An Improved Method for Test Case Prioritization in Continuous Integration based on Reinforcement Learning

Yanan Han*,[1], Gang Chen[1], Bin Han[2]

[1]School of Information Management and Engineering, Shanghai University of Finance and Economics, No. 777 Guoding Road, Yangpu District, Shanghai, 200433
[2]School of Materials Science and Engineering, China University of Petroleum (East China), No. 66 Changjiang West Road, Huangdao District, Qingdao City, Shandong Province, 266580

*Correspongding uthor.Email:hanyanan@163.sufe.edu.cn

Abstract. The iterative update of software leads to frequent continuous integration, so the testing in the continuous integration environment should also be fast and accurate. Reinforcement learning is often used in the research of continuous integration testing because of its sequential strategy and good robustness. Some existing methods use reinforcement learning to solve test case prioritization problem, which provides a good idea, but the experimental defect detection rates are relatively low. Therefore, based on the existing reinforcement learning framework, this article proposes a reward mechanism to provide additional rewards for newly emerging test cases in each integration cycle. Through experiments on three industrial datasets, it has been proven that this mechanism improves the defect detection rate, the recall rate of failed test cases, and the efficiency of test feedback in the testing process.

Keywords: test case prioritization, continuous integration, reinforcement learning, additional rewards for new test cases

1 Introduction

With the development of computer and software, Devops has become the main mode of current software development, and one of the key links is continuous integration (CI). Continuous integration refers to the process that developers continuously integrate new code into existing systems, and compile, test, and release it [1]. For large systems, continuous integration is usually performed once or more a day. Therefore, testing in the continuous integration environment is carried out in the case of very limited time resources, which requires the testing framework to be as fast and efficient as possible [2]. The strict requirement for timeliness is the biggest difference between the continuous integration test environment and the traditional software test environment, which also means that the traditional test optimization methods may not be directly migrated to the continuous integration environment [3].

A. Rauf et al. (eds.), *Proceedings of the 3rd International Conference on Management Science and Software Engineering (ICMSSE 2023)*, Atlantis Highlights in Engineering 20,
https://doi.org/10.2991/978-94-6463-262-0_99

In each cycle of continuous integration, a test case set will be generated according to the current system, and a part of the test case set will be run to detect whether the system has defects. There are many ways to conduct test optimization, such as Test Case Selection (TCS) [4], Test Case Prioritization (TCP) [5], Test Suite Minimization (TSM) [6], etc. Test case selection selects a part of the test cases according to some standards, and test case prioritization sorts them according to the properties of the test cases so that some cases can be executed first, while test suite minimization remove some redundant or expired test cases. Because the time intervals between continuous integration cycles are short, it is unrealistic to run all the test cases in the test case set [7]. We will focus on test case prioritization so that those important test cases can be tested preferentially and the testing time can be fully utilized. Test case prioritization is often related to code. Many methods consider code coverage [8], defect detection rate [9], requirement correlation [10], test case similarity [11], test history [12] and other information when ranking test cases [13]. In recent years, machine-learning-based and search-based test case sorting methods have become mainstream [14], however, the data used for training in most methods still needs to be extracted from system code. For continuous integration testing, if the source code needs to be analyzed every cycle, it is difficult to form a lightweight framework even with incremental analysis [15]. Therefore, in order to achieve intelligent data analysis suitable for continuous integration testing, the data used to train should meet the intuitive and accessible nature, such as historical execution information, and a sorting model that can simulate the continuous integration environment and process test case information is needed.

There has been a lot of research on test case prioritization in continuous integration environment. Lima et al. [16] improves COLEMAN, a learning-based sorting method, and puts forward two strategies to deal with variables, which makes COLEMAN practicable for highly-configurable software in continuous integration; Rosenbauer et al. [17] proposed and optimized a test case ranking model based on Learning Classification System (LCS), and demonstrated through experiments that it performs better than network-based models; Xiao et al. [18] proposed a sorting method based on Long Short-Term Memory Network (LSTM) and applied it to embedded software, which improved its fault detection rate in the continuous integration environment; Ali et al. [19] proposed a clustering method that clusters and sorts test cases based on their historical failure frequency and coverage criteria, achieving a defect detection rate of over 90%.

The test case prioritization in continuous integration environment is essentially a sequential decision-making process [20]. Selecting the test cases that are most likely to discover defects in each integration cycle and executing them preferentially coincide with the idea of reinforcement learning. Reinforcement learning [21] is a branch of machine learning that involves the continuous interaction and learning between the agent and the environment. Several important elements are state, action, and reward. The agent takes an action based on the current state and feedback from the previous environment, changes the state of the environment, and receives reward of the action from the environment. Therefore, reward will gradually guide the agent to make more correct choices. Because test cases that previously detected defects are highly likely to still detect defects in subsequent cycles [22], the mechanism of reinforcement learning

to provide reward feedback to the agent can actually help the agent to "remember" which test cases are more likely to detect defects.

The RETECS framework proposed by Spieker et al. [23] in 2017 is the first framework that applies reinforcement learning to test case prioritization in continuous integration environment. The process of this framework is shown in Fig. 1. Reinforcement learning adds a feedback loop to the original continuous integration process, enabling the agent to continuously improve the sorting strategy and the sorting outcome according to the feedbacks. There are three reward mechanisms proposed simultaneously with this framework: (1) FC (Failure Count) reward, which rewards all test cases in the test case set, with the reward value being the number of failed test cases in the test sequence; (2) TF (Test Case Failure) reward, only rewards failed test cases in the test sequence, with the reward value of 1; (3) time-rank reward, is a reward for test cases in the test sequence, with reward value determined based on the test result of the test case and its position in the test sequence. According to Spieker et al.'s experiment, the TF reward performs the best.

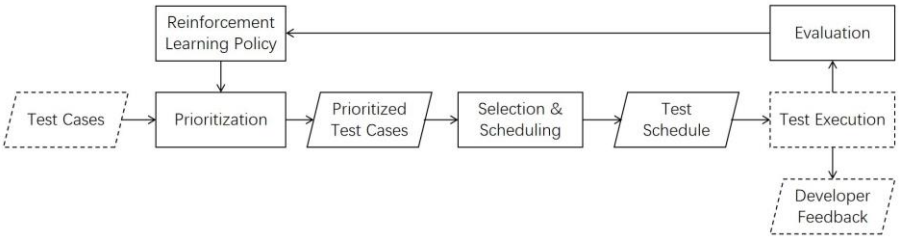

Fig. 1. the Process of RETECS

Afterwards, the framework also underwent many improvements, mainly in terms of the reward mechanism. He et al. [24] proposed two new reward functions: (1) HFC (Historical Failure Count) reward, considers the historical failure count of test cases; (2) APHF (Average Percentage of Historical Failure) reward, takes into account the execution history of test cases, resulting in higher reward values for test cases with more recent failures. Thereafter, Wu et al. [25] proposed a reward function based on time windows, which only uses historical information from the last few cycles. At the same time, Yang et al. [26] summarized the reward object selection strategy, in addition to total and partial rewards, they proposed a fuzzy reward strategy, which determines whether a test case is rewarded based on its historical failure rate. In addition, in the design of reward functions, there are studies that consider the testing frequency of test cases [27] and failure location [28]; In terms of the selection of reward objects, there are also studies that determine whether to reward based on testing frequency [27], as well as the similarity between test cases [29]. Although the above researches have made some advancements compared to the initial version, most of these methods only consider the historical execution information of test cases. Different reward mechanisms are essentially the interception and weighted calculation of historical execution sequences, and when allocating rewards, test cases are mostly divided into failed test cases and passed test cases. Thus, improvements can be made in the following directions, such as classifying test cases into the new ones and the old ones and assigning

different reward values to them [30]; or utilize additional requirement-based information to supplement the lack of historical execution information [31].

2 The Improved Method with Additional Rewards for New Test Cases

2.1 The Structure of Reinforcement Learning for Test Case Prioritization

The test case prioritization model based on reinforcement learning is shown in Fig. 2. The Environment of reinforcement learning is the continuous integration cycle, and the Agent of reinforcement learning is the sorting strategy. The State corresponds to metadata of test cases, such as execution time, execution history, etc.; the Action corresponds to assigning priority to test cases; the Reward corresponds to the feedback on test results. Therefore, the learning process of a certain integration cycle is as follows: the integration cycle first hands over the data of test cases to the Agent, and the Agent assigns priority to these test cases based on the information of these cases and the "knowledge" previously learned from the Environment. The test cases are sorted based on priority and submitted to the Environment for testing. Then, the Environment rewards some test cases and feeds back to the Agent according to the test results. Among them, which test cases are rewarded and how much reward they should obtain are the concerns of reward mechanism. The reward mechanism here mainly includes the design of reward functions and the selection of reward objects.

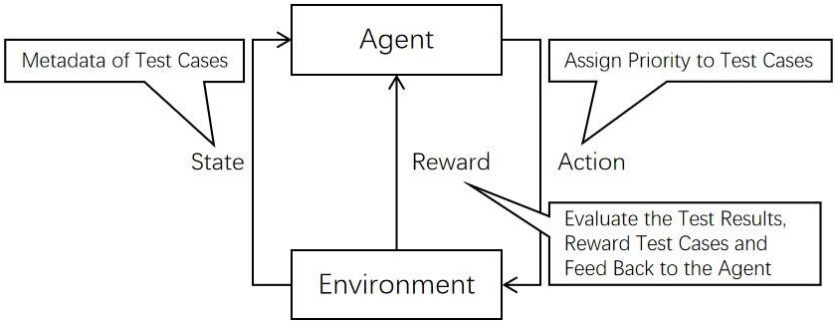

Fig. 2. the Test Case Prioritization Model based on Reinforcement Learning

When rewarding test cases, most existing reward mechanisms consider the historical execution information of test cases, but for newly emerging test cases in a cycle, their historical information is empty and cannot be rewarded. However, in the continuous integration process, the new test cases in each cycle correspond to the changed code. The regression testing process not only needs to ensure the normal operation of the original functions, but also needs to detect whether there are faults in the new functions [32]. Therefore, it is necessary to give additional rewards to these new test cases. In previous studies, the two reward functions that performed well are TF reward and APHF reward. Therefore, this article proposes two reward mechanisms based on these

two reward mechanisms: TF_NA (TF Reward plus New Test Case Additional Reward), and APHF_NA (APHF plus New Test Case Additional Reward).

2.2 TF Reward plus New Test Case Additional Reward

The test case sets in continuous integration are defined as follows: the test case set submitted in the i-th integration cycle is TS_i^{total}; the failure cases in this set form a subset $TS_i^{total,fail}$; the test sequence for the i-th integration cycle is TS_i, which is obtained by sorting and selecting from TS_i^{total}; the failure cases in this test sequence form a subset TS_i^{fail}; the set of test cases newly appearing in the i-th integration cycle is TS_i^{new}.

The TF reward only applies to the failed test cases in the test sequence with a reward value of 1, which does not take into account the importance of new test cases, but new test cases need to be given reward values to reflect their testing priority. Therefore, we combine the TF reward and additional reward for new test cases, proposing the TF_NA reward. The definition of this reward mechanism is as follows:

$$reward_i^{TF\_NA}(t)=\begin{cases}2, & t\in TS_i^{fail}\wedge t\in TS_i^{new}\\ 1, & t\in TS_i^{fail}\wedge t\notin TS_i^{new}\\ 1, & t\notin TS_i^{fail}\wedge t\in TS_i^{new}\\ 0, & \text{others}\end{cases} \quad (1)$$

According to the formulation, test cases will receive rewards with a value of 1 based on whether they are failure cases in the test sequence or new cases, respectively. Here the additional reward value for new test cases is set to 1, which means that failed test cases and new test cases have equal priority, because these two kinds of test cases correspond to the error prone points of original codes and the changed parts in the program, respectively. Both of them are very important and need to be prioritized during testing.

2.3 APHF plus New Test Case Additional Reward

APHF is a highly effective reward function proposed after TF, and many studies based on this framework use this reward function as a benchmark for comparison [25-29]. Combining this reward with the new test case additional reward, a new reward mechanism APHF_NA is proposed. The definition of this reward mechanism is as follows:

$$reward_i^{APHF\_NA}(t)=\begin{cases}APHF_i(t), & t\in TS_i^{total}\wedge m\neq 0\\ NAPFD_i, & t\in TS_i^{new}\\ 0, & \text{others}\end{cases} \quad (2)$$

$$\text{with} \quad APHF = 1 - \frac{\sum_{j=1}^{m} R_j}{m \times n} + \frac{1}{2 \times n} \quad (3)$$

In the calculation formula of APHF [24], n represents the number of historical executions of the test case, m represents the number of historical failures of the test case, and R_j represents the reverse order of the j-th failure of the test case in the execution history. According to this formula, only test cases that have failed in history can calculate the APHF value. Therefore, there is no overlap between test cases that receive APHF rewards and those that receive new test case additional rewards. The APHF reward in the APHF_NA reward mechanism is a total reward, which rewards all test cases that can calculate APHF rewards. NAPFD [33] is one of the evaluation indicators for test cases prioritization, whose formula and definition will be given in section 3.2. Reinforcement learning evaluate the sorting outcome after executing the test sequence, and NAPFD is a numerical expression of the evaluation, which makes it natural to set NAPFD as the additional reward value for new test cases. Of course, other evaluation values or constants can also be chosen as additional rewards for new test cases here, but using NAPFD is the most intuitive and effective. In addition, since the range of APHF is between 0 and 1, and the range of NAPFD is also between 0 and 1, reinforcement learning does not have to worry about normalization during training.

3 Experimental Analysis

3.1 Datasets

This paper uses three industrial datasets: ABB Paint Control, ABB IOF/ROL, and GSDTSR. The first two datasets are from ABB Robotics Norway, and GSDTSR is the test suite data shared by Google. Spieker et al. [23] and He et al. [24] used these three datasets for experiments, and other studies on continuous integration test optimization also used the above datasets. In order to facilitate the comparison of experimental results, this article also uses these three datasets, and the basic information of the datasets is shown in Table 1. It can be seen that the number of integration cycles of all three datasets have exceeded 300, but the GSDTSR in terms of test execution results is much larger than the first two datasets. In addition, the failure rate of the first two datasets is relatively high, which means that once the test case is executed, there is a high probability of failure; The ABB Paint Control and GSDTSR datasets have a higher testing frequency, indicating a higher probability of test cases being executed.

Table 1. Information of Datasets

Datasets	Test Cases	CI Cycles	Results	Failure Rate	Frequency
Paint Control	114	352	25594	19.36%	0.82
IOF/ROL	2086	320	32260	28.43%	0.05
GSDTSR	5555	336	1260617	0.25%	0.68

3.2 Evaluation Metrics

The evaluation metric used in this article is mainly NAPFD [33] (Normalized Average Percentage of Faults Detected), which is the standardized average defect detection percentage. This is improved from APFD [34] (Average Percentage of Faults Detected). This indicator is concerned about the number and ranking of failed test cases in the test sequence. The more failed test cases are found and the higher the position, the closer the value of this indicator is to 1, and the better the sorting effect of the test sequence. The calculation formula for this indicator is as follows:

$$NAPFD_i = p - \frac{\sum_{t \in TS_i^{fail}} rank(t)}{\left|TS_i^{fail}\right| \times \left|TS_i\right|} + \frac{p}{2 \times \left|TS_i\right|} \left(\text{with } p = \frac{\left|TS_i^{fail}\right|}{\left|TS_i^{total,fail}\right|} \right) \quad (4)$$

In the calculation formula of NAPFD, $rank(t)$ is the position of the failed test case in the test sequence, and the meanings of other elements have been discussed before. Due to the inability to test all test cases in continuous integration testing, NAPFD has standardized APFD. If the test sequence can detect all defects in the test case set, then $p = 1$, and NAPFD is equal to APFD.

In addition, this article also considers two auxiliary indicators: (1) TTF, which represents the position of the first failed test case in the test sequence. The smaller the indicator, the earlier the defect is discovered in the test sequence, allowing developers to receive feedback earlier and make adjustments to the program. (2) recall, also known as recall rate, is the proportion of defects found in the test sequence to defects in the test case set. The higher the value, the more complete the defects found in the test sequence.

3.3 Experimental Setup

To verify the effectiveness of additional rewards for new test cases in improving the effect of test case prioritization in continuous integration environment, we conduct experiments on three datasets with four reward mechanisms: APHF, APHF_NA, TF, TF_NA, and propose the following three research questions:

RQ1: Can additional rewards for new test cases improve the effectiveness of test case prioritization?

RQ2: Is there a significant difference in the improvements of sorting effect when TF and APHF combine with additional rewards for new test cases respectively?

RQ3: Will adding additional rewards for new test cases bring too much time burden to the reinforcement-learning-based test case prioritization framework?

Among them, RQ1 is concerned about the actual effect of additional rewards for new test cases, which is the key research issue of this article. By comparing NAPFD, TTF, recall of APHF_NA and APHF, TF and TF_NA, it is clear to illustrate whether the reward mechanism with additional rewards for new test cases can improve the sorting effect in continuous integration testing. RQ2 is concerned about the difference in the effect improvement generated by the combination of additional rewards for new test cases and different reward functions. Although APHF and TF are combined with additional rewards for new test cases, APHF_NA and TF_NA have different additional reward value, and the reward objects are not entirely the same, which may lead to differences in the final results. RQ3 is concerned about whether adding additional rewards for new test cases will cause additional time burden, because continuous integration testing requires a lightweight test optimization model. If the new reward mechanism brings too much time consumption, the model will not be applicable to the continuous integration environment. This research question is to confirm the feasibility of the new reward mechanism.

During the experiment, due to the inability to obtain the available testing time for each integration cycle in advance, we used 50% of the total testing time for all test cases in that cycle as the available testing time [23]. After sorting the test cases, execute them in order until the available testing time is exhausted. In addition, for some cycles where there is no failed test case, it is meaningless to consider the sorting effect. If it is also included in the calculation of evaluation indicators, the value of them will appear to be artificially high. Therefore, the data for the subsequent experimental results in this article is based on the data of the cycles in which failed test cases occurred. To reduce the interference of randomness during the experimental process, this article conducted 30 repeated experiments, and the subsequent result data was the average of 30 repeated experiments.

3.4 Results and Analysis

Fig. 3 shows the NAPFD values of four reward mechanisms on three datasets, with the abscissa representing the continuous integration cycle, the black line representing the trend fitting of NAPFD values, and (a) representing APHF reward, (b) representing APHF_NA reward, (c) represents TF reward, (d) represents TF_NA rewards. Table 2 shows the average NAPFD values for all failure cycles on three datasets with different reward mechanisms, and the bold data represents the better result between the original reward and those adding additional rewards for new test cases.

Analysis of RQ1. According to Fig. 3 and Table 2, adding additional rewards for new test cases can effectively improve the NAPFD value of test case prioritization. Both APHF_NA and TF_NA has a positive slope of their trend fitting lines, indicating that reward mechanism with additional rewards for new test cases can continuously optimize the sorting model and improve the sorting effect under the guidance of feedback from reinforcement learning. In order to further compare the sorting effects of the reward mechanisms with and without additional rewards for new test cases, we need to

analyze their TTF and recall values. Table 3 shows the average TTF values of all failure cycles using different reward mechanisms on the datasets, and Table 4 shows the average recall values of all failure cycles using different reward mechanisms on the datasets. The bold data represents the better value between the original reward and those adding additional rewards for new test cases.

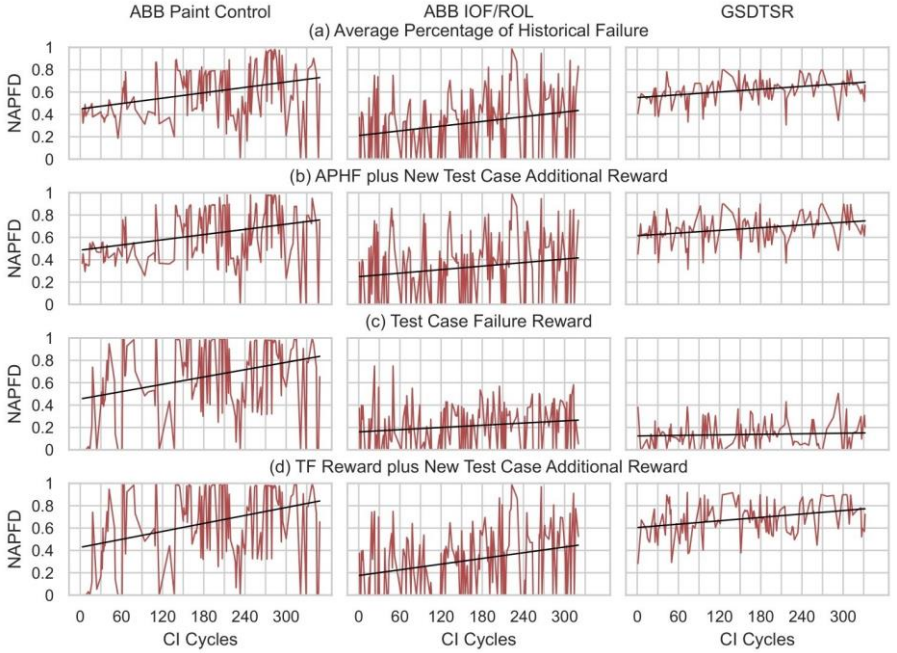

Fig. 3. NAPFD on three datasets: (a) APHF, (b) APHF_NA, (c) TF, (d) TF_NA

Table 2. Average NAPFD on three Datasets with four Reward Mechanisms

Reward Mechanisms	APHF	APHF_NA	TF	TF_NA
ABB Paint Control	0.6017	**0.6330**	**0.6624**	0.6535
ABB IOF/ROL	0.3203	**0.3305**	0.2118	**0.3079**
GSDTSR	0.6176	**0.6792**	0.1372	**0.6859**

Table 3. Average TTF on three Datasets with four Reward Mechanisms

Reward Mechanisms	APHF	APHF_NA	TF	TF_NA
ABB Paint Control	**3.79**	3.93	**3.88**	4.00
ABB IOF/ROL	2.56	**2.05**	5.48	**1.68**
GSDTSR	77.94	**8.63**	520.55	**46.19**

Table 4. Average Recall on three Datasets with four Reward Mechanisms

Reward Mechanisms	APHF	APHF_NA	TF	TF_NA
ABB Paint Control	0.6893	**0.7250**	**0.7646**	0.7550
ABB IOF/ROL	0.4062	**0.4191**	0.2852	**0.3861**
GSDTSR	0.6590	**0.7263**	0.1838	**0.7323**

In terms of NAPFD, it can be seen that the average NAPFD values of APHF_NA on all three datasets are slightly higher than APHF. While NAPFD of TF_NA is somewhat lower than that of TF on the ABB Paint Control dataset, there is a significant difference between them on the remaining two datasets, and even on the GSDTSR dataset, the average NAPFD of TF_NA is 0.5487 higher than TF. This shows that in the sorting model based on reinforcement learning, using reward mechanisms that assign additional rewards to new test cases can indeed improve the sorting effect. In terms of TTF values, except for the ABB Paint Control dataset where the original reward can obtain smaller TTF values, the remaining two datasets both witness that mechanisms with additional rewards for new test cases obtain smaller TTF values, especially on the GSDTSR dataset where the location of the first failed case discovered by APHF_NA is 69.31 ahead of APHF, and that discovered by TF_NA was 474.36 earlier than TF, indicating that adding additional rewards for new test cases increased the priority of new test cases and enabled earlier defect detection. In terms of recall, TF can be slightly larger than TF_NA on the ABB Paint Control dataset, while in other scenarios, the reward mechanisms adding additional rewards for new test cases are used to obtain higher recall values. The distribution characteristics of this metric are similar to that of NAPFD, indicating that the additional rewards for new test cases cause the model to discover more failed test cases in each cycle. This is due to the priority given to new test cases, which causes the failed new test cases to be executed preferentially.

From a perspective of datasets, it is shown that adding additional rewards for new test cases is not significantly improved the sorting effect on the ABB Paint Control dataset, and the average NAPFD values of each reward mechanism are between 0.6 and 0.7. This may be due to the small number of test cases and high testing frequency in this dataset. On the one hand, this leads to fewer new test cases and less additional rewards can be obtained. On the other hand, due to the high frequency of testing, the probability of test cases being selected into the testing sequence is high, and the additional rewards for new test cases become insignificant. On the ABB IOF/ROL dataset, the performance of various reward mechanisms is not ideal, and even the best performing reward APHF_NA has the average NAPFD value of only 0.3305 and the recall value of only 0.4191, which means that only 40% of defects are found on average during the integration cycle. The reason for this may be that the reward strategies used in the text, such as TF, APHF, and additional rewards for new test cases, do not grasp the characteristics of the dataset well, resulting in the model not assigning high priority values to those truly important test cases. More suitable reward mechanisms for this dataset can be analyzed in future research. On the GSDTSR dataset, the performance of additional rewards for new test cases is particularly outstanding, especially TF_NA significantly improves sorting performance compared to TF. The dataset has a low failure rate, therefore, the TF and APHF reward values that test cases can obtain are

relatively low. At this point, the additional reward for new test cases becomes the main rewards for test cases, indicating that the reward mechanism with additional rewards for new test cases can better adapt to this integration testing environment.

Analysis of RQ2. In the analysis to RQ1, it can already be observed that TF_NA has achieved more improvement to TF than that of APHF_NA to APHF. Especially on the ABB IOF/ROL and GSDTSR datasets, the NAPFD values of TF_NA are 0.0961 and 0.5487 higher than TF respectively, while the NAPFD values of APHF_NA are 0.0102 and 0.0616 higher than APHF respectively. The reasons for the differences are analyzed as follows: firstly, the TF reward function itself may not be suitable for these two datasets, as the performance of the TF reward on the ABB Paint Control dataset is not significantly different from other reward functions. Secondly, the additional rewards for new test cases are combined with TF and APHF in different ways, in TF_NA there are some test cases that can receive both TF rewards and additional rewards for new test cases; but in APHF_NA, there is no overlap between test cases that receive APHF rewards and additional rewards for new test cases. Test cases either receive only one of these two rewards or do not receive any rewards. Thirdly, the reward values assigned to test cases are different, TF_NA assigns the same priority to failure test cases and new cases, believing that they are equally important; while in APHF_NA, test cases that has failed receive the priority value of their APHF value, and new test cases receive the priority value of the NAPFD value of sorting result. These two values are not equal, and the reward values for test cases that receive APHF rewards is also different, which is related to the execution history of the test cases. However, the reward values for test cases that receive new test case additional reward is exactly the same. The above analysis explains the reason why the additional rewards for new test cases differ in improving sorting effects when different reward functions are combined, and also inspires us to further study the setting and combination methods of reward values for new test cases in the future.

Analysis of RQ3. Table 5 shows the average calculation time per cycle, in seconds, with different reward mechanisms on the datasets. It can be seen that in each cycle of continuous integration, the time used to calculate the priority of test cases, sort them, evaluate and feedback the results is actually very short, which makes most of the test time available for the execution of test cases, rather than for priority analysis. It is very friendly to the continuous integration environment. The reward mechanisms that add additional rewards for new test cases has a slightly higher computational time than the original mechanisms, which is the time they take to assign additional rewards to new test cases, but it is completely within an acceptable time growth range. Even in the GSDTSR dataset with the most test cases, the maximum time growth is only 0.0581 seconds. As a result, the additional rewards for new test cases not only improve the sorting performance of the reinforcement learning framework, but also do not bring too much time burden.

Table 5. Average Calculation Time per Cycle (s)

Reward Mechanisms	APHF	APHF_NA	TF	TF_NA
ABB Paint Control	0.0110	0.0110	0.0119	0.0133
ABB IOF/ROL	0.0127	0.0126	0.0161	0.0194
GSDTSR	0.6506	0.6710	1.4889	1.5470

4 Conclusion

Based on the reinforcement learning framework for continuous integration test case prioritization, this paper improves the reward mechanism of reinforcement learning, and proposes two reward mechanism TF_NA and APHF_NA that gives additional rewards to new test cases, and conducted experiments on industrial datasets. The experiments show that: (1) Additional rewards for new test cases can effectively improve the optimization effect of continuous integration test case set, which makes the sorting results have a certain degree of improvement in defect detection rate, recall rate of failed test cases, and efficiency of test feedback; (2) The additional rewards for new test cases have different effects when combined with different reward functions, specifically manifested in that TF_NA can significantly improve the sorting performance of TF, while APHF_NA's improvement to APHF is relatively small; (3) Additional rewards for new test cases will not result in excessive time consumption, and the model using TF_NA or APHF_NA reward mechanism is still a lightweight reinforcement learning model.

The method proposed in this paper conforms to the idea of regression testing and the characteristics of continuous integration testing. The experimental results are clear, which can demonstrate the importance of giving high priority to new test cases in regression testing. However, there is still room for improvement in this method, such as considering more kinds of reward functions and combination methods when combining with additional rewards for new test cases. As for the additional reward value, although different attempts have been made in experiments, it is still not systematic enough and can be analyzed and studied more comprehensively.

References

1. F. Cannizzo, R. Clutton and R. Ramesh, "Pushing the Boundaries of Testing and Continuous Integration," Agile 2008 Conference, Toronto, ON, Canada, 2008, pp. 501-505, doi: 10.1109/Agile.2008.31.
2. Pinto, G, Castor, F, Bonifacio, R, Rebouças, M. Work practices and challenges in continuous integration: A survey with Travis CI users. Softw Pract Exper. 2018; 48: 2223– 2236. https://doi.org/10.1002/spe.2637
3. D. Mondal, H. Hemmati and S. Durocher, "Exploring Test Suite Diversification and Code Coverage in Multi-Objective Test Case Selection," 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), Graz, Austria, 2015, pp. 1-10, doi: 10.1109/ICST.2015.7102588.

4. D. Mondal, H. Hemmati and S. Durocher, "Exploring Test Suite Diversification and Code Coverage in Multi-Objective Test Case Selection," 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), Graz, Austria, 2015, pp. 1-10, doi: 10.1109/ICST.2015.7102588.
5. Hao, D., Zhang, L. & Mei, H. Test-case prioritization: achievements and challenges. Front. Comput. Sci. 10, 769–777 (2016). https://doi.org/10.1007/s11704-016-6112-3
6. Mohapatra, S.K., Mishra, A.K. & Prasad, S. Intelligent Local Search for Test Case Minimization. J. Inst. Eng. India Ser. B 101, 585–595 (2020). https://doi.org/10.1007/s40031-020-00480-7
7. G. Rothermel, R. H. Untch, Chengyun Chu and M. J. Harrold, "Prioritizing test cases for regression testing," in IEEE Transactions on Software Engineering, vol. 27, no. 10, pp. 929-948, Oct. 2001, doi: 10.1109/32.962562.
8. Dusica Marijan. 2015. Multi-perspective Regression Test Prioritization for Time-Constrained Environments. In Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS '15). IEEE Computer Society, USA, 157–162. https://doi.org/10.1109/QRS.2015.31
9. Nayak, S., Kumar, C. & Tripathi, S. Enhancing Efficiency of the Test Case Prioritization Technique by Improving the Rate of Fault Detection. Arab J Sci Eng 42, 3307–3323 (2017). https://doi.org/10.1007/s13369-017-2466-6
10. Vescan, Andreea & Serban, Camelia & Chisalita-Cretu, Camelia & Diosan, Laura. (2017). Requirement dependencies-based formal approach for test case prioritization in regression testing. 181-188. 10.1109/ICCP.2017.8117002.
11. Noor T B, Hemmati H. A similarity-based approach for test case prioritization using historical failure data[C]//2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2015: 58-68.
12. Hema Srikanth, Mikaela Cashman, Myra B. Cohen, Test case prioritization of build acceptance tests for an enterprise cloud application: An industrial case study, Journal of Systems and Software, Volume 119, 2016, Pages 122-135, ISSN 0164-1212, https://doi.org/10.1016/j.jss.2016.06.017.
13. Yingling Li, Qing Wang. Review of test case set optimization in continuous integration [J]. Journal of Software, 2018, 29(10):3021-3050.DOI:10.13328/j.cnki.jos.005613.
14. Rahmani A, Ahmad S, Jalil I E A, et al. A Systematic Literature Review on Regression Test Case Prioritization[J]. International Journal of Advanced Computer Science and Applications, 2021, 12(9).
15. Y. Zhu, E. Shihab and P. C. Rigby, "Test Re-Prioritization in Continuous Testing Environments," 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 2018, pp. 69-79, doi: 10.1109/ICSME.2018.00016.
16. Prado Lima, J.A., Mendonça, W.D.F., Vergilio, S.R. et al. Cost-effective learning-based strategies for test case prioritization in continuous integration of highly-configurable software. Empir Software Eng 27, 133 (2022). https://doi.org/10.1007/s10664-021-10093-3
17. Rosenbauer, L., Pätzel, D., Stein, A. et al. A Learning Classifier System for Automated Test Case Prioritization and Selection. SN COMPUT. SCI. 3, 373 (2022). https://doi.org/10.1007/s42979-022-01255-1
18. Xiao, L., Miao, H., Shi, T. et al. LSTM-based deep learning for spatial–temporal software testing. Distrib Parallel Databases 38, 687–712 (2020). https://doi.org/10.1007/s10619-020-07291-1
19. Ali, S., Hafeez, Y., Hussain, S. et al. Enhanced regression testing technique for agile software development and continuous integration strategies. Software Qual J 28, 397–423 (2020). https://doi.org/10.1007/s11219-019-09463-4

20. A. G. Barto, R. S. Sutton, and C. J. C. H. Watkins. 1990. Sequential decision problems and neural networks. Advances in neural information processing systems 2. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 686–693.
21. Veanes, M., Roy, P., Campbell, C. (2006). Online Testing with Reinforcement Learning. In: Havelund, K., Núñez, M., Roşu, G., Wolff, B. (eds) Formal Approaches to Software Testing and Runtime Verification. FATES RV 2006 2006. Lecture Notes in Computer Science, vol 4262. Springer, Berlin, Heidelberg. https://doi.org/10.1007/11940197_16
22. Cave, P. (2000), The error of excessive proximity preference – a modest proposal for understanding holism. Nursing Philosophy, 1: 20-25. https://doi.org/10.1046/j.1466-769x.2000.00003.x
23. Helge Spieker, Arnaud Gotlieb, Dusica Marijan, and Morten Mossige. 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017). Association for Computing Machinery, New York, NY, USA, 12–22. https://doi.org/10.1145/3092703.3092709
24. Liuliu He, Yang Yang, Zheng Li, et al. Reinforcement learning reward mechanism for continuous integration testing optimization [J]. Journal of Software, 2019, 30(05):1438-1449.DOI:10.13328/j.cnki.jos.005714.
25. Zhaolin Wu, Yang Yang, Zheng Li, and Ruilian Zhao. 2019. A Time Window based Reinforcement Learning Reward for Test Case Prioritization in Continuous Integration. In Proceedings of the 11th Asia-Pacific Symposium on Internetware (Internetware '19). Association for Computing Machinery, New York, NY, USA, Article 4, 1–6. https://doi.org/10.1145/3361242.3361258
26. Yang Yang, Zheng Li, Liuliu He, Ruilian Zhao, A systematic study of reward for reinforcement learning based continuous integration testing, Journal of Systems and Software, Volume 170, 2020, 110787, ISSN 0164-1212, https://doi.org/10.1016/j.jss.2020.110787.
27. Ying Shang, Qianyu Li, Yang Yang, and Zheng Li. 2020. Occurrence Frequency and All Historical Failure Information Based Method for TCP in CI. In Proceedings of the International Conference on Software and System Processes (ICSSP '20). Association for Computing Machinery, New York, NY, USA, 105–114. https://doi.org/10.1145/3379177.3388903
28. G. Li, Y. Yang, Z. Wu, T. Cao, Y. Liu and Z. Li, "Weighted Reward for Reinforcement Learning based Test Case Prioritization in Continuous Integration Testing," 2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC), Madrid, Spain, 2021, pp. 980-985, doi: 10.1109/COMPSAC51774.2021.00132.
29. Yang Yang, Chaoyue Pan, Tiange Cao, et al. Reinforcement learning reward strategy of CITCP based on similarity [J]. Computer Systems & Applications, 2022, 31(02):325-334.DOI:10.15888/j.cnki.csa.008300.
30. Li Zhang, Lili Dai, Lan Du. Regression testing case prioritization in agile development mode [J]. Microelectronics & Computer, 2020, 37(12):48-52.DOI:10.19304/j.cnki.issn1000-7180.2020.12.010.
31. R. Chen, Z. Xiao, L. Xiao and Z. Li, "Regression Testing Prioritization Technique Based on Historical Execution Information," 2022 International Conference on Machine Learning, Cloud Computing and Intelligent Mining (MLCCIM), Xiamen, China, 2022, pp. 276-281, doi: 10.1109/MLCCIM55934.2022.00054.
32. Parsons, D., Susnjak, T. & Lange, M. Influences on regression testing strategies in agile software development environments. Software Qual J 22, 717–739 (2014). https://doi.org/10.1007/s11219-013-9225-z

33. D. Marijan, A. Gotlieb and S. Sen, "Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study," 2013 IEEE International Conference on Software Maintenance, Eindhoven, Netherlands, 2013, pp. 540-543, doi: 10.1109/ICSM.2013.91.
34. Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). Association for Computing Machinery, New York, NY, USA, 235–245. https://doi.org/10.1145/2635868.2635910