Test Case Prioritization Algorithm Based on Improved Code Coverage

Yanan Zhu, Feng Liu

***Abstract*—The objective of test case prioritization is to improve failure detection rates by executing more important test cases earlier. Due to its simplicity and effectiveness, the greedy algorithm based on code coverage is widely utilized. However, when adopting the coverage information, the current technology only considers the number of code units covered by test cases, regardless of the importance of the code units. This paper employs the approach as the coverage granularity to construct a method weight definition model (MWDM), which extends the traditional coverage algorithm model and proposes an improved greedy algorithm based on MWDM (MWDM-IGA), in order to address the aforementioned issue. At the same time, to utilize the project change history, a file-level defect prediction model is introduced into the above algorithm. Experimental validation on the real-error dataset——Defects4j demonstrates that MWDM-IGA outperforms other algorithms based on overlay information in terms of average failure detection rate (APFD) and the introduction of defect prediction into TCP will further improve the performance of the algorithm. During prioritization, the occurrence of 'ties' caused by multiple test cases covering the same number of code units decreases, thereby significantly enhancing the efficiency of the algorithm.**

***Index Terms*—test case priority, coverage granularity, method weight, greedy algorithm, defect prediction.**

I. Introduction

IN the contemporary software development process, agile development is gradually gaining a foothold, and consequently, continuous integration is becoming increasingly prevalent. When updating and iterating software, regression tests are conducted. Nonetheless, regression testing is a time-intensive endeavor. To improve the cost-effectiveness of regression testing, various techniques have been proposed, including test case selection [1], test case reduction [1], and test case prioritization [2]. The first two methods, as their names imply, will discard test cases and may overlook crucial ones. Therefore, test case prioritization techniques provide the optimal balance between security and cost-effectiveness.

Test Case Prioritization (TCP) seeks to determine the execution order of test cases to achieve intermediate goals, thereby maximizing test efficiency and system reliability at the fastest possible rate. These intermediate objectives include achieving maximum code coverage as quickly as possible [3]–[8], maximizing test case diversity [9]–[12], balancing costs and benefits [13]–[16], and detecting faults as soon as possible [17]–[19], etc. According to the statistical results [20], coverage-aware prioritization methods prevail, and such algorithms utilize structural coverage as a criterion with the assumption that test cases with greater coverage may exhibit a higher probability of detecting faults. Greedy strategies excel in the process of prioritization [21], ranking test cases by the number of code units covered. The commonly employed greedy strategies can be categorized as either total strategies or additional strategies. The total strategy sorts the test cases by the number of code elements they cover in descending order. While the additional strategy is an overall strategy that introduces a feedback strategy and iteratively determines the next test case that provides the maximum coverage of code units not yet covered by previously prioritized test cases. Existing prioritization algorithms based on coverage information, however, default to assigning all code units the same level of significance, which is frequently inconsistent with reality. In addition, [22] demonstrated that when traditional coverage information is employed to rank test cases, it is easy to have a 'tie' phenomenon, i.e., multiple test cases covering the same number of code units. In contrast, if they are not processed further, a random algorithm will be used to select the next test case, which will significantly reduce the efficiency of the algorithm.

Based on the above problems, the method will serve as the coverage granularity, based on the dependency relationship between methods, a weight definition model is created first. The concept of weight is then introduced into the traditional coverage model and applied to the two greedy algorithm strategies. Finally, we introduce file-level defect prediction models into TCP to continue to enhance the defect detection capabilities of the ranking algorithm; The paper is organized as follows: in section II, we present the work of others that is relevant to this article. Section III describes our proposed MWDM-GA method in detail and combined with defect prediction to guide test case sequencing. In section IV, we demonstrate the experimental outcomes of our methodology. We conclude with a summary of this paper in the final chapter.

II. Related Work

A. Test case prioritization based on coverage information

Coverage-based ranking techniques aim to maximize the coverage of program code units, such as statements, branches, and methods, by executing test cases with high coverage as soon as possible. Maximum code coverage is a secondary objective; the ultimate objective is to increase the fault detection rate. In recent years, numerous coverage-based ranking

Manuscript received September 01, 2022; revised April 12, 2023.

This work was supported by he Laboratory Research Fund Program (K21L01050)

Yanan Zhu is a postgraduate student of Beijing Jiaotong University, Beijing,100044, China (email:yananzhu@bjtu.edu.cn).

Feng Liu is a professor of Computer Information and Technology Department, University of Beijing Jiaotong , Beijing,100044, China(corresponding author to provide email:fliu@bjtu.edu.cn).

methods have been proposed. Huang et al. address the issue of the detrimental effects of considering code units separately and in isolation in [5] and suggest a new coverage criterion, code combination coverage, which is applied to the ranking algorithm to increase the fault detection rate. To boost user satisfaction with high-quality software and ultimately to increase the rate at which serious vulnerabilities are discovered, Krishnamoorthi et al. [23] proposed a system-level test case prioritization model starting from the specification of software requirements. Mei et al. [24] address the issue of missing or inaccurate data when collecting dynamic coverage by utilizing the dynamic call graph of the program in the Junit test suite. This accomplishes the estimation of each test case's ability to obtain code coverage and demonstrates that using static coverage does not significantly reduce the efficacy of fault detection. zhang et al. [25] decompose the test requirements into test points, propose a new evaluation strategy, Average Percentage Test-point Cover(APTC), and apply it to the fitness function of the genetic algorithm to guide the process of TCP. Fu et al. [26] proposed a new test case prioritization algorithm based on program changes and method invocation relationships, demonstrating that changed methods and methods with more invocations are riskier and simpler to detect failures. According to the evidence of Hao et al [27], the simple additive coverage-based technique's final ranking results are only marginally inferior to the optimal technique.

The majority of the literature, however, applies prioritization methods to code units of different granularities or proposes new evaluation methods without considering the possibility that different code units feature varying levels of importance.

B. *Test case prioritization based on defect prediction*

In order to take advantage of the project's change history, many algorithms have introduced various defect prediction algorithms into the prioritization of test cases.Wang et al. [28] considered the distribution of errors in the source code, and first proposed a quality-aware test case prioritization, that is, using code inspection techniques such as statistical defect prediction models and static error finders to detect error-prone source code, Then adjust the existing coverage-based TCP algorithm by considering the weighting of the source code.Inspired by this work, Mahdieh et al. [29] used the defect history of software to introduce defect prediction methods to learn neural network models, use this model to estimate the failure propensity of each code unit of source code, and then incorporate these estimates into coverage-based TCP method.Mahdieh et al. [11] recently proposed a TCP technique that considers test case coverage data, error history, and test case diversification, first based on using a clustering method to group similar test cases, and then grouping test cases between clusters Prioritized, and finally achieved performance optimization.To leverage test history, Engstrom et al. [30] propose to exploit previously fixed bugs to select a small subset of test cases for regression test selection. Laali et al. [31] proposed an online TCP approach that exploits fault locations revealed by executed test cases in order to prioritize unexecuted test cases. Kim et al. used methods from fault localization to improve the prioritization of test cases, and they exploited the observation that defects were fixed after they were found, proposing that test cases that covered previous faults were less likely to find faults [32].

However, as mentioned in the related work mentioned above, the defect prediction used in TCP always uses methods such as machine learning to predict the number and types of defects in software projects, but due to the extreme imbalance of positive and negative samples in the Defects4j dataset, using Machine learning algorithms to predict defects in code units can lead to severe bias and inaccuracy in predictive models. Therefore, we employ a time-weighted risk algorithm as a predictive model, which is simple but has been shown to perform similarly to more sophisticated methods [33], which is consistent with the idea of Occam's razor. At the same time, various indicators are used in the forecasting algorithm to enhance the robustness and robustness of the model.

C. *Commonly used evaluation indicators*

The average percentage of fault detection (APFD) is the most significant processing priority indicator. The APFD [34] is determined as follows:

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n} \tag{1}$$

where n is the number of test cases in the test case set, m is the number of errors that can be detected in this test case set, $\mathbf{TF}_i$ denotes the order of the test cases in the sequence in which defect i is first found in the sorted set of test cases T'. APFD is a non-negative value, and a higher APFD value indicates faster error detection for this test sequence. Additionally, there are variants of APFD metrics, such as $APFD_c$ [35] and NAPFD [36]. They are described respectively in equations (2) and (3).

$$APFD_c = \frac{\sum_{i=1}^{m} se_i \times \left(\sum_{j=TF_i}^{n} time_j - \frac{1}{2} time_{TF_i} \right)}{\sum_{i=1}^{n} time_j \times \sum_{i=1}^{m} se_i} \tag{2}$$

$$NAPFD = p - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{p}{2n} \tag{3}$$

where n is the number of test cases in the test case set, m is the number of errors that can be detected in this test case set, $\mathbf{TF}_i$ denotes the order of the test cases in the sequence in which defect i is first found in the sorted set of test cases T',$\mathbf{se}_i$ denotes the severity of the i-th fault, $\mathbf{time}_j$ represents the execution time of the j-th test case, p is the ratio of the number of faults detected in the sorted test case sequence T' to the total number of faults in the program. According to the formula, it can be noted that $APFD_c$ takes into account both the execution time of the test cases and the fault severity. NAPFD considers that the set of test cases may not detect all faults or may not be able to execute all test cases due to resource constraints.

Scholars have also proposed a metric APxC [37] capable of covering all code units faster, where x refers to the granularity of code units, which can be statements, branches, code blocks, methods, loops, etc. APxC is calculated as shown in (4).

$$APxC = 1 - \frac{TX_1 + TX_2 + \cdots + TX_m}{nm} + \frac{1}{2n} \quad (4)$$

where n is the number of test cases in the test case set, m is the number of errors that can be detected in this test case set, $\mathbf{TX}_i$ is the first test case in order T' of T which covers code unit i. APxC measures the weighted average of the percentage of code units that have been tested over the lifetime of a test suite.

III. IMPROVED GREEDY ALGORITHM BASED ON METHOD WEIGHT DEFINITION MODEL AND DEFECT PREDICTION

In this section, we introduce coverage with weights-based information and fault prediction to guide the TCP process.

An overview of our improved greedy approach is given in Fig.1, which contains three stages: Obtain the dependency relationship and dependency depth between methods, Assign Weights to Methods, and Prioritization. The first two stages are MWDM's process, where we perform static analysis of the source code, get a list of methods and extract dependencies, and then analyze the dependency hierarchy of the functions, assigning weights to all methods according to the dependency depth. The third stage is to obtain the weighted coverage information of the test cases and guide the formation of the final sorted sequence.

The Fig. 2 is the process of introducing defect prediction into TCP to jointly guide sequencing. First, use the defect prediction algorithm to sort the error probability of the file in descending order, and use the MWDM-IGA algorithm to sort the corresponding test cases inside the file.

A. Method Weight Definition Model (MWDM)

In this paper, we argue that it is unreasonable for the default level of importance for different methods in the prioritization algorithm to be the same because software unit faults are not distributed evenly. Therefore, we intend to assign varying weights to various methods. In this paper, we simply use inter-method invocation data to determine the relative importance of each method i.e., we assume that the importance of a method increases if it is considered directly or indirectly by more methods. We first obtain the direct dependencies between methods using an analysis tool for code. Then we obtain the indirect dependencies between methods. Afterward, we can get the number of invocations of each method. The final step is to normalize the number of invocations of each method as its weight. We set the normalization interval to [0.1, 0.9] since we can conceive of a method that is ineffective for fault detection. As a result, the lower limit is set to 0.1 and the upper limit is taken freely. Algorithm 1 provides the MWDM pseudocode.

Algorithm 1: Main process of MWDM

```
Input:
   Direct Dependency Matrix $[\mathbf{C}(i,j)]_{m\times m}$
1. Initialization:
   weight vector wv← $[0,\cdots,0]_m$,rech_matrix←C
2. while rech_matrix × C ! = rech_matrix do
3.     rech_matrix←rech_matrix × C
4. end while
5. for i← 0 to $m-1$ do
6.     $wv[i] \leftarrow sum(rech\_matrix[\,][i])$
7. end for
8. for i← 0 to $m-1$ do
9.     $wv[i] \leftarrow 0.1 + (0.9-0.1) \times \frac{wv[i]-min(wv)}{max(wv)-min(wv)}$
10.end for
Output: $wv[i]_{1\times m}$.
```

B. The Improved greedy Algorithm Based on MWDM (MWDM-IGA)

In the traditional greedy algorithm for sorting test cases, the number of methods covered by the test cases influences the final sequence; that is, if two test cases cover the same number of methods, they are considered to have the same ability to detect failures, resulting in a 'ties' situation. Without additionally addressing this 'tie' situation, the algorithm is randomly ordered, which drastically reduces the algorithm's efficiency. In this paper, method weights are incorporated into the ranking procedure. Consequently, the final sequence is affected not only by the number of methods covered by the test cases, but also by the weights corresponding to those methods.

MWDM-IGA consists of the MWDM-based conventional greedy algorithm (MWDM-TGA) and the MWDM-based additional greedy algorithm (MWDM-AGA). The pseudocodes for MWDM-TGA and MWDM-AGA are shown in Algorithms 2 and 3, respectively.

C. TCP merging MWDM-IGA and Defect Prediction

In order to apply project change history, we want to introduce defect prediction models into TCP. This idea is supported by experiments by Lewis et al. [38], who demonstrate that defect prediction methods can be used to automate tasks such as prioritization of test cases prioritization. The TCP process of fusing defect prediction and coverage information is shown in the Fig. 2.

The time-weighted risk algorithm used by Google only checks files related to bug fixes. However, new submissions of projects may sometimes add new features even though they are not intended to fix bugs. At this time, we cannot ignore them. Evidence shows that for files modified by multiple developers, the probability of errors becomes greater [39], and the two best predictors of errors are a priori errors and previous changes [40]. So we expand the factors that affect the occurrence of defects, we pay attention to each code submission and multi-developer modification instead of only focusing on defect fixes, and simplify the standardized timestamp used in the original model to the version number of the project. The improved file risk score calculation method is shown in equations (5)-(7).

$$Score = r_\omega \times revisions + d_\omega \times develops + a_\omega \times arrivals \quad (5)$$

$$TWR(t_i) = \frac{1}{e^{-12t_i+12}} \quad (6)$$

$$t_i = \frac{Version_{commit}}{Version_{total}} \quad (7)$$

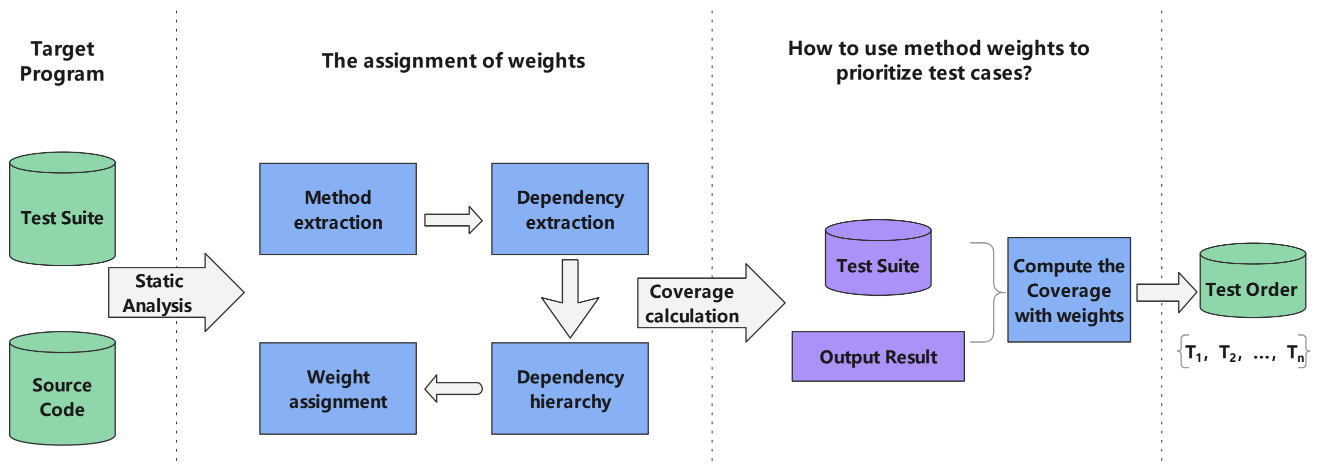

Fig. 1: Overview of the proposed MWDM-IGA.

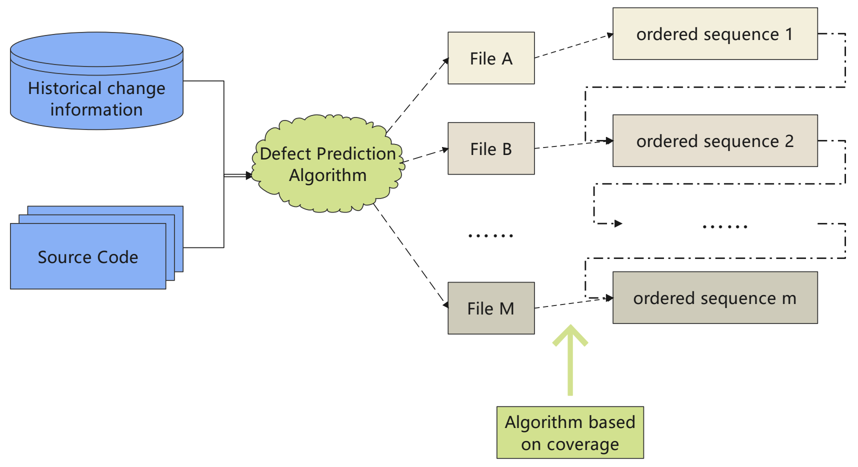

Fig. 2: Overview of TCP incorporate Defect Prediction.

where r_ω, d_ω, and a_ω respectively represent the weights of the three factors(Changes to the file, files modified by multiple developers, whether the file was pulled in on an update) affecting the final risk score of the file, and the sum of their weights is 1. revisions is the twr value based on the number of times the file has been changed in all submissions, develops is the twr value based on the number of times the file has been modified by multiple developers in all submissions, and arrivals is based on the twr value calculated when the file is newly introduced. $Version_{commit}$ represents the current version number when the code submission occurs, and $Version_{total}$ represents the total version number of the project.

Algorithm 2: Main process of MWDM-TGA

```
Input:
    coverage Matrix $[\mathbf{M}(i,j)]_{n\times m}$,weight vector $\mathbf{wv}[0,\cdots,m-1]$
1. Initialization:
    Prioritized Suite PS← {} ,ties number N← 0 ,
    coverage rate cv ← $[0,\cdots,0]_{1\times n}$
2. for $i \leftarrow 0$ to $n-1$ do
3.      $cv[i] \leftarrow$ the inner product of the $i$−th row of M and wv
4. end for
5. while $i \leftarrow 0 < n-1$ do
6.      if the maximum value in $cv[n]$ is not unique then
7.          $N \leftarrow N+1$
8.      end if
9.      select a highest value $cv[l]$ in cv
10.     $PS \leftarrow PS \bigcup \{l\}$
11.end while
Output: $PS, N$.
```

IV. Experiments of MWDM-IGA

Hao et al. [27] demonstrate that intermediate goals cannot be pursued excessively and that the optimal backpack strategy based on coverage test case prioritization is not as competitive in terms of the ultimate goal as the additional greedy strategy (fault detection rate). However, it is impractical to apply a fault detection rate to guide the prioritization process as it is impossible to determine whether a test case can detect a fault until it is executed and a fault is discovered. Therefore, we decided to optimize the coverage-based greedy strategy further and combine it with defect prediction models to guide the ranking process. Since there is a one-to-one or one-to-many relationship between source files and test cases in defects4j, it is logical to apply file-level defect prediction algorithms to

Algorithm 3: Main process of MWDM-AGA

```
Input:
  coverage Matrix $[\mathbf{M}(i,j)]_{n\times m}$,weight vector $\mathbf{wv}[0,\cdots,m-1]$
1. Initialization:
  Prioritized Suite PS← {} , Uncovered method Set UcS,
  ties number N← 0 , Covered method Set CS← {}, vec ← $wv$
2. UcS ← all methods
3. while The number of elements in the set PS < $n$ do
4.     for $i \leftarrow 0 < n-1$ do
5.         $cv[i]$ ← the inner product of the $i-$th row of M and vec
6.     end for
7.     if the maximum value in $cv[0,\cdots,n-1]$ is not unique then
8.         $N \leftarrow N+1$
9.     end if
10.    if $cv[l] = maxinum(cv)$ and l not in Set PS then
11.        $PS \leftarrow PS \bigcup \{l\}$
12.        CS ← $CS \bigcup$ all methods covered by the $i$-th test case
13.        UcS ← $UcS - CS$
14.        Set the $l-$th row of matrix M to 0
15.        Set the weights of all methods covered by the $l-$th test case
           in vec to 0
16.    end if
17.    if Number of elements in the set UcS = m then
18.        $UcS \leftarrow \{\}$
19.    end if
20.    if vec = 0 then
21.        $vec \leftarrow wv$
22.    end if
23.end while
Output: $PS, N$.
```

TCP.

We conduct experiments on the public dataset Defects4j [41], which is a database and extensible framework that provides Java source code and Junit test cases for various projects. Moreover, this dataset stores in a database the actual bugs discovered during each update cycle. Due to its integrity and accessibility, this dataset is popular among software engineering academics.

To demonstrate the effectiveness of the method, we compare the MWDM-GA proposed in this paper with other TCP-based algorithms based on coverage or different code granularity, and Table I shows the different techniques we compaerd. Afterwards, the effectiveness of introducing defect prediction into MWDM-GA is further verified.

A. Experiment preparation

Since Defects4j does not provide the required direct dependencies between methods, we utilize the code static analysis tool UnderStand[1] to obtain them and examine the methods covered by each junit test case. After data collection was complete, we chose as our evaluation metrics the most commonly used APFD values. In this paper, we only detect defects in the latest version of each Java project, which contains only one real bug; therefore, the APFD calculation has been simplified to Equation 8. Consequently, our evaluation metric corresponds to the proportion of test cases executed before a fault is detected; that is, the smaller the value of TF_1/n, the faster the defect detection speed of the algorithm. In addition, we intended to minimize algorithm inefficiency due to the occurrence of random phenomena during the prioritization process, so the rate of reduction in the number of 'ties' is included as an additional evaluation criterion. The calculation is illustrated by Equation 9.

$$APFD = 1 - \frac{TF_1}{n} + \frac{1}{2n} \quad (8)$$

$$DR = \frac{Num_{old} - Num_{new}}{Num_{old}} \quad (9)$$

where Num_{old} is the number of 'ties' occurring in the algorithm before the improvement, Num_{new} is the number of 'ties' that occur in the improved algorithm.

All the experiments in this paper were implemented in Python and run on a Windows system with a CPU(Intel(R) Core I5-12500H @ 2.50GHz) and 16GB of RAM.

B. Experimental Results

We have designated five open-source Java programs from Defects4j on Github: jfreeechart, jodatime, commons.lang, commons.math, and the Closure Compiler. Each version of these programs contains at least one real and exploitable bug; each program corresponds to its own set of junit test cases; and at least one test case exists to detect the bugs in each program. The MWDM-GA experimental results are shown in Tables II, III, and Fig.3. Further verification of the effectiveness of introducing defect prediction into TCP is shown in the Fig. 4.

The values of TF_1/n for the different TCP technologies are compared in Table II, the higher the value, the earlier the defect was discovered. It can be seen in the table that the MWDM-IGA proposed in this paper can detect defects earlier than other algorithms. Table III depicts the reduction rate of MWDM-TGA and MWDM-AGA compared to the average number of 'tie' times with other TCP technologies. Fig.3 depicts the APFD values for various TCP technologies. Based on the boxplot, we can find that our method always stands out in terms of average failure detection capability, whether compared with technologies such as GTS, GAB, and GA-APTC at different granularities or compared with GTM and GAMS based on the same coverage criteria. Fig. 4 verifies the effectiveness of introducing defect prediction into MWDM-GA, where DPTGA and DPAGA are the algorithms after introducing defect prediction in the two strategies respectively.

The preceding experimental results illustrate that, when compared to traditional greedy algorithm, our proposed MWDM-IGA performs better in terms of both faster fault detection and a reduction in tie-breaks between test cases. At the same time, the application of defect prediction in TCP will make the sorting algorithm more superior.

V. Conclusion

In this paper, we propose a prioritization algorithm that uses the weights of the methods in the coverage model and fault prediction to direct the greedy algorithm in completing the ranking. This strategy factors in the fact that actual defects are not distributed evenly throughout the code units. We also seek to reduce the incidence of random occurrences during the prioritization process and want to take advantage of the project change history. Experiments conducted on five open-source projects containing actual bugs demonstrated several

[1]https://scitools.com/

TABLE I: Studies TCP techniques

Tag	Description
GTS	Greedy total(statement-level)
GAS	Greedy additional(statement-level)
GTB	Greedy total(block-level)
GAB	Greedy additional(block-level)
GTM	Greedy total(method-level)
GAM	Greedy additional(method-level)
GA-APTC	Genetic algorithms based APTC
GTMS	Greedy total based on static coverage(method-level)
GAMS	Greedy additional based on static coverage(method-level)
MWDM-TGA	Greedy total based on our improve coverage(method-level)
MWDM-AGA	Greedy additional based on our improve coverage(method-level)

TABLE II: Percentage of test cases executed before the fault was detected for different technologies

Subject Programs	jfreechart	jodatime	commons.lang	commons.math	commons.compiler	**average**
GTS	28.03	25.92	41.89	45.55	27.31	**33.73**
GAS	27.69	23.67	35.44	41.32	29.89	**31.60**
GTB	32.76	25.99	36.54	39.26	26.11	**32.13**
GAB	30.07	31.24	33.56	39.06	28.57	**32.50**
GTM	31.73	26.81	40.3	38.19	28.42	**33.09**
GAM	38.47	22.49	42.03	31.61	20.17	**30.96**
GA-APTC	35.77	30.44	40.03	38.79	28.57	**34.72**
GTMS	30.59	28.92	35.87	39.22	25.69	**32.06**
GAMS	33.78	24.63	41.59	32.19	27.11	**31.86**
MWDM-IGA	31.78	21.55	38.62	32.46	22.87	**29.46**
MWDM-AGA	36.42	20.93	39.9	29.06	16.46	**28.55**

improvements in fault detection rate and tie reduction. These scenarios arise when multiple test cases receive equal coverage during the prioritization process.

REFERENCES

[1] S. Yoo and M. Harman, “Regression testing minimization, selection and prioritization: a survey,” *Software testing, verification and reliability*, vol. 22, no. 2, pp. 67–120, 2012.

[2] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, “Test case prioritization: An empirical study,” in *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM’99).’Software Maintenance for Business Change’(Cat. No. 99CB36360)*. IEEE, 1999, pp. 179–188.

[3] ——, “Prioritizing test cases for regression testing,” *IEEE Transactions on software engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[4] C. Fang, Z. Chen, and B. Xu, “Comparing logic coverage criteria on test case prioritization,” *Science China Information Sciences*, vol. 55, no. 12, pp. 2826–2840, 2012.

[5] R. Huang, Q. Zhang, D. Towey, W. Sun, and J. Chen, “Regression test case prioritization by code combinations coverage,” *Journal of Systems and Software*, vol. 169, p. 110712, 2020.

[6] I. Alazzam and K. M. O. Nahar, “Combined source code approach for test case prioritization,” in *Proceedings of the 2018 International Conference on Information Science and System*, 2018, pp. 12–15.

[7] U. Badhera, G. Purohit, and D. Biswas, “Test case prioritization algorithm based upon modified code coverage in regression testing,” *International Journal of Software Engineering & Applications*, vol. 3, no. 6, p. 29, 2012.

[8] P. Konsaard and L. Ramingwong, “Total coverage based regression test case prioritization using genetic algorithm,” in *2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*. IEEE, 2015, pp. 1–6.

TABLE III: the reduction rate (%) of the number of 'ties' for different programs

Subject Programs	jfreechart	jodatime	commons.lang	commons.math	clourse compiler	average
MWDM-TGA	40.03	8.33	19.35	22.46	27.55	21.54
MWDM-AGA	10.45	23.47	36.21	13.54	16.88	20.11

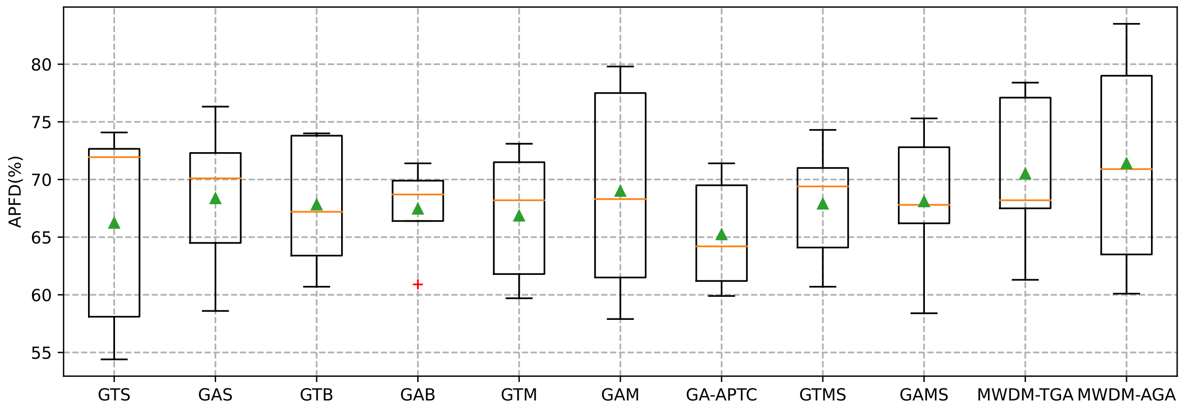

Fig. 3: APFD values for different tcp technologies.

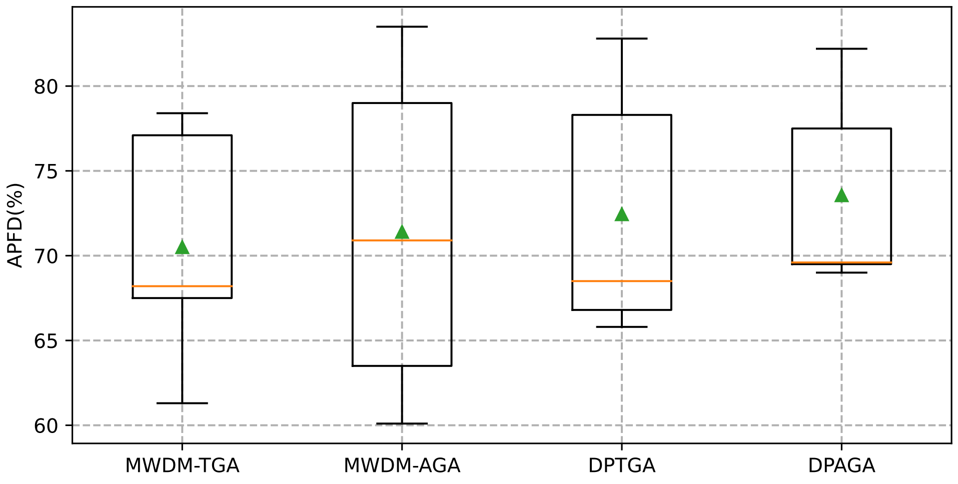

Fig. 4: Further verification.

[9] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, H. N. A. Hamed, and M. D. M. Suffian, "Test case prioritization using firefly algorithm for software testing," *IEEE access*, vol. 7, pp. 132 360–132 373, 2019.

[10] J. Chen, L. Zhu, T. Y. Chen, D. Towey, F.-C. Kuo, R. Huang, and Y. Guo, "Test case prioritization for object-oriented software: An adaptive random sequence approach based on clustering," *Journal of Systems and Software*, vol. 135, pp. 107–125, 2018.

[11] M. Mahdieh, S.-H. Mirian-Hosseinabadi, and M. Mahdieh, "Test case prioritization using test case diversification and fault-proneness estimations," *Automated Software Engineering*, vol. 29, no. 2, pp. 1–43, 2022.

[12] R. Huang, Y. Zhou, W. Zong, D. Towey, and J. Chen, "An empirical comparison of similarity measures for abstract test case prioritization," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2017, pp. 3–12.

[13] A. G. Malishevsky, J. R. Ruthruff, G. Rothermel, and S. Elbaum, "Cost-cognizant test case prioritization," Citeseer, Tech. Rep., 2006.

[14] Y. Wong, H. Zeng, H. Miao, H. Gao, and X. Yang, "The cuckoo search and integer linear programming based approach to time-aware test case prioritization considering execution environment," in *International Conference on Collaborative Computing: Networking, Applications and Worksharing*. Springer, 2018, pp. 734–754.

[15] R. Mukherjee and K. S. Patnaik, "Introducing a fuzzy model for cost cognizant software test case prioritization," in *TENCON 2019-2019 IEEE Region 10 Conference (TENCON)*. IEEE, 2019, pp. 504–509.

[16] S. Dirim and H. Sozer, "Prioritization of test cases with varying test costs and fault severities for certification testing," in *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2020, pp. 386–391.

[17] T. B. Noor and H. Hemmati, "Studying test case failure prediction for test case prioritization," in *Proceedings of the 13th international conference on predictive models and data analytics in software engineering*, 2017, pp. 2–11.
[18] F. Palma, T. Abdou, A. Bener, J. Maidens, and S. Liu, "An improvement to test case failure prediction in the context of test case prioritization," in *Proceedings of the 14th international conference on predictive models and data analytics in software engineering*, 2018, pp. 80–89.
[19] J. Joo, S. Yoo, and M. Park, "Poster: Test case prioritization using error propagation probability," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2020, pp. 398–401.
[20] R. Mukherjee and K. S. Patnaik, "A survey on different approaches for software test case prioritization," *Journal of King Saud University-Computer and Information Sciences*, vol. 33, no. 9, pp. 1041–1054, 2021.
[21] S. Iqbal and I. Al-Azzoni, "Test case prioritization for model transformations," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 8, pp. 6324–6338, 2022.
[22] S. Eghbali, V. Kudva, G. Rothermel, and L. Tahvildari, "Supervised tie breaking in test case prioritization," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 242–243.
[23] R. Krishnamoorthi and S. S. A. Mary, "Factor oriented requirement coverage based system test case prioritization of new and regression test cases," *Information and Software Technology*, vol. 51, no. 4, pp. 799–808, 2009.
[24] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing junit test cases," *IEEE transactions on software engineering*, vol. 38, no. 6, pp. 1258–1275, 2012.
[25] W. Zhang, B. Wei, and H. Du, "Test case prioritization based on genetic algorithm and test-points coverage," in *Algorithms and Architectures for Parallel Processing: 14th International Conference, ICA3PP 2014, Dalian, China, August 24-27, 2014. Proceedings, Part I 14*. Springer, 2014, pp. 644–654.
[26] W. Fu, H. Yu, G. Fan, X. Ji, and X. Pei, "A regression test case prioritization algorithm based on program changes and method invocation relationship," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2017, pp. 169–178.
[27] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie, "To be optimal or not in test-case prioritization," *IEEE Transactions on Software Engineering*, vol. 42, no. 5, pp. 490–505, 2015.
[28] S. Wang, J. Nam, and L. Tan, "Qtep: Quality-aware test case prioritization," in *Proceedings of the 2017 11th Joint Meeting on foundations of software engineering*, 2017, pp. 523–534.
[29] M. Mahdieh, S.-H. Mirian-Hosseinabadi, K. Etemadi, A. Nosrati, and S. Jalali, "Incorporating fault-proneness estimations into coverage-based test case prioritization methods," *Information and Software Technology*, vol. 121, p. 106269, 2020.
[30] E. Engström, P. Runeson, and G. Wikstrand, "An empirical evaluation of regression testing based on fix-cache recommendations," in *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 2010, pp. 75–78.
[31] M. Laali, H. Liu, M. Hamilton, M. Spichkova, and H. W. Schmidt, "Test case prioritization using online fault detection information," in *Reliable Software Technologies–Ada-Europe 2016: 21st Ada-Europe International Conference on Reliable Software Technologies, Pisa, Italy, June 13-17, 2016, Proceedings 21*. Springer, 2016, pp. 78–93.
[32] S. Kim and J. Baik, "An effective fault aware test case prioritization by incorporating a fault localization technique," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2010, pp. 1–10.
[33] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 489–498.
[34] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Transactions on Software Engineering*, vol. 32, no. 9, pp. 733–752, 2006.
[35] S. Elbaum, A. 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*. IEEE, 2001, pp. 329–338.
[36] X. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," in *2007 IEEE International Conference on Software Maintenance*. IEEE, 2007, pp. 255–264.
[37] 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.
[38] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead, "Does bug prediction support human developers? findings from a google case study," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 372–381.
[39] J. Ratzinger, M. Pinzger, and H. Gall, "Eq-mine: Predicting short-term defects for software evolution," in *Fundamental Approaches to Software Engineering: 10th International Conference, FASE 2007, Held as Part of the Joint European Conferences, on Theory and Practice of Software, ETAPS 2007, Braga, Portugal, March 24-April 1, 2007. Proceedings 10*. Springer, 2007, pp. 12–26.
[40] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," *IEEE Transactions on software engineering*, vol. 26, no. 7, pp. 653–661, 2000.
[41] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

Yanan Zhu is currently pursuing the MS degree in the School of Computer and Information Technology, Beijing Jiaotong University. She received her BS degree in Software Engineering from Zhengzhou University in 2019. Her current research interests are mainly in software testing.

Feng Liu is a professor in the School of Computer and Information Technology at Beijing Jiaotong University. He received his BS degree in computer software and MS degree in computer application from Beijing Jiaotong University in 1983 and 1988, respectively, and his PhD degree in economics from Renmin University of China in 1997. His current research interests include Software Testing and Cloud Testing.