Hindawi
IET Software
Volume 2024, Article ID 9942959, 14 pages
https://doi.org/10.1049/2024/9942959

Research Article

Exploiting DBSCAN and Combination Strategy to Prioritize the Test Suite in Regression Testing

Zikang Zhang,[1] Jinfu Chen,[1] Yuechao Gu,[1] Zhehao Li,[1] and Rexford Nii Ayitey Sosu[1,2]

[1]*School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang 202013, China*
[2]*Faculty of Computing and Information Systems, Ghana Communication Technology University, Accra 233021, Ghana*

Correspondence should be addressed to Jinfu Chen; jinfuchen@ujs.edu.cn

Received 15 October 2023; Revised 28 February 2024; Accepted 14 March 2024; Published 4 April 2024

Academic Editor: Hui Liu

Test case prioritization techniques improve the fault detection rate by adjusting the execution sequence of test cases. For static black-box test case prioritization techniques, existing methods generally improve the fault detection rate by increasing the early diversity of execution sequences based on string distance differences. However, such methods have a high time overhead and are less stable. This paper proposes a novel test case prioritization method (DC-TCP) based on density-based spatial clustering of applications with noise (DBSCAN) and combination policies. By introducing a combination strategy to model the inputs to generate a mapping model, the test inputs are mapped to consistent types to improve generality. The DBSCAN method is then used to refine the classification of test cases further, and finally, the Firefly search strategy is introduced to improve the effectiveness of sequence merging. Extensive experimental results demonstrate that the proposed DC-TCP method outperforms other methods in terms of the average percentage of faults detected and exhibits advantages in terms of time efficiency when compared to several existing static black-box sorting methods.

1. Introduction

Regression testing [1, 2] is an important software testing technique designed to ensure that software with fixed defects does not introduce new functional failures and still meets user requirements and security specifications. Therefore, regression testing is an important part of the software development and maintenance cycle, and it plays a key role in software quality and reliability. However, there are some issues with regression testing, such as the selection of test data, the maintenance of test cases, and the testers' knowledge levels. In addition, the scope of regression testing may also vary depending on the project's complexity, leading to increased difficulty and time cost of testing. Therefore, regression testing must be carefully planned and executed during development to ensure maximum value. In order to reduce the overall test time overhead, there has been much research on regression testing, including test set reduction techniques [3, 4], test case selection [5, 6], and test case prioritization [7, 8]. Test set reduction techniques aim to improve testing efficiency by reducing the number of test cases, mainly by removing redundant ones, thereby reducing testing time and ensuring the test suite's quality. The test case prioritization technique aims to find the best order of test case execution and then prioritize the execution of important test cases to maximize the effort of testers. While test case reduction techniques truncate the original test set, test case prioritization techniques do not remove any test cases. In other words, test case prioritization techniques are relatively conservative in comparison, yet they provide enhanced security and reliability. Therefore, researchers are more interested in test case prioritization techniques.

Test case prioritization techniques can be divided into dynamic and static categories, depending on the required links and dependent information. Dynamic ranking techniques are widely studied due to their high effectiveness, but these types of techniques are usually more complex and difficult to reuse, as they depend on historical execution information of the software. Static techniques, on the other hand, can sort test cases when execution information is not available. Also, since static sorting techniques can also sort newly generated test cases, they are more widely applicable.

Many test case ranking methods have been proposed based on static test case information. Among them, the string distance-based test case ordering method is one of the classical and effective prioritization methods [9]. It relies solely on the text of the test cases for ranking. The existing black-box test case sorting technique based on string distance is simple to use and demonstrates a certain sorting effect without relying on execution information. However, it has limitations on performance due to its time overhead, and the string similarity metric tends to ignore some of the information available for the test cases, thereby impacting the sorting effect. In light of the drawbacks of static test case ranking methods, one solution is to use clustering analysis techniques [10] to differentiate test cases more granularly to process test cases individually based on the clustering results. In the previous study, the proposed K-medoids and similarity-based test case prioritization (KS-TCP) [11, 12] method can efficiently rank test cases with a certain degree of efficiency. Nonetheless, there are still some limitations in terms of applicability as well as stability.

This paper proposes a novel test case ranking method based on density-based spatial clustering of applications with noise (DBSCAN) and a combinatorial strategy to address these issues. First, the concept of the combinatorial strategy is presented to tackle the limitation in applicability observed in the KS-TCP method, particularly concerning program input types. This approach transforms test cases into consistent numerical parameters, thereby sidestepping the need for specialized treatment of intricate types and enhancing the inclusiveness of the black-box static sorting method. Subsequently, the DBSCAN method is used to improve the clustering effect to better cope with the transformed input types and improve the method's stability. Finally, the search strategy in the Firefly algorithm, as well as the random and rest mechanisms, are combined to guide the sequence merging process and improve the selection efficiency of the algorithm. The experiments apply the sorting method to the combinatorial testing technique to guide the testing process, and the experimental results show that the DBSCAN-based test case prioritization (DC-TCP) algorithm has certain advantages in algorithm stability while ensuring a certain fault detection rate.

The main contributions of this paper are as follows:

(1) Introduction of combination strategy: This paper introduces a combination strategy for conducting similarity analysis on static test cases. By mapping test inputs to a consistent type, this method enhances the generality of the test case sorting approach. The application of this combination strategy enables the sorting method to better handle different types of test cases, improving the consistency and accuracy of the sorting results.

(2) Clustering analysis based on DBSCAN: To further enhance the classification effectiveness of test cases, this paper employs the DBSCAN method. By utilizing the DBSCAN method for finer clustering of test cases, it becomes possible to better distinguish between different types of test cases and enhance the stability of the sorting method.

(3) Introduction of Firefly search strategy: To improve the effectiveness of sequence merging, this paper introduces the Firefly search strategy. By utilizing the Firefly search strategy to guide the sequence merging process, the algorithm's selection efficiency is enhanced, further optimizing the sorting results of test cases.

(4) Extensive experimental evaluation: To assess the effectiveness and efficiency of the proposed DC-TCP method, this paper conducts extensive experimental comparisons on 10 Java datasets. The experimental results indicate that the DC-TCP method outperforms other methods in terms of the average percentage of faults detected (APFD) metric. Moreover, it exhibits a significant advantage in terms of time efficiency compared to existing static black-box sorting methods.

The rest of the paper is organized as follows: Section 2 presents some background knowledge. Section 3 presents the methodology. Section 4 evaluates the effectiveness and efficiency of the proposed method through an empirical study. Section 5 discusses potential limitations to the validity of our work. Section 6 gives the conclusion.

2. Background and Related Work

2.1. Test Case Prioritization. Test case prioritization techniques are designed to adjust the order of test case execution to improve overall test efficiency as well as test quality. Suppose the execution of a scheduled test set is interrupted or stopped for any reason. In that case, the important test cases will also be prioritized, thus minimizing the loss of complete execution due to sudden changes in test costs. Test case prioritization allows the perfect sequencing of test cases to generate an execution sequence that meets testers' expectations optimally. Therefore, the test case prioritization strategy becomes critical, and a wrong prioritization strategy can even lead to the exact opposite result, increasing the required testing cost. Among the test case prioritization techniques, the following four types of techniques can be classified based on the required information as well as the state of the environment [13, 14].

(1) White-box execution-based prioritization is a ranking method based on the dynamic execution information of the software to be tested and the test cases. Generating the execution sequence requires knowledge of the software's source code to be tested and the historical execution flow information of each test case in the software source code.

(2) Black-box execution-based prioritization is a ranking method based on the dynamic execution information

of the test cases, where the execution sequence is generated using the execution logs of the test cases and does not depend on the actual source code of the software to be tested.

(3) White-box static prioritization is a ranking method based on information about the software to be tested and test cases. The execution sequence is generated based on software knowledge and static information about the test cases and does not depend on the historical execution information of the test cases.

(4) Black-box static prioritization is a ranking method based on test case information. The execution sequence is generated based on the test cases themselves and does not require the historical execution information of the software to be tested or knowledge of the model; it only relies on some information that exists in the test cases themselves.

Dynamic test case prioritization techniques are more effective for regression testing applications than static methods. However, dynamic techniques are usually more complex and may have some technical difficulties in practical applications; in addition, the technique requires historical versions of execution information (e.g., source code [15–17], coverage information [18, 19], fault severity [20], etc.)

Dynamic techniques usually do not avoid the question: Is the historical execution information of the test cases available? Dynamic sequencing techniques rely heavily on execution information, and it is difficult to function when execution information is absent [21]. For example, it is too expensive for larger software systems to collect and maintain execution information as software versions evolve, and the execution information needs to be updated as the source code changes. In contrast to dynamic sequencing techniques, static sequencing techniques do not rely on historical execution information or knowledge of the software but only on the current version of the software or information that already exists on its own [22]. In addition, static black-box techniques can sort newly generated test cases and can be applied in the initial testing of the software. It can also be combined with different test case generation techniques for testing [23] and thus has a broader scope of application.

The discussion above focuses on traditional TCP methods, but recent approaches often integrate machine learning techniques for improved results [24–26]. Recent studies [27–29] also propose supervised machine learning techniques for test prioritization as a ranking problem. However, this paper primarily emphasizes traditional TCP methods.

2.2. Coverage-Based Test Case Prioritization Techniques. The coverage-based test case prioritization technique is one of the basic strategies commonly used in regression testing, and it is generally believed that increasing software execution coverage can improve the efficiency of software defect detection. For example, if test cases A and B cover more functions, branches, and statements, test case A may have a higher probability of detecting triggered software defects than test case B.

Mahdieh et al. [30] improved the coverage-based ranking method by considering the distribution of fault propensity over code blocks. The proposed method introduces defect prediction techniques and trains a neural network model using historical error records of the software, which is then used to predict the fault propensity of each region of the source code and merge the estimation results into the coverage-based ranking method. The method proposes a generalized strategy that can be applied to most coverage-based methods. Its experimental results show that using appropriate historical error records can improve the coverage-based ranking method.

Rothermel et al. [8] first defined the test case prioritization problem and also proposed the original coverage-based ranking technique, which mainly consists of two greedy algorithms. The first is the global greedy algorithm, greedy essential, which treats each test case individually and ranks them according to execution coverage from the highest to lowest. The other one considers the overall code coverage of the combination and ranks the test cases according to their contribution to the extra code coverage, known as the extra greedy algorithm, greedy redundant essential. Various test case ranking techniques are proposed based on statement coverage and branch coverage; in addition to the two greedy algorithms, there is also a capability for detecting test case history. These ranking techniques are compared with random sampling sequences and optimal sequences in experiments, and the results show that these techniques can effectively improve the detection efficiency.

2.3. Requirements-Based Test Case Sequencing Techniques. Srikanth et al. [31] first used test requirements (including customer priority, error-prone probability, fluctuation value, and execution difficulty) in a test case prioritization technique, but this technique is more subjective due to the need for human prediction of requirement attributes. Srikanth et al. [32] found in their recent study that the effect of prioritization of two or more factors is better than that of a single factor in terms of test validity.

Muthusamy and Seetharaman [33] proposed a new algorithm for prioritizing test cases grounded in requirements (including traceability, completeness, requirement error impact, requirement changes, customer priority, and developer view). It hinges on multiple weighting factors, strategically employing diverse attributes to enhance the efficacy of the ranking process. The method is more efficient in fault detection compared to random ranking methods.

2.4. Search-Based Test Case Sequencing Techniques. There has been a lot of research related to search-based test case ranking methods, such as genetic algorithms [34, 35], greedy algorithms [36], ant colony algorithms [37], etc. Recent research by Li et al. [38] has shown that genetic algorithms are less effective in data generation than greedy algorithms. However, the application in search-based algorithms may differ depending on the chosen test set, input criteria, fitness function, etc. Although the experimental results demonstrate the advantages of genetic algorithms applied to test case ranking methods, there are also some disadvantages; for example, the slow ranking process using genetic algorithms can lead to a high time overhead.

2.5. Model-Based Test Case Prioritization Techniques. Fraser and Wotawa [39] proposed a method for test case ranking using a model detector, the main advantage of which is that it can cover critical paths in a software system and rank them efficiently according to the criticality of the test cases, thus covering as many important modules of the software system as possible with limited testing time. Its experimental results show that this model-based ranking technique can obtain better testing results than the coverage-based ranking technique, but it relies on the quality of the model specification description.

Korel et al. [40] proposed a system model-based prioritization method that first uses a system model to describe the system to be tested and then uses the system model to calculate similarity to generate execution sequences, thereby quickly identifying similarity and reducing testing costs. Its experimental results show that the system model can effectively be used in test case prioritization and improve overall effectiveness. However, the technique relies on the accuracy of the system model and requires significant upfront work, such as building the system model and generating the relevant sequences.

2.6. Text-Based Test Case Prioritization Techniques. String distance-based test case prioritization (SD-TCP) [9] prioritizes test cases based on string distance, relying solely on test case text and independent of program knowledge. It calculates similarity using string distance metrics like Euclidean, Manhattan, Hamming, and edit distance, with Manhattan distance proving more effective. The core concept involves adding each test case to the least similar sequence in each iteration for prioritization.

Firefly algorithm for test case prioritization (FA-TCP) [41] is a method for prioritizing software test cases using the Firefly algorithm. This approach optimizes the sorting of test cases by applying the Firefly algorithm and a fitness function defined by a similarity distance model.

KS-TCP [12] efficiently organizes test cases using K-medoids and similarity. This method groups test cases, applies a greedy strategy for prioritization within each group, and employs a polling mechanism for the final execution order. Prioritization considers the entire set of test cases, reducing the impact of extreme cases. Experiments reveal that, compared to random sample test case prioritization (RS-TCP) and SD-TCP, KS-TCP achieves a higher APFD and better time efficiency.

3. Proposed Method

3.1. Motivation. The KS-TCP method we proposed in our previous study can effectively reduce the time overhead of the SD-TCP method in terms of testing efficiency. However, the algorithm may not perform better in terms of applicability for different types of inputs. In particular, the KS-TCP method generally performs well for more complex types of inputs. Two aspects of the method could be improved, as follows:

KS-TCP starts the process of similarity analysis between test cases by treating the test cases as strings and using Manhattan distances. A correlation distance matrix is generated to provide further calculations. However, using a single-string distance metric does not guarantee that differences between test cases can be extracted. In addition, using Manhattan distances shows good variability for numeric test cases. This variability does not consistently translate to other data types (e.g., string inputs, table inputs, composite inputs, and some difficult-to-parse file inputs). For example, here are three test cases as follows.

Test Case 1: *server read only close;*

Test Case 2: *server read only close prepared true;*

Test Case 3: *webserver updatable hold.*

Test Case 1 has one more last parameter compared to Test Case 2, while Test Case 2 and Test Case 3 have the same length but almost no intersection of content. If measured by string distance, the distance between Test Case 2 and Test Case 1 is much greater than the distance between Test Case 2 and Test Case 3. However, from the semantic point of view, the similarity between Test Case 1 and Test Case 2 may be higher. To address these challenges, a combination strategy is introduced. The method's applicability is further improved by extracting models to transform the test case presentation for complex and composite types of inputs so that all programs can get consistent inputs and facilitate the subsequent similarity analysis process.

KS-TCP uses the K-medoids clustering method to refine the test case classification and provide the basis for subsequent operations. However, the K-medoids method itself has some problems. First of all, the original method mainly targets numerical data clustering, which is adapted in the KS-TCP method by changing the centroid selection strategy. In addition, the K-medoids method is prone to sensitivity regarding its initial conditions, and the number of clusters necessitates manual setting. The initial conditions may change, greatly impacting the final clustering results. The setting of the number of clusters, if for long-term regression testing, may have a certain setting basis for the parameters, which can weaken the advantage of the static test case prioritization technique for ranking new test cases. On the other hand, introducing the combination strategy leads to a change in test cases and the need to find a more appropriate clustering method. For this reason, this method introduces DBSCAN density clustering to alleviate this one problem.

DBSCAN, a density-based clustering algorithm, stands out for its effective handling of noise and outliers compared to distance-based algorithms like K-means. In software testing, where uncertainties and exceptional situations arise, DBSCAN excludes outliers by setting the neighborhood radius parameter, enhancing clustering accuracy and stability.

Unlike algorithms requiring a prespecified number of clusters, DBSCAN adapts to varying density and shape structures, automatically discovering patterns without constraints on cluster numbers. In test case prioritization, especially with uneven test case distribution, DBSCAN accommodates such

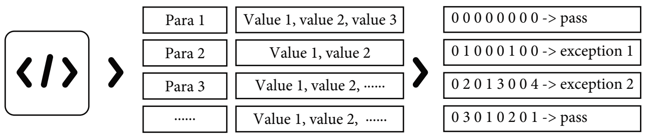


Figure 1: The framework of the KS-TCP approach.

scenarios, revealing potential clustering structures. Its insensitivity to the order of data points is crucial for test case prioritization, where execution order influences results.

Choosing DBSCAN over OPTICS, a similar algorithm, is driven by its simpler computational process. OPTICS involves calculating reachability distances, constructing graphs, tuning parameters, and adding complexity and implementation difficulty. DBSCAN's streamlined approach makes it a preferred choice.

In conclusion, DBSCAN can automatically identify the number of clusters to discover arbitrarily shaped cluster classes and is not limited to numerical inputs when targeting complex types of inputs, which can be an improvement to KS-TCP in terms of applicability and stability.

3.2. Combination Strategy. In the KS-TCP method, the selected string metric does not cope well with all types of inputs for different types of inputs. It is difficult to achieve the same prioritization effect for complex and compound types of inputs. Also, it may be necessary to design specific input and output configurations for the software to be tested before prioritizing the test cases.

Therefore, unifying different input types is a key to similarity analysis. This method first builds a model for the test case inputs of the program to be tested. It extracts the generic input parameters and then feeds these extracted parameters into the model. Multiple options are available for each parameter position of the program under test. This method numbers the set of parameters that appear at each parameter location and then maps the test cases. As shown in Figure 1, the program to be tested in the figure has multiple input parameters, and the list represents the optional inputs, which can be represented as the test cases on the right side. In Figure 1, 0 indicates the selection of the first optional option, and so on. This generates a new test case composed of numbers. Afterward, the test cases are sorted. Then, reflect the projection and recover the original test cases.

With the above strategy, the software to be tested will be modeled so that there is no need to use specific metrics and different configurations for complex types, allowing the next algorithmic framework to handle consistent data and improve the generality of the approach.

3.3. DBSCAN Cluster. To alleviate the initial condition-sensitive problem of clustering methods in KS-TCP and to better cope with the combinatorial strategies used, this method introduces the DBSCAN method. This can automatically identify the number of clusters and discover cluster classes of arbitrary shapes, providing stable clustering results.

The DBSCAN algorithm is applied in the DC-TCP method as follows: first, randomly select an unvisited test case; then find all test cases in the neighborhood with this

```
Input: $T_{num}$, $distMatrix$, $\varepsilon$, $Minpts$
Output: $Clusters$
 1: Clusters ← ∅
 2: while $T_{num} \neq \emptyset$ do
 3:     $temp\_list$ ← {}
 4:     start ← RandomChoice(T)
 5:     Add $start$ to $temp\_list$
 6:     Delete $start$ from $T_{num}$
 7:     $Expand(start, T_{num}, temp\_list, \varepsilon, Minpts)$
 8:     Add $temp\_list$ to $Clusters$
 9: end while
10: return $Clusters$
```

Algorithm 1: DBSCAN.

test case, and if the number of test cases in the neighborhood is greater than or equal to the set threshold, add these test cases to the same set and mark them as visited; for the test cases that have been added to the set, continue to search for test cases in their neighborhoods by threshold until no more new test cases can be added; then, continue to randomly select the next test case that has not been visited and repeat the above steps until all test cases are visited; finally, output the clustered set.

Algorithm 1 shows the operation flow in DC-TCP using the DBSCAN clustering method. The algorithm requires input test set T_{num}, distance matrix *distMatrix*, neighborhood radius e, and domain density *Minpts*, and the final output result Clusters. First, the set Clusters is initialized (line 1), and an element start is randomly selected from T_{num}, and start is put into *temp_list* and selected from T_{num} (lines 4–6). Then, a recursive, iterative process is executed to find all neighbors of start and remove them from T_{num} based on the distance division radius. If the number of neighbors exceeds the density threshold *Minpts*, it indicates that start is a core point, and the neighbor points of start need to be recursively continued to be divided (line 7) until they do not exceed the threshold or until all have been visited. Then, the clustering set *temp_list* is put into Clusters (line 8). Finally, when all elements in T_{num} are picked, it means the DBSCAN clustering process is completed, and the final clustering result Clusters (line 10) is output for the input of the subsequent process.

3.4. Firefly Search. To further improve the efficiency of greedy search in the KS-TCP method, the search strategy in the Firefly algorithm [41] is introduced in this method to select the test cases in the set. The KS-TCP method utilizes the greedy idea of the original method to improve the early diversity of the execution sequence. However, this strategy still has a high time overhead for larger clustering sets and may lack some stability for different test set classification cases. Therefore, this method helps to merge the classified test cases by leveraging the search strategy in the Firefly algorithm. Compared with the original greedy strategy, it not only achieves a similar greedy selection effect to improve the early diversity of the execution sequence but also ensures the stability of test case selection.

This strategy can consider the set after clustering as a Firefly. The attractiveness of light sources between fireflies depends on the similarity distance between them, and finally, the paths of fireflies move to build the execution sequence. Suppose there are four sets after clustering and many test cases in each set. The test cases in the sets are compared with those in the other sets, and their distances are used as the factors of light source attraction. First, the first set is randomly selected and moved from its first element, and each time, the movable test cases are calculated to the next set point, and then the next position is calculated from the test cases in the next position, and so on, to get the final search path. Searching the path with the corresponding test cases is added to form the execution sequence.

The search strategy of the Firefly algorithm acts directly on the distance of the set of test cases. Although it can achieve a greedy selection effect, but at the same time, there are some problems. One is that there may be a problem of local optimum. The second point is that since the greedy algorithm selects the brightest Firefly if Firefly A1 is similar to Firefly A2 when the Firefly flies to the distant Firefly B, there's a possibility that it might subsequently return to the position of Firefly A2. Such a strategy could impact the final generated diversity. Therefore, this method uses the following two mechanisms to alleviate the aforementioned issues:

(1) Multiple candidate sets are selected each time for the computed optimal position for the search process. Then, a random strategy is used to select a set where the first test case is placed in the execution sequence. The suboptimal solution is selected with random probability to reduce the number of cases that fall into one error orientation.
(2) A rest queue is used to store the set number during the selection process, and if the set has been selected recently, it is added to the rest queue. In this scenario, it is imperative to note that any previous choices made will only be considered once. Every new selection will be added to the remaining queue without fail. Once the queue reaches its maximum capacity, the initial item in the collection will be reintroduced into the search process without exception. In addition, the size of the rest queue is used to control the number of rest rounds, thus alleviating the above possible ABA problem. The detailed algorithm execution process is described in Section 4.2.3.

Algorithm 2 shows a detailed description of the process of introducing the search strategy in the Firefly algorithm in the DC-TCP method and the related improvements.

The algorithm first initializes the rest queue, the candidate set, and the result sequence (lines 1–3). Then, an initial position Cur (line 4) is randomly selected and placed into the sequence. Then, the steps in rows 6–12 are looped until the termination condition is reached.

```
Input: Cluters, RestQueueSize
Output: S
 1: RestQueue ← Queue(RestQueueSize)
 2: candidateQueue ← ∅
 3: Res ← ∅
 4: Cur ← RandomChoice(Clusters)
 5: Add Cur to Res
 6: while Clusters ≠ ∅ do
 7:     candidateQueue ← LightFind(Clusters, RestQueue)
 8:     T_Clu ← RandomChoice(candidateQueue)
 9:     Cur ← T_Clu[0]
10:     Add Cur to Res
11:     Add T_Clu to RestQueue
12: end while
13: return S
```

Algorithm 2: Firefly-Search.

The cyclic process is described as follows: first, multiple brightest candidate sets are selected using the brightness adaptation function defined by the Firefly algorithm (line 7), from which one candidate set is randomly selected, and the other candidates are released (line 8). Finally, the position is updated (line 9), the test case is moved into the execution sequence, and the corresponding set is moved into the rest queue (line 11). Repeat this process until all sets have been picked.

3.5. Algorithm Framework and Process. The general framework of the DC-TCP approach is shown in Figure 2. It consists of four main phases as follows:

(1) First is the initialization phase, where the method will generate the corresponding input model based on the input configuration of the software to be tested and then generate the corresponding test set using the combined test method and map the initial test case set to the new alternative test set by the model.
(2) The newly generated alternative test sets are then subjected to similarity analysis using Manhattan distance to generate a distance matrix that acts on the subsequent stages.
(3) Next, based on the distance matrix obtained in the previous step, the alternative test sets are clustered and grouped using a density clustering algorithm to generate the corresponding clusters, and each cluster set is used as the subsequent input.
(4) Finally, an improved Firefly search strategy selects the sets, and test cases are selected to be added to the alternative execution sequence. After the selection is completed, the original test case type is restored using the combined input model reflective projection to generate the final execution sequence.

The ordering measures of the DC-TCP algorithm have been described in detail above, and the entire algorithm is

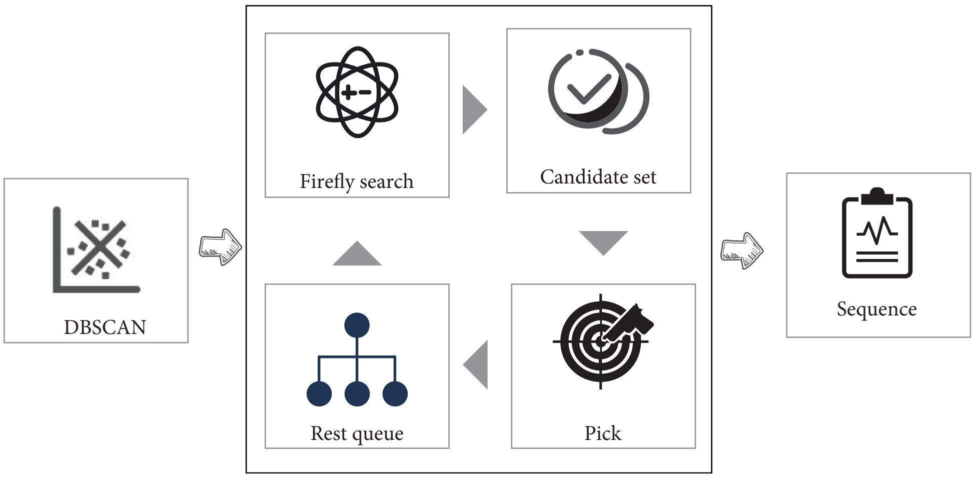


FIGURE 2: The framework of the DC-TCP approach.

```
Input: $T$, $threshold$, $RestQueueSize$
Output: $S$
 1: $S \leftarrow \{\}$
 2: $Clusters \leftarrow \{\}$
 3: $T_{\mathrm{num}} \leftarrow \emptyset$
 4: $S_{\mathrm{num}} \leftarrow \emptyset$
 5: $\mathrm{model} \leftarrow \mathrm{Model\_Build}(\mathrm{T})$
 6: $T_{\mathrm{num}} \leftarrow \mathrm{map}(T, \mathrm{model})$
 7: $distMatrix \leftarrow \mathrm{getDistance}(T_{num})$
 8: $Clusters \leftarrow DBSCAN(threshold, T_{num}, distMatrix)$
 9: $S_{num} \leftarrow Firefly(Clusters, \mathrm{RestQueueSize})$
10: $S \leftarrow \mathrm{reflect}(S_{\mathrm{num}}, \mathrm{model})$
11: return $S$
```

ALGORITHM 3: DC-TCP.

explained in detail below. The pseudo-code of the DC-TCP algorithm is shown in Algorithm 3. The algorithm requires the input of the original test set T, the threshold value threshold, and the rest queue size RestQueueSize. First, the relevant variables are initialized (lines 1–4), including the execution sequence, the cluster set Clusters, the alternative test case T_{num}, and the alternative execution sequence S_{num}. Then, the model mod is constructed based on the parameter input of the test set T (line 5), which maps T into T_{num} (line 6). The similarity analysis of the test set is then performed on T_{num} to generate the distance matrix distMatrix (line 7). Then, the clustering of T_{num} is performed using distMatrix to generate the set Clusters (line 8). Finally, the Clusters are searched using the search strategy to generate the execution sequence S_{num} (line 9), and then the model mod is used to reflect the projection S_{num} (line 10) to obtain the final execution sequence S (line 11).

4. Empirical Study and Analysis

In this section, a series of comparison experiments and a description of the related experimental setup are designed to verify the effectiveness of the DC-TCP algorithm. The SD-TCP algorithm [9], FA-TCP algorithm [41], RS-TCP [42] algorithm, and KS-TCP [12] algorithm are selected for comparison of their effectiveness. The research questions, real program experimental setups, metrics, results, and analysis of the comparison experiments are presented.

RS-TCP [12, 42] is a test case prioritization method that employs a random sampling approach to enhance the sequencing of software test cases. It will be subsequently utilized as one of the baselines to demonstrate the improvements achieved by the proposed DC-TCP.

4.1. Selection of Baselines. In this study, selected methods were used as baseline experiments due to the reliance on a static black-box approach for test case prioritization, avoiding the need for historical execution data. The choice of baseline methods is guided by specific constraints:

(1) The experiments do not depend on historical execution data, necessitating the choice of a method that does not require access to such information. Traditional methods based on code coverage information are deemed unsuitable in this context.
(2) The research focuses on static black-box methods, prioritizing test cases solely based on the test cases themselves without relying on additional information. Given the relatively few relevant methods in this domain, text-based TCP methods like SD-TCP, FA-TCP, and KS-TCP are chosen as baseline methods for comparative research.

Furthermore, the DC-TCP proposed in this paper is an improvement over KS-TCP for specific scenarios. Choosing KS-TCP facilitates a more intuitive comparison to observe the contributions and effects of the improvements made by the DC-TCP. By selecting these methods as baselines, the performance of the proposed method can be evaluated in a static black-box environment and compared with other methods. This approach helps in better understanding the advantages and limitations of DC-TCP, providing a reference and benchmark for future research.

Table 1: Details of the Java real programs.

Program	Description	Version	Lines	Dimension	Class	Defects
Hsqldb	Database management software	2rc8	139,425	12	495	5
		2.2.5	156,066	17	508	2
		2.2.9	162,784	11	525	18
Commons-Cli	Parsing command line options passed to the program	1.2	4,630	8	44	2
		1.3.1	6,433	8	48	3
Joda-Time	Java standard date and time library	2.3	82,158	9	317	2
		2.9.1	85,512	7	330	2
Jsoup	Java HTML parser	1.8.3	10,295	9	55	2
		1.9.1	10,489	7	56	3

4.2. Question. The DC-TCP method addresses some applicability and stability problems of the KS-TCP method by using techniques such as DBSCAN clustering and the Firefly algorithm and mitigating these problems using various mechanisms. These processes, however, have an impact on the KS-TCP method fault detection efficiency as well as on the time overhead. For this reason, this section conducts many comparative experiments to analyze the effectiveness and time overhead of the DC-TCP method. The experiments focus on answering the following two questions.

(1) RQ1: What is the average fault detection rate of the DC-TCP method, and has the stability improved?
(2) RQ2: Does the DC-TCP method improve the time efficiency of the KS-TCP method?

4.3. Experiment Set. The automation platform for this experiment was built with Python 3.8.10 on Ubuntu 20.04. The Java assembly was compiled using the JDK version V11.0.11. Each set of experiments was repeated more than 300 times, and the average value was used as the final result, and the related statistical analysis was performed.

Four open-source software, Hsqldb, Commons-Cli, Joda Time, and Jsoup, were selected for this experimental study. Also, to further verify the applicability of the black-box static ranking method, this experiment is combined with the combinatorial testing technique for practical testing applications. For the combinatorial strategy, java assemblies [43, 44] used in recent combinatorial testing studies were chosen because of the possible difficulty in covering the general program to be tested due to the insufficient normative description of the Siemens assembly input and the large deviation of the model extracted from the initial test set. They contain different versions and are all highly configurable.

For the parameter configuration of DBSCAN clustering, after a lot of preliminary experiments, we found that for the threshold value e of the combination strategy, if greater than 2, most of the programs will only form one classification. At the same time, the number of classifications will be too large when it is set to 1, which will have some influence on the subsequent sorting steps. Therefore, 2 is chosen as a fixed value in the experiment to ensure a better clustering effect can be obtained. The parameter minpts is used to control the number of samples in the domain, which has less influence compared to e. The experimental setting of 50 is a fixed value.

To further demonstrate the effectiveness of the test case ordering method, the test case set is processed in this experiment. According to the combined dictionary order, only 1 test case that can find software defects is retained. Due to the performance limitation of the experimental platform, the overall number of test cases that reach the million level is kept within 5,000. For the Hsqldb-2.25 version, since each of its test cases is able to find defects, it is treated separately, and all test cases that find the first bug are retained to expand the number of test sets. Table 1 lists the selected real assemblies to be tested with their brief descriptions, version information, number of lines of code, number of classes in the project, input dimensions, and number of defects present.

The key steps in the experiment are shown in detail in Figure 3. First, a mapping model is constructed based on the program's input under test to convert the original test cases into a new test set. Then, the test case distances are calculated using the similarity metric, and the results are saved in a distance matrix for the subsequent prioritization process. Based on the distance matrix, the original test case set is then prioritized using the DC-TCP method, as well as several other comparison methods. This produces execution sequences for each ranking technique while also recording the generation time. Afterward, the execution sequences are reflected in the original type of test sets using the model. The final obtained test sets are executed on the programs to be tested, and their outputs are saved. The sequence numbers of the test cases containing the faults found, and the order in which the faults were found are recorded. Finally, the evaluation metrics of the test cases are calculated based on the program execution records to evaluate the execution sequences generated by each test case prioritization method. Then the execution efficiency of the different sequencing methods is evaluated based on the generation time of the records.

4.4. Evaluation Metrics. We selected the APFD value as the main metric for this experiment, which was first proposed by Rothermel et al. [8]. This metric indicates the speed of defect detection in test case sequences and is widely used to evaluate the effectiveness of prioritization methods. The APFD value ranges from 0% to 100%, with closer to 100% indicating that the method is relatively more effective. Given a test set T

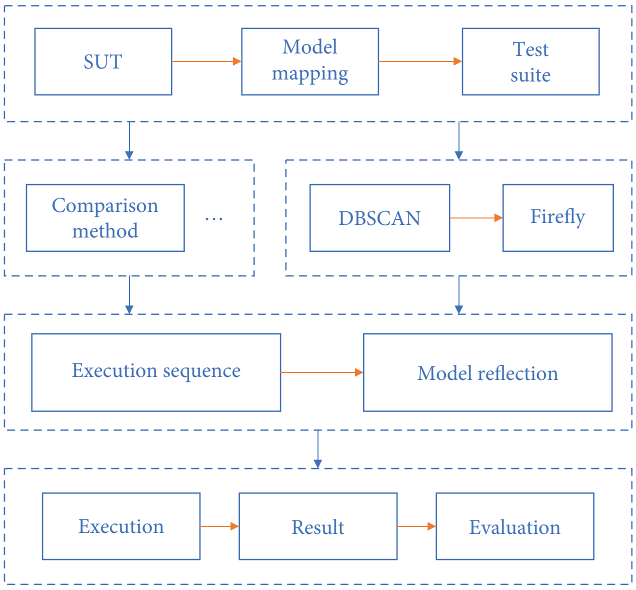


Figure 3: Experiment flow.

containing n test cases and m defects, for a given test execution sequence, TF_i denotes the position in the execution sequence of the first test case in which defect i is detected. The calculation of the APFD value is shown in Equation (1).

$$\mathrm{APFD} = 1 - \frac{\mathrm{TF}_i + \mathrm{TF}_2 + \cdots + \mathrm{TF}_m}{nm} + \frac{1}{2n}. \quad (1)$$

It can be seen from the Equation (1) that the higher the APFD value indicates that the sequence of test cases finds defects faster.

5. Results and Analysis

This section shows the results of the experiments in this section, comparing the DC-TCP method with the SD-TCP method, KS-TCP method, FA-TCP method, and RS-TCP method in terms of effectiveness and efficiency and answering two research questions.

5.1. Validity Analysis. Figure 4 and Table 2 show the effective performance of the DC-TCP, SD-TCP, KS-TCP, FA-TCP, and RS-TCP methods under real programs. Figure 4 is a box plot; the upper and lower bounds of the box represent the upper and lower quartiles of the data, respectively. The horizontal line inside the box denotes the median. The red dot signifies the mean, referring to the arithmetic average of a set of numbers, while black squares represent outliers. In Table 2, the mean corresponds to the red dot in the box plot and the p-value is used to assess the degree of contradiction between the observed data and the null hypothesis in hypothesis testing. It helps determine whether the experimental data are attributable to random chance.

5.1.1. DC-TCP Method vs. SD-TCP Method. Among the 10 Java programs, the average APFD values of the experimental results of the DC-TCP method outperformed those of the SD-TCP method in all cases. This disparity was particularly pronounced in the case of the three versions of the Hsqldb program. Also, there is some significant difference in the p-value. In terms of stability, the DC-TCP and SD-TCP methods share similar box sizes, while the SD-TCP method has some outliers in the sorting results of some programs, while the DC-TCP method performs well. Therefore, the overall comparison results show that the DC-TCP method has some improvement in the effectiveness of the SD-TCP method.

5.1.2. DC-TCP Method vs. KS-TCP Method. Among the 10 Java programs, the average APFD values of the DC-TCP method are either close to or even higher than the KS-TCP method for most of the programs. It is slightly lower than the KS-TCP method in three of the programs. On the top, there is a significant difference in all seven programs except for Commons-Cli-1.2 and Hsqldb-2rc8 programs. In terms of stability, the DC-TCP method's box is flatter and more stable than the KS-TCP method on most programs. Only on the Jsoup-1.9.1 program does the KS-TCP method perform very well, which may be because the critical test cases of Jsoup-1.9.1 are more likely to be selected by the KS-TCP method. Therefore, the DC-TCP method is overall more stable than KS-TCP and can achieve the performance of the KS-TCP method.

5.1.3. DC-TCP Method vs. FA-TCP Method. In this experiment, The results are stable and constant due to the heuristic algorithm search of the FA-TCP algorithm itself. Among the 10 Java programs, the FA-TCP method performed well on the Commons-Cli, Joda-Time, and Jsoup-1.8.3 programs. However, it performed poorly on the other five programs, especially on the Hsqldb-2.25 program, and even worse than the RS-TCP method. The FA-TCP method selected a path to construct the execution sequence using a search algorithm. However, its effect was less stable for the different programs to be tested, and misleading sequencing guidelines may have appeared. Relative to the DC-TCP method, the sequencing process is guided by using an early diversity strategy to perform stably across programs. In terms of aspect, the FA-TCP method has a significant and significant difference due to the algorithm's stability.

The above analysis answers research question 1: The DC-TCP method has some advantages over several baseline black-box static ranking methods with strong significant differences. Compared with the FA-TCP method, although the FA-TCP method shows higher stability on several of the programs, it also shows almost the opposite results on the other programs, which are less stable. Compared with the KS-TCP method, the DC-TCP method exhibits a higher level of stability, attaining and outperforming the KS-TCP method on most of the programs. Thus, DC-TCP ensures algorithmic effectiveness in the process of targeting the stability of the KS-TCP method, while the combined strategy also performs well in terms of applicability.

5.2. Efficiency Analysis. Figure 5 and Table 3 show the performance of the DC-TCP method, SD-TCP method, KS-TCP method, and FA-TCP method in terms of sorting

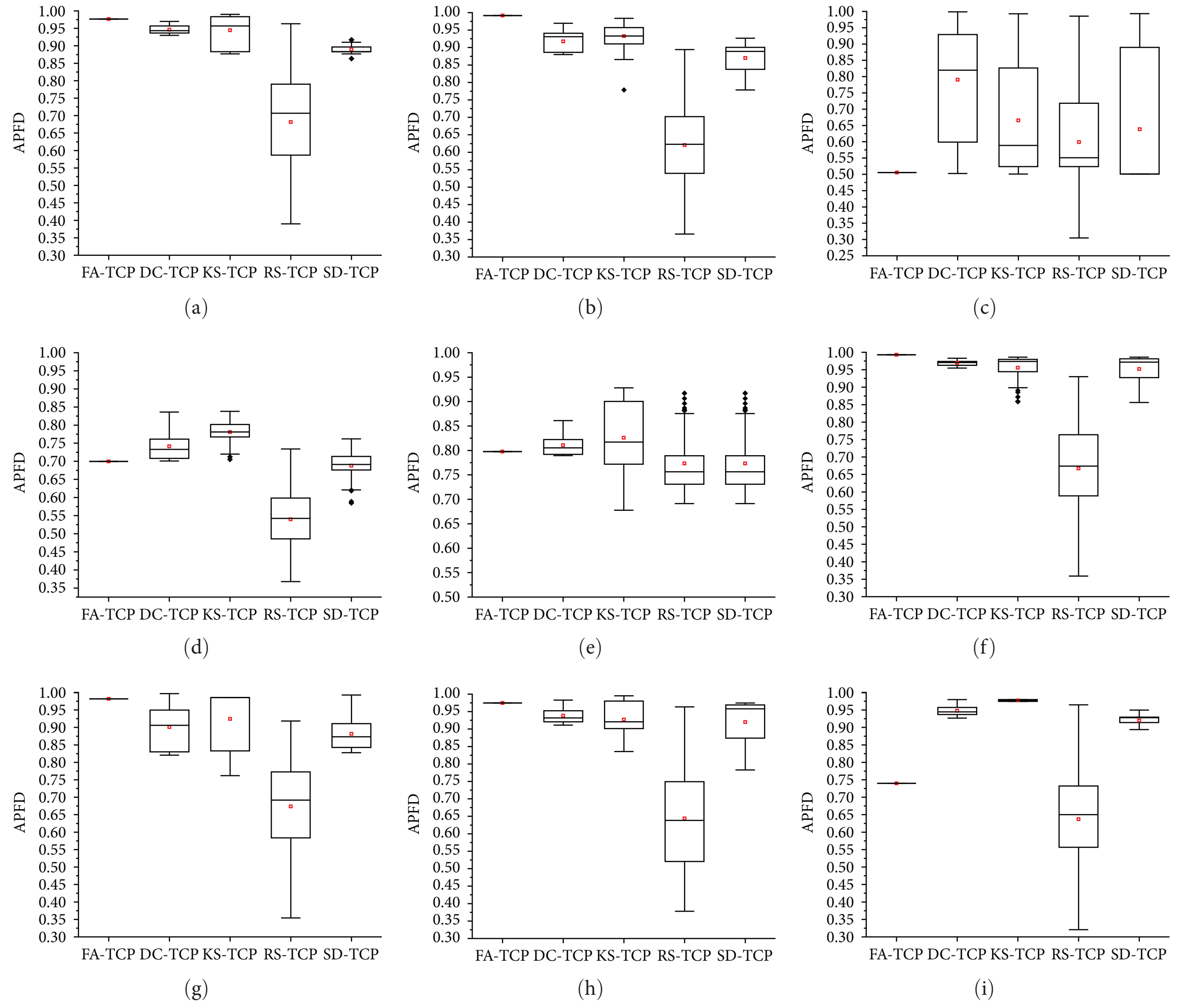


FIGURE 4: The box plot of APFD: (a) cli-1.2, (b) cli-1.3, (c) hsqldb-2.25, (d) hsqldb-2.29, (e) hsqldb-2rc8, (f) joda-time-2.3, (g) joda-time-2.9.1, (h) jsoup-1.8.3, and (i) jsoup-1.9.1.

efficiency under real programs, and the execution time of each method is recorded.

Figure 5 exclusively presents the results of four methods, excluding RS-TCP due to its nature as a random sample TCP method. Comparing its execution time may have limited significance. Nevertheless, the figure effectively illustrates the noticeable improvement achieved by DC-TCP.

It is noteworthy that in most cases, DC-TCP exhibits relatively lower execution times, demonstrating higher performance. Taking Commons-Cli-1.2, Commons-Cli-1.3.1, and Jsoup-1.9.1 as examples, the execution times for DC-TCP are 0.0014, 0.011, and 0.0115 s, respectively, showing a significant performance improvement compared to other methods. This can also be seen from the bar chart; the bar is almost unseeable due to its low time consumption. In some programs, such as Hsqldb-2.2.5, Hsqldb-2.9.1, and Hsqldb-2rc8, DC-TCP also performs exceptionally well. Especially on Hsqldb-2.2.5, the execution time for DC-TCP is significantly lower than the other three methods, with times of 0.6396, 3.2614, and 1.3952 s, respectively.

Overall, In terms of time efficiency, the DC-TCP method shows better results on all data sets, and the KS-TCP method has a greater advantage in the analysis of the experimental results. According to the time complexity analysis in Sections 3.2.4 and 4.2.4, the time consumption of the KS-TCP algorithm mainly comes from the K-medoids clustering process, while the time complexity of the DC-TCP algorithm is also mainly in the DBSCAN process. Where the time complexity of K-medoids is $O(nkt)$, t is the number of iterations, and k is the number of clusters. The time complexity of DBSCAN is $O(n\log(n))$. In the experimental results, the time efficiency of the DC-TCP method is higher than the KS-TCP method in all cases. Although the time complexity of K-medoids can reach the linear level, it may not maintain a strong efficiency advantage for small and medium datasets when the number of iterations and the number of clusters are large.

Table 2: The APFD result of five TCP methods.

Programs	FA-TCP	DC-TCP	KS-TCP	RS-TCP	SD-TCP
Commons-Cli-1.2					
Mean	0.976667	0.946267	0.944444	0.681533	0.889654
p-Value	1.24E-66	—	0.673754	5.10E-45	4.22E-78
Commons-Cli-1.3.1					
Mean	0.991871	0.917703	0.932602	0.620305	0.870224
p-Value	5.41E-67	—	0.000356	4.86E-68	1.79E-18
Hsqldb-2.2.5					
Mean	0.505381	0.790283	0.665543	0.5990850	0.638170
p-Value	6.50E-43	—	2.18E-07	4.01E-15	6.72E-09
Hsqldb-2.9.1					
Mean	0.699793	0.741197	0.780869	0.539773	0.688091
p-Value	3.21E-23	—	5.52E-15	7.21E-58	3.64E-19
Hsqldb-2rc8					
Mean	0.797631	0.810610	0.824234	0.773319	0.773319
p-Value	1.03E-09	—	0.076612	6.11E-10	6.11E-10
Joda-Time-2.3					
Mean	0.992845	0.969778	0.955621	0.667413	0.952232
p-Value	2.89E-65	—	0.000119	2.42E-57	0.000157
Joda-Time-2.9.1					
Mean	0.981834	0.900819	0.924198	0.67372	0.881482
p-Value	5.15E-32	—	0.022778	1.18E-38	0.007497
Jsoup-1.8.3					
Mean	0.974518	0.938182	0.927039	0.643678	0.919821
p-Value	7.50E-43	—	0.016564	2.78E-50	0.007687
Jsoup-1.9.1					
Mean	0.739899	0.947652	0.977525	0.636995	0.919722
p-Value	3.9E-205	—	1.07E-52	2.33E-60	3.43E-27

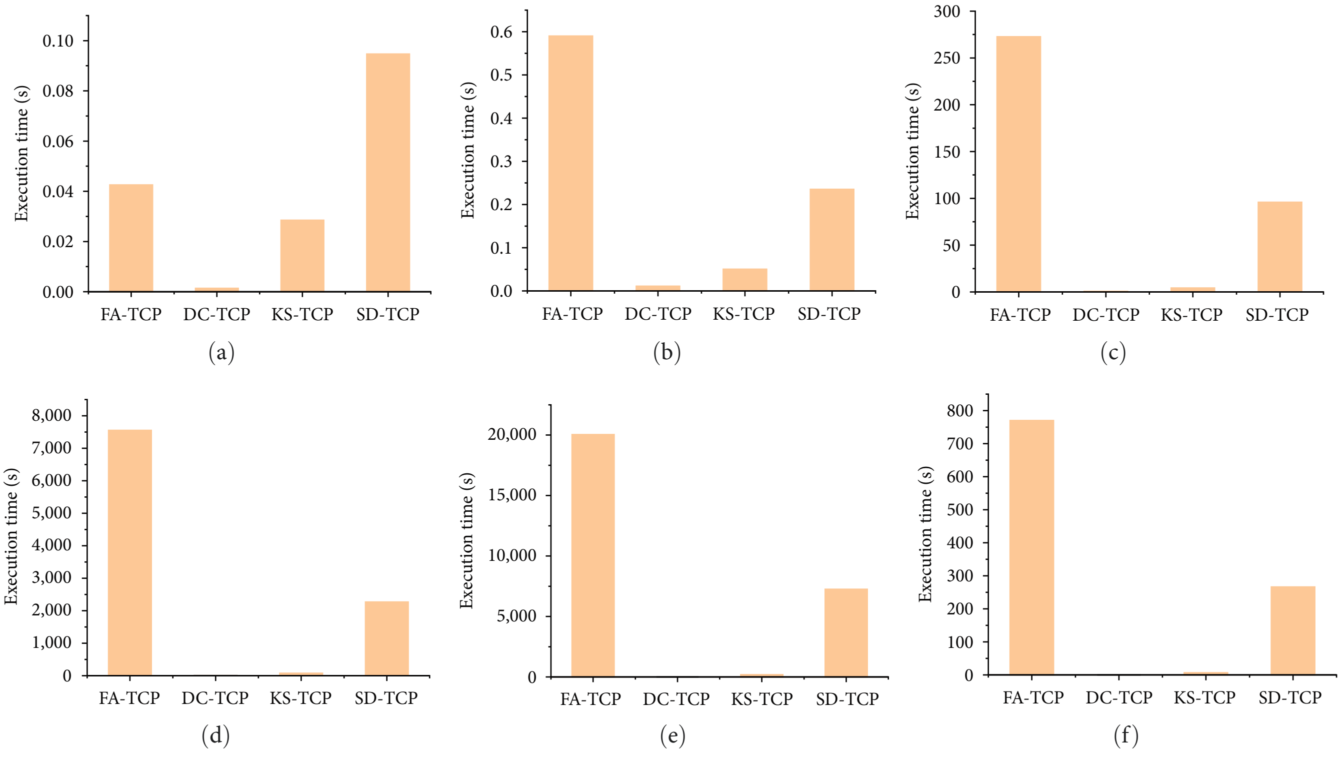
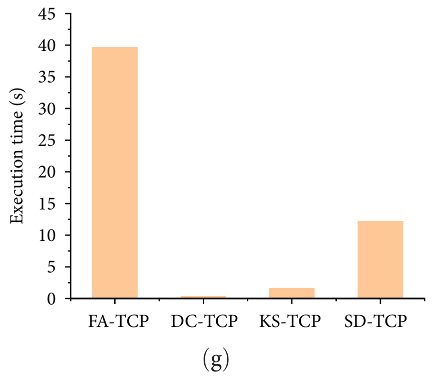
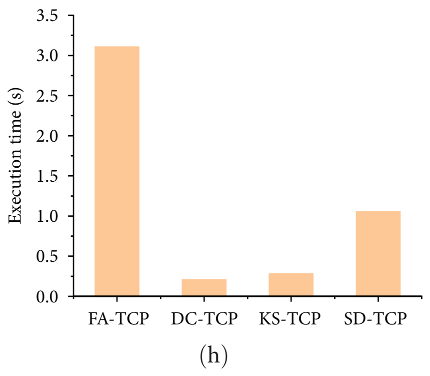


Figure 5: Continued.





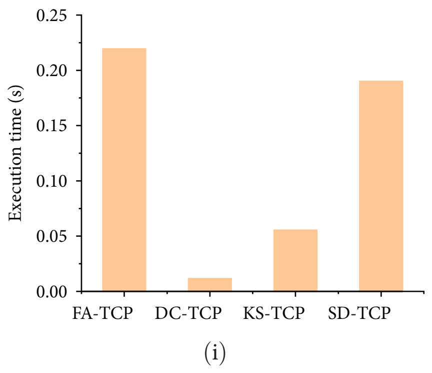


Figure 5: The bar chart of execution time: (a) cli-1.2, (b) cli-1.3, (c) hsqldb-2.25, (d) hsqldb-2.29, (e) hsqldb-2rc8, (f) joda-time-2.3, (g) joda-time-2.9.1, (h) jsoup-1.8.3, and (i) jsoup-1.9.1.

Table 3: The execution time of four TCP methods.

Programs	FA-TCP (s)	DC-TCP (s)	KS-TCP (s)	SD-TCP (s)
Commons-Cli-1.2	0.042576174	0.0013566273	0.0284932321	0.094671390
Commons-Cli-1.3.1	0.589904398	0.011024305	0.050519592	0.235138911
Hsqldb-2.2.5	272.7259341	0.6395944821	4.213021665	95.86287254
Hsqldb-2.9.1	7554.271532	3.261357493	79.11503234	2271.508149
Hsqldb-2rc8	20042.83161	1.395196066	194.3872924	7252.688482
Joda-Time-2.3	770.0889478	0.169968938	6.552740538	266.5852311
Joda-Time-2.9.1	39.60215481	0.281344791	1.553541989	12.14473887
Jsoup-1.8.3	3.106531238	0.205778519	0.281324892	1.052573894
Jsoup-1.9.1	0.219398454	0.011527408	0.055429106	0.190094076

The above analysis answers research question 2: The DC-TCP method significantly improves the efficiency of the black-box static test case sorting method in the Java experimental program set compared to the three black-box static sorting methods compared. The sorting efficiency is still further improved compared to the KS-TCP method. Thus, the DC-TCP method also maintains the efficiency advantage of the KS-TCP method with further improvement in the efficiency of generating execution sequences in the 10 Java programs.

6. Conclusion and Future Work

This paper first details the main elements of the KS-TCP algorithm, points out some of its problems, i.e., low applicability and stability, and proposes relevant improvement schemes. The applicability is improved by introducing the mechanism of combined policies, and the KS-TCP method is balanced in terms of stability and time overhead by combining the DBSCAN algorithm and the Firefly search strategy. Additionally, to accommodate other problems introduced by the improvement scheme, two mechanisms are used to mitigate their impact. Moreover, the DC-TCP method is applied in practice on a combined test. To verify the effectiveness and efficiency of the DC-TCP method, a series of real program experiments were designed and implemented. The KS-TCP, FA-TCP, SD-TCP, and RS-TCP methods were selected for experimental comparison. The experimental results show that, in terms of applicability, the combined strategy can indeed improve the applicability of the black-box static methods. At the same time, it can still maintain the advantage over SD-TCP and several other algorithms in terms of method effectiveness and efficiency. The stability is improved without degrading the effectiveness of KS-TCP.

Further exploration can be conducted on the semantic analysis of test cases. While the DC-TCP method proposed in this paper introduces clustering analysis and combination strategies to improve the sorting performance in test case prioritization, there is still potential to enhance the understanding and analysis capability of the test case semantics. Advanced natural language processing and machine learning techniques can be explored to accurately capture the semantic relationships between test cases and apply them to the sorting process. This can further enhance the accuracy and stability of the sorting. Besides, for the time challenge that still exists with large-scale test sets, further exploration can be done to improve time efficiency. Although the DC-TCP method demonstrates advantages in terms of time efficiency compared to other static black-box sorting methods, there may still be high time costs for extremely large test sets. Future work can consider utilizing techniques such as parallel computing, distributed computing, or optimization algorithms to accelerate the sorting process. Additionally, the algorithm can be further optimized to reduce computational complexity and improve sorting efficiency. By conducting further research and exploration in these two aspects, the performance and practicality of test case prioritization methods can be further enhanced, providing better support for practical applications in the field of software testing.

Data Availability

The data that support the findings of this study are available from the corresponding author upon reasonable request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

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

References

[1] W. Lam, A. Shi, R. Oei, S. Zhang, M. D. Ernst, and T. Xie, "Dependent-test-aware regression testing techniques," in *ISSTA 2020: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 298–311, ACM, USA, 2020.

[2] R. H. Rosero, O. S. Gomez, and G. Rodriguez, "Regression testing of database applications under an incremental software development setting," *IEEE Access*, vol. 5, pp. 18419–18428, 2017.

[3] R. Noemmer and R. Haas, "An evaluation of test suite minimization techniques," in *Software Quality: Quality Intelligence in Software and Systems Engineering*, vol. 371 of *Lecture Notes in Business Information Processing*, pp. 51–66, Springer, Vienna, Austria, 2020.

[4] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.

[5] E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010.

[6] G. Rothermel and M. J. Harrold, "Analyzing regression test selection techniques," *IEEE Transactions on Software Engineering*, vol. 22, no. 8, pp. 529–551, 1996.

[7] D. Hao, L. Zhang, and H. Mei, "Test-case prioritization: achievements and challenges," *Frontiers of Computer Science*, vol. 10, no. 5, pp. 769–777, 2016.

[8] 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.

[9] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran, "Prioritizing test cases with string distances," *Automated Software Engineering*, vol. 19, no. 1, pp. 65–95, 2012.

[10] P. Ramya, V. Sindhura, and P. V. Sagar, "Clustering based prioritization of test cases," in *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*, pp. 1181–1185, IEEE, Coimbatore, India, 2018.

[11] 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*, Article ID e2565, 2023.

[12] J. Chen, Y. Gu, S. Cai, H. Chen, and J. Chen, "KS-TCP: an efficient test case prioritization approach based on k-medoids and similarity," in *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pp. 105–110, IEEE, Wuhan, China, 2021.

[13] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, vol. 19, no. 1, pp. 182–212, 2014.

[14] J. A. P. Lima and S. R. Vergilio, "Test case prioritization in continuous integration environments: a systematic mapping study," *Information and Software Technology*, vol. 121, Article ID 106268, 2020.

[15] C. Malz, N. Jazdi, and P. Göhner, "Prioritization of test cases using software agents and fuzzy logic," in *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pp. 483–486, IEEE, Montreal, QC, Canada, 2012.

[16] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, "Bridging the gap between the total and additional test-case prioritization strategies," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 192–201, IEEE, San Francisco, CA, USA, 2013.

[17] Y. Lu, Y. Lou, S. Cheng et al., "How does regression test prioritization perform in real-world software evolution?" in *Proceedings of the 38th International Conference on Software Engineering*, pp. 535–546, IEEE, Austin, TX, USA, 2016.

[18] D. D. Nardo, N. Alshahwan, L. C. Briand, and Y. Labiche, "Coverage-based test case prioritisation: an industrial case study," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pp. 302–311, IEEE, Luxembourg, Luxembourg, 2013.

[19] 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)*, pp. 392–394, IEEE, Tokyo, Japan, 2017.

[20] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering, ICSE 2001*, pp. 329–338, IEEE, Toronto, Ontario, Canada, 2001.

[21] M. Azizi, "A tag-based recommender system for regression test case prioritization," in *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 146–157, IEEE, Porto de Galinhas, Brazil, 2021.

[22] W. Liu, X. Wu, W. Zhang, and Y. Xu, "The research of the test case prioritization algorithm for black box testing," in *2014 IEEE 5th International Conference on Software Engineering and Service Science*, pp. 37–40, IEEE, Beijing, China, 2014.

[23] J. Chen, H. Chen, Y. Guo, M. Zhou, R. Huang, and C. Mao, "A novel test case generation approach for adaptive random testing of object-oriented software using K-means clustering technique," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 6, no. 4, pp. 969–981, 2022.

[24] D. Marijan, "Comparative study of machine learning test case prioritization for continuous integration testing," *Software Quality Journal*, vol. 31, no. 4, pp. 1415–1438, 2023.

[25] M. Bagherzadeh, N. Kahani, and L. Briand, "Reinforcement learning for test case prioritization," *IEEE Transactions on Software Engineering*, vol. 48, no. 8, pp. 2836–2856, 2022.

[26] V. H. Durelli, R. S. Durelli, S. S. Borges et al., "Machine learning applied to software testing: a systematic mapping study," *IEEE Transactions on Reliability*, vol. 68, no. 3, pp. 1189–1212, 2019.

[27] B. Busjaeger and T. Xie, "Learning for test prioritization: an industrial case study," in *FSE 2016: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 975–980, Association for Computing Machinery, New York, NY, USA, 2016.

[28] A. Bertolino, A. Guerriero, B. Miranda, R. Pietrantuono, and S. Russo, "Learning-to-rank vs ranking-to-learn: strategies for regression testing in continuous integration," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 1–12, ACM, 2020.

[29] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, and I. Schaefer, "System-level test case prioritization using machine learning," in *2016 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 361–368, IEEE, Anaheim, CA, USA, 2016.

[30] 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, Article ID 106269, 2020.

[31] H. Srikanth, L. A. Williams, and J. A. Osborne, "System test case prioritization of new and regression test cases," in *2005 International Symposium on Empirical Software Engineering (ISESE 2005)*, pp. 64–73, IEEE, Noosa Heads, QLD, Australia, 2005.

[32] H. Srikanth, C. Hettiarachchi, and H. Do, "Requirements based test prioritization using risk factors: an industrial study," *Information and Software Technology*, vol. 69, pp. 71–83, 2016.

[33] T. Muthusamy and K. Seetharaman, "A new effective test case prioritization for regression testing based on prioritization algorithm," *International Journal of Applied Information Systems (IJAIS)*, vol. 6, no. 7, pp. 21–26, 2014.

[34] A. Panichella, R. Oliveto, M. D. Penta, and A. De Lucia, "Improving multi-objective test case selection by injecting diversity in genetic algorithms," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 358–383, 2015.

[35] F. Yuan, Y. Bian, Z. Li, and R. Zhao, "Epistatic genetic algorithm for test case prioritization," in *Search-Based Software Engineering*, vol. 9275 of *Lecture Notes in Computer Science*, pp. 109–124, Springer, Bergamo, Italy, 2015.

[36] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Transactions on Software Engineering*, vol. 33, no. 4, pp. 225–237, 2007.

[37] W. Zhang, Y. Qi, X. Zhang, B. Wei, M. Zhang, and Z. Dou, "On test case prioritization using ant colony optimization algorithm," in *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 2767–2773, IEEE, Zhangjiajie, China, 2019.

[38] S. Li, N. Bian, Z. Chen, D. You, and Y. He, "A simulation study on some search algorithms for regression test case prioritization," in *2010 10th International Conference on Quality Software*, pp. 72–81, IEEE, Zhangjiajie, China, 2010.

[39] G. Fraser and F. Wotawa, "Test-case prioritization with model-checkers," in *25th conference on IASTED International*, pp. 1–6, Citeseer, 2007.

[40] B. Korel, G. Koutsogiannakis, and L. H. Tahat, "Application of system models in regression test suite prioritization," in *24th IEEE International Conference on Software Maintenance (ICSM 2008)*, pp. 247–256, IEEE, Beijing, China, 2008.

[41] M. Khatibsyarbini, M. A. Isa, D. N. A. Jawawi, H. N. A. Hamed, and M. D. M. Suffian, "Test case prioritization using firefly algorithm for software testing," *IEEE Access*, vol. 7, pp. 132360–132373, 2019.

[42] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: a family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[43] X. Niu, C. Nie, J. Y. Lei, H. Leung, and X. Wang, "Identifying failure-causing schemas in the presence of multiple faults," *IEEE Transactions on Software Engineering*, vol. 46, no. 2, pp. 141–162, 2020.

[44] X. Niu, C. Nie, H. Leung et al., "An interleaving approach to combinatorial testing and failure-inducing interaction identification," *IEEE Transactions on Software Engineering*, vol. 46, no. 6, pp. 584–615, 2020.