APFD value on program Tcas.

	RP	SD-TCP	KS-TCP	FA-TCP
Mean	0.911206095	0.918744236	0.921360120	0.872095316
Std-dev	0.033790672	0.011297074	0.014009661	0
p value	3.33e−4	2.23e−2	-	1.50e−183

TABLE 7 Overall APFD value on program Totinfo.

	RP	SD-TCP	KS-TCP	FA-TCP
Mean	0.947993569	0.942519011	0.963137636	0.898557613
Std-dev	0.019374333	0.004073075	0.009931330	0
p value	7.16e−9	4.17e−49	-	4.00e−127

In addition, we also statistically analyze our method with each other methods using *p* value, and the results are shown in the last row of the table. The experimental result shows that all *p* values are less than 0.05, which verifies the significant differences between our method and other methods.

For static black box information-based sorting, the single use of string distance metric may not fully reflect the differences between string distances. There are some limitations for some input formats, such as tables, and multiline text, which requires targeted modification of the metric. Our experimental setup of the metric may be more suitable for string parameters with restricted length and numeric parameters with fixed dimensionality, such as Tcas programs. In addition, the fixed dimensional input can achieve better sorting results.

5.3 | Execution time

Figure 6 shows the time overhead comparisons for SD-TCP, KS-TCP, and FA on using the original test case set, where the size of test sets varies from 1052 to 5542. The results are very similar on all benchmark programs, and KS-TCP looks like a horizontal line due to the large deviation of the values. As can be seen from the results, the proposed KS-TCP greatly reduces the time overhead based on SD-TCP. In the preprocessing phase, both methods need to calculate the distance matrix for the subsequent sorting process. CompareD with SD-TCP, the proposed method performs the sorting process on a subset of test cases; in contrast, SD-TCP performs several iterations and picks on all test cases. Compared with SD-TCP, the use of partitioning process in our method saves much time overhead. The time complexity of SD-TCP is almost close to $O(n^2)$, while the time complexity of KS-TCP is $O(k*(n/k)^2)$, when the size of K is related to n, it can reach $O(n)$. For FA, although the experimental configuration of weights is missing, it does not increase the execution time. FA needs to update the state of fireflies each time according to the brightness and finally select the shortest path, while the update of the state needs a higher time cost; thus, FA requires a higher execution time than the other two methods. All three methods require the extraction of the test case distance, and they have the same calculation strategy; thus, we omitted the extraction time in our experiments to make a more direct comparison. Table 8 shows the specific average values, and in general, KS-TCP has an obvious advantage as the number of test cases increases.

5.4 | The first errors found

Recording the location of the first test case to find a bug is often used to evaluate test case generation methods,[40,41] and black-box static test case prioritization methods can be combined with these methods to test the program under test. We assume that the original test cases is

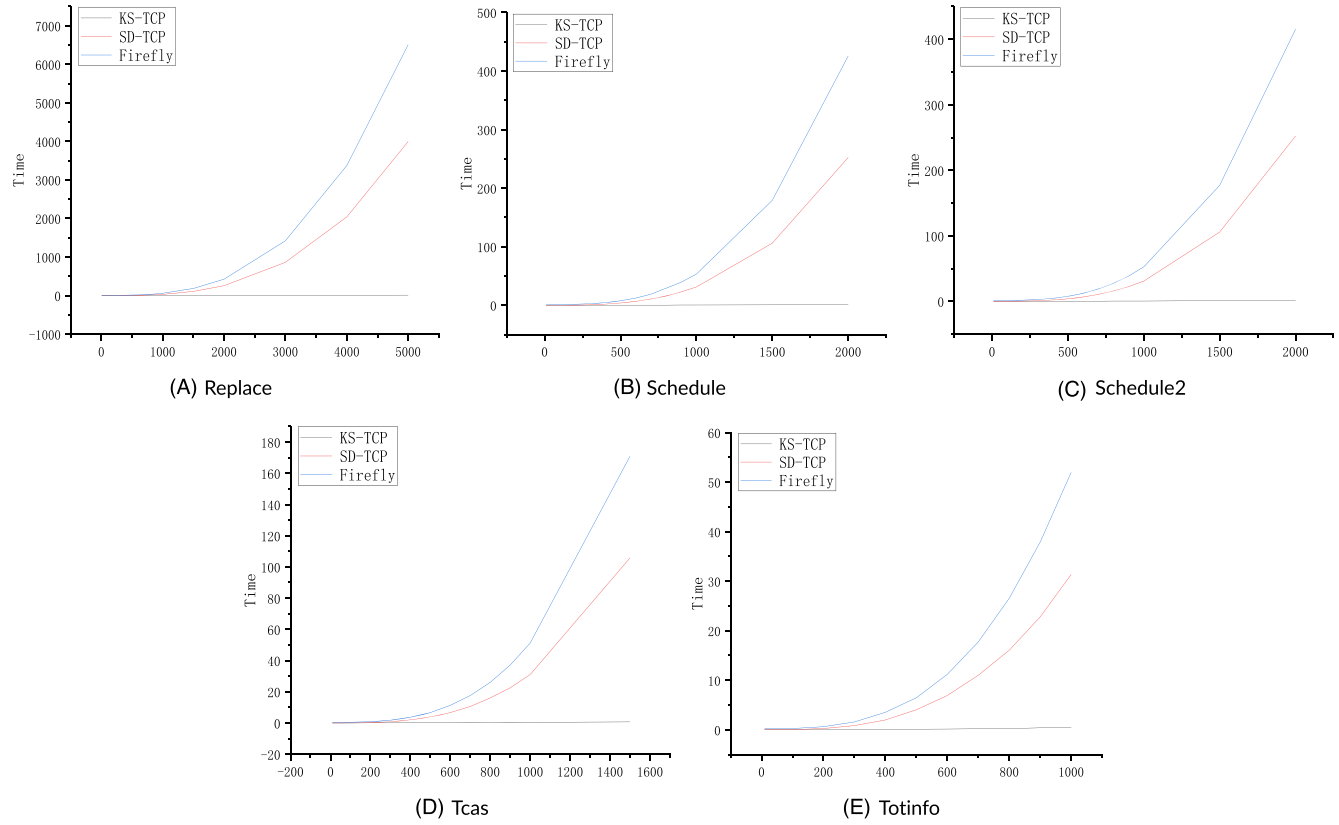

FIGURE 6 The time of Siemens suite.

TABLE 8 Average execution time.

	Replace	Schedule	Schedule2	Tcas	Totinfo
SD-TCP	3990.994	252.171	252.142	105.764	31.336
KS-TCP	14.406	1.800	1.283	0.718	0.485
FA-TCP	6504.712	425.472	415.319	170.893	51.947

TABLE 9 First error.

	Replace	Schedule	Schedule2	Tcas	Totinfo
RP	4.72	4.3	14.32	3.48	1.96
SD-TCP	6.48	3.56	1.98	2.96	1.83
KS-TCP	2.17	3.39	2.17	2.89	1.23
FA-TCP	1	8	6	2	2

generated by one of the generation methods. and different sequencing methods prioritize the execution of important test cases. We additionally record the position of first test case that finds a bug in each execution sequence after sequencing in our experiments, and then only show the final average values. Table 9 shows that all three sorting methods have good performance, where the proposed KS-TCP method performs better on three benchmark procedures of Replace, Schedule, and Schedule2.

Table 10 summarizes the APFD results for the four test case prioritization algorithms. Overall, we can find from the above tables and figures that the proposed KS-TCP method outperforms SD-TCP, FA and RP in effectiveness and has a big improvement (at least 50%) over SD-TCP in sorting efficiency.

TABLE 10 Overall.

	Replace	Schedule	Schedule2	Tcas	Totinfo
RP	0.913527	0.913393	0.899415	0.911206	0.947993
SD-TCP	0.965476	0.926386	0.892269	0.918744	0.942519
KS-TCP	0.970527	0.951910	0.909666	0.92136	0.963137
FA-TCP	0.898077	0.846268	0.884276	0.872095	0.898557

6 | CONCLUSION

To improve prioritization efficiency and diminish the effects of extreme inputs on the SD-TCP, this paper proposes a static black-box test case prioritization method called KS-TCP. The method uses a new medoid picking strategy based on K-medoids to cope with the black-box static information of the test cases. In the KS-TCP, the test suites will be divided more finely based on similarity, and the subsets are then prioritized using a greedy strategy. The final execution sequence is selected by polling. Our approach improves the impact of extreme inputs, and the grouping process enhances the efficiency of test case prioritization. Related empirical studies have shown that the proposed method is significantly improved in time efficiency compared to several other static black-box TCP methods as well as has a better level on the effectiveness.

In future, we will try our best to seek for other reasonable static black-box test case similarity metrics for test case prioritization as well as use other clustering methods for further comparison. In addition, we will also try to combine our strategy with white-box TCP techniques to improve the usability of the proposed approach.

ACKNOWLEDGMENTS

This work was partly supported by the National Key R&D Program of China (Grant no. 2020YFB1005500), the National Natural Science Foundation of China (NSFC) (Grant nos. 62172194, 62202206, and U1836116), the Leading-edge Technology Program of Jiangsu Natural Science Foundation (Grant no. BK20202001), the Natural Science Foundation of Jiangsu Province (Grant no. BK20220515), the China Postdoctoral Science Foundation (Grant no: 2021M691310), the Postdoctoral Science Foundation of Jiangsu Province (Grant no: 2021K636C), and Qinglan Project of Jiangsu Province.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are available in SIR at https://sir.csc.ncsu.edu/php/index.php. These data were derived from the following resources available in the public domain: Siemens suite, https://sir.csc.ncsu.edu/php/showfiles.php.

ORCID

Jinfu Chen https://orcid.org/0000-0002-3124-5452
Yuechao Gu https://orcid.org/0009-0007-0375-8326
Saihua Cai https://orcid.org/0000-0003-0743-1156
Jingyi Chen https://orcid.org/0000-0003-2668-6592

REFERENCES

1. Alkawaz MH, Silvarajoo A. A survey on test case prioritization and optimization techniques in software regression testing. In: 2019 IEEE 7th Conference on Systems, Process and Control (ICSPC); 2019:59-64.
2. Xia C, Zhang Y, Hui Z. Test suite reduction via evolutionary clustering. *IEEE Access*. 2021;9:28111-28121. doi:10.1109/ACCESS.2021.3058301
3. Ashima, Shaheamlung G, Rote K. A comprehensive review for test case prioritization in software engineering. In: 2020 International Conference on Intelligent Engineering and Management (ICIEM); 2020:331-336.
4. Zhang X, Xie X, Chen TY. Test case prioritization using adaptive random sequence with category-partition-based distance. In: 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS); 2016:374-385.
5. Malz C, Jazdi N, Gohner P. Prioritization of test cases using software agents and fuzzy logic. In: 2012 IEEE fifth International Conference on Software Testing, Verification and Validation; 2012:483-486.
6. Zhang L, Hao D, Zhang L, Rothermel G, Mei H. Bridging the gap between the total and additional test-case prioritization strategies. In: 2013 35th International Conference on Software Engineering (ICSE); 2013:192-201.
7. Lu Y, Lou Y, Cheng S, Zhang L, Hao D, Zhou Y, Zhang L. How does regression test prioritization perform in real-world software evolution? In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE); 2016:535-546.
8. Di Nardo D, Alshahwan N, Briand L, Labiche Y. Coverage-based test case prioritisation: an industrial case study. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation; 2013:302-311.

9. Zhou J, Hao D. 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); 2017:392-394.
10. Elbaum SG, Malishevsky AG, Rothermel G. Incorporating varying test costs and fault severities into test case prioritization; 2001:329-338. doi:10.1109/ICSE.2001.919106
11. Zheng Z, Li C, Liu Y, Xi Z. A phase-type expansion approach for the performability of composite web services. *IEEE Trans Reliab*. 2022;71(2):579-589. doi:10.1109/TR.2022.3145381
12. Azizi M. A tag-based recommender system for regression test case prioritization. In: 2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW); 2021:146-157.
13. Liu W, Wu X, Zhang W, Xu Y. The research of the test case prioritization algorithm for black box testing. In: 2014 IEEE 5th International Conference on Software Engineering and Service Science; 2014:37-40.
14. Xi WANG. Application of UML statechart diagram in regression test. *Comput Eng*. 2008;35(4):63-65.
15. Gu Q, Tang B, Chen D-X. A test suite reduction technique for partial coverage of test requirements. *Jisuanji Xuebao(Chin J Comput)*. 2011;34(5):879-888.
16. Thomas SW, Hemmati H, Hassan AE, Blostein D. Static test case prioritization using topic models. *Empir Softw Eng*. 2014;19(1):182-212. doi:10.1007/s10664-012-9219-7
17. Chen J, Kuo F, Chen TY, Towey D, Su C, Huang R. A similarity metric for the inputs of OO programs and its application in adaptive random testing. *IEEE Trans Reliab*. 2017;66(2):373-402. doi:10.1109/TR.2016.2628759
18. Ledru Y, Petrenko A, Boroday S, Mandran N. Prioritizing test cases with string distances. *Autom Softw Eng*. 2012;19(1):65-95. doi:10.1007/s10515-011-0093-0
19. Lou Y, Chen J, Zhang L, Hao D. Chapter one—a survey on regression test-case prioritization. *Adv Comput*. 2019;113:1-46. doi:10.1016/bs.adcom.2018.10.001
20. Suleiman D, Alian M, Hudaib A. A survey on prioritization regression testing test case. In: 2017 8th International Conference on Information Technology (ICIT); 2017:854-862.
21. Nishino K, Kitamura T, Kishi T, Artho C. Toward an encoding approach to interaction-based test suite minimization. In: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW); 2020:211-212.
22. Dhareula P, Ganpati A. Prevalent criteria's in regression test case selection techniques: an exploratory study. In: 2015 International Conference on Green Computing and Internet of Things (ICGCIOT); 2015:871-876.
23. Fraser G, Rojas JM. Software testing. In: Cha S, Taylor RN, Kang KC, eds. *Handbook of Software Engineering*: Springer; 2019:123-192. doi:10.1007/978-3-030-00262-6_4
24. Huang Y-C, Peng K-L, Huang C-Y. A history-based cost-cognizant test case prioritization technique in regression testing. *J Syst Softw*. 2012;85(3):626-637. doi:10.1016/j.jss.2011.09.063
25. Bajaj A, Sangwan OP. A survey on regression testing using nature-inspired approaches. In: 2018 4th International Conference on Computing Communication and Automation (ICCCA); 2018:1-5.
26. Yoo S, Harman M. Regression testing minimization, selection and prioritization: a survey. *Softw Test Verification Reliab*. 2012;22(2):67-120. doi:10.1002/stv.430
27. Rothermel G, Untch RH, Chu C, Harrold MJ. Prioritizing test cases for regression testing. *IEEE Trans Softw Eng*. 2001;27(10):929-948. doi:10.1109/32.962562
28. Ramya P, Sindhura V, Vidya Sagar P. Clustering based prioritization of test cases. In: 2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT); 2018:1181-1185.
29. Chen J, Zhu L, Chen TY, Huang R, Towey D, Kuo F-C, Guo Y. An adaptive sequence approach for OOS test case prioritization. In: 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW); 2016:205-212.
30. Chen J, Zhu L, Chen TY, Towey D, Kuo F-C, Huang R, Guo Y. Test case prioritization for object-oriented software: an adaptive random sequence approach based on clustering. *J Syst Softw*. 2018;135:107-125. doi:10.1016/j.jss.2017.09.031
31. Chen J, Chen H, Guo Y, Zhou M, Huang R, Mao C. A novel test case generation approach for adaptive random testing of object-oriented software using K-means clustering technique. *IEEE Trans Emerg Top Comput Intell*. 2022;6(4):969-981. doi:10.1109/TETCI.2021.3122511
32. Yoo S, Harman M, Tonella P, Susi A. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In: Rothermel G, Dillon LK, eds. *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, Il, USA, july 19-23, 2009*: ACM; 2009:201-212.
33. Carlson R, Do H, Denton A. A clustering approach to improving test case prioritization: an industrial case study. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM); 2011:382-391.
34. Ackah-Arthur H, Chen J, Towey D, Omari M, Xi J, Huang R. One-domain-one-input: adaptive random testing by orthogonal recursive bisection with restriction. *IEEE Trans Reliab*. 2019;68(4):1404-1428. doi:10.1109/TR.2019.2907577
35. Li Z, Harman M, Hierons RM. Search algorithms for regression test case prioritization. *IEEE Trans Softw Eng*. 2007;33(4):225-237. doi:10.1109/TSE.2007.38
36. Khatibsyarbini M, Isa MA, Jawawi DNA, Hamed HNA, Suffian MDM. Test case prioritization using firefly algorithm for software testing. *IEEE Access*. 2019;7:132360-132373. doi:10.1109/ACCESS.2019.2940620
37. Do H, Elbaum SG, Rothermel G. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empir Softw Eng*. 2005;10(4):405-435. doi:10.1007/s10664-005-3861-2
38. Hasnain M, Ghani I, Pasha MF, Hong C, Jeong SR. A comprehensive review on regression test case prioritization techniques for web services. *KSII Trans Internet Inf Syst*. 2020;14(5):1861-1885.
39. Elbaum SG, Malishevsky AG, Rothermel G. Test case prioritization: a family of empirical studies. *IEEE Trans Softw Eng*. 2002;28(2):159-182. doi:10.1109/32.988497

40. Lian J, Cui C, Sun W, Wu Y, Huang R. KD-RRT: restricted random testing based on k-dimensional tree. In: 8th International Conference on Dependable Systems and their Applications, DSA 2021, Yinchuan, China, August 5-6, 2021. IEEE; 2021:590-599.
41. Huang R, Sun W, Chen TY, Ng S, Chen J-F. Identification of failure regions for programs with numeric inputs. *IEEE Trans Emerg Top Comput Intell*. 2021;5(4):651-667. doi:10.1109/TETCI.2020.3013713

How to cite this article: Chen J, Gu Y, Cai S, Chen H, Chen J. A novel test case prioritization approach for black-box testing based on K-medoids clustering. *J Softw Evol Proc*. 2024;36(4):e2565. doi:10.1002/smr.2565