Focus on New Test Cases in Continuous Integration Testing based on Reinforcement Learning

Fanliang Chen[1], Zheng Li[1,*], Ying Shang[1,*], and Yang Yang[2]
[1]College of Information Science and Technology, Beijing University of Chemical Technology, Beijing, China
[2]School of Information Science and Engineering, Zhejiang Sci-Tech University, Hangzhou, Zhejiang, China
2021200836@mail.buct.edu.cn, lizheng@mail.buct.edu.cn, shangy@mail.buct.edu.cn, yangyang0070@zstu.edu.cn
*corresponding author

2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS) | 978-1-6654-7704-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/QRS57517.2022.00088

***Abstract*—In software regression testing, newly added test cases are more likely to fail, and therefore, should be prioritized for execution. In software regression testing for continuous integration, reinforcement learning-based approaches are promising and the RETECS (Reinforced Test Case Prioritization and Selection) framework is a successful application case. RETECS uses an agent composed of a neural network to predict the priority of test cases, and the agent needs to learn from historical information to make improvements. However, the newly added test cases have no historical execution information, thus using RETECS to predict their priority is more like 'random'. In this paper, we focus on new test cases for continuous integration testing, and on the basis of the RETECS framework, we first propose a priority assignment method for new test cases to ensure that they can be executed first. Secondly, continuous integration is a fast iterative integration method where new test cases have strong fault detection capability within the latest periods. Therefore, we further propose an additional reward method for new test cases. Finally, based on the full lifecycle management, the 'new' additional rewards need to be terminated within a certain period, and this paper implements an empirical study. We conducted 30 iterations of the experiment on 12 datasets and our best results were 19.24%, 10.67%, and 34.05 positions better compared to the best parameter combination in RETECS for the NAPFD (Normalized Average Percentage of Faults Detected), RECALL and TTF (Test to Fail) metrics, respectively.**

Keywords*—*Continuous integration; new test case; test case prioritization; reinforcement learning; reward function; regression testing

I. Introduction

Software products are updating faster and faster. The high-speed iteration of the software is closely related to teamwork and the joint development of software companies. When multiple programmers develop a software product together, they need to integrate the codes to integrate the product functions, which is a specific practice of continuous integration (CI) [1]. To ensure the quality of products, it is usually to conduct regression testing [2] during integration. However, the highly frequent CI imposes time constraints and resource constraints, which in turn demand faster fault detection. Traditional regression testing can hardly meet this requirement and several test case optimization methods have been proposed successively with the aim of quickly detecting failures in regression testing.

The test case optimization methods in software testing can be roughly divided into test case minimization (TCM), test case selection (TCS) and test case prioritization (TCP) [3]. TCM refers to identifying and eliminating outdated or redundant test cases [4]. TCS refers to selecting test cases that are more likely to make failures for testing [5]. TCP is to give a priority to every test case and then use the priority to determine the execution order of test cases [6]. TCP technology is the most commonly used in recent years, and the difficulties lie in what kind of priority should give. The main solution to solving this problem is extracting information from different dimensions of test cases, such as the coverage information of test cases and the historical execution information of test cases [7]. These features are then analyzed using heuristics or machine learning methods as the foundation for priority assignment. With the rise and development of machine learning, many machine learning methods have been applied to solve the TCP problem, such as reinforcement learning, clustering, ranking model and natural language processing. Particularly, reinforcement learning is a widely researched and successfully applied technique [8].

Reinforcement learning, as an unsupervised machine learning method, has achieved excellent results in many applications in the real world, such as the famous AlphaGo [9]. In previous studies, reinforcement learning [10] has been successfully applied to test case prioritization in continuous integration (TCPCI). Reinforcement learning relies on the continuous interaction between the agent and the environment to learn the corresponding skills. In the problem of TCPCI, the continuous iteration of the integration cycle makes test cases and code continuously interact. Thus, Spieker et al. [11] proposed the RETECS framework, the first reinforcement learning framework used in TCPCI.

RETECS uses an agent network to predict the priority of test cases and uses historical trajectory information (states, rewards) from the experience pool [12] to update the parameters of the agent network. All the test case priorities are generated by the agent network predictions when the agent network starts training.The deep learning experience in supervised learning indicates that when a network predicts what it has not learned, the results tend to be worse than optimal. For new test cases, there is no historical interaction information for the agent network to learn, so the agent may not be able to predict the appropriate priority for these new test cases. The reward functions used by the RETECS framework mainly focus on the failed test cases, but based on our statistical information on the 12 datasets, the new test cases are more likely fail in early executions, a feature they were not designed with in

2693-9177/22/$31.00 ©2022 IEEE
DOI 10.1109/QRS57517.2022.00088

mind.

In this paper, we first propose a priority assignment method that directly assigns the highest priority to new test cases so that they are executed first. This approach takes into account the fact that the RETECS agent network may not be able to predict the priority of new test cases reasonably and the traditional strategy that new test cases will be executed first in regression testing [13]. Secondly, consider that the highest priority approach works only until the first execution of the test case, it can only indirectly impact the subsequent execution order by affecting the reward value to impact the update of the agent network and the effect of this indirect impact can be very fragile. In order to directly guide the update of the agent, and considering the new test cases are more likely to fail in their first execution, we propose additional reward method [14]. Finally, based on the full lifecycle management of test cases, new test cases are more likely to fail in their early lifecycle and have stronger fault detection capabilities. Therefore, we may not be able to only reward the first execution of a test case, and in turn, we explore the termination period of additional rewards, i.e., the threshold of the number of rewards.

To validate the above methods, we conducted experiments on 12 datasets. Also, to mitigate the impact of stochasticity in reinforcement learning, we repeated the experiment 30 times to take the average of the results. Finally, we evaluate our proposed improved method on four metrics: NAPFD, RECALL, TTF and Duration. The results show that our methods are more effective than RETECS.

Our main contributions are as follows:

- We propose a priority assignment method for reinforcement learning-based TCPCI, which handles the first execution of new test cases more reasonable.
- Based on the fact that newly added test cases are usually more likely to fail in regression testing, we further propose an additional reward for new test cases, thus allowing the agent network to give appropriate guidance on the subsequent executions sequence of new test cases.
- The empirical study is conducted on 12 real-world datasets to verify the proposed methods, and finally, the termination timing of the additional reward for the new test case is explored experimentally to make the additional reward function more reasonable.

This paper is organized as follows: Section II describes the background and related work. Section III describes our approach in detail. Section IV describes the experimental setup, and Section V analyzes the experimental results. Section VI is the conclusion.

II. Background and related work

In this section, we mainly introduce the test case prioritization method in traditional regression testing and continuous integration testing and the application of reinforcement learning.

A. Test case prioritization in traditional regression testing

The purpose of TCP is to improve the efficiency of fault detection. Rothermel et al. [15] first proposed the formal description of TCP:

$$T' \in TS \ s.t. \ (\forall T'' \in TS)\,(T'' \neq T')\,[f\,(T') \geq f\,(T'')] \quad (1)$$

Where T' and T'' represent the permutation of test suite TS, and function f represents the fault detection efficiency.

Previous studies have shown that extracting information from test cases and then using heuristics can achieve good results on this problem. There are various kinds of information extracted, such as coverage information of test cases, historical execution information of test cases, code similarity of test cases, the complexity of test cases, user input, etc [7]. Rothermel et al [15] proposed a TCP method using the test case coverage information, including branch coverage, statement coverage and error-exposing-potential, which laid a foundation for the TCP research based on coverage information. Li et al. [13] compared multiple heuristics (Greedy algorithm, Additional Greedy Algorithm, 2-Optimal Algorithm, Hill-climbing method and Genetic algorithm) with coverage information, and noted that Additional Greedy Algorithm and 2-Optimal Algorithm performed the best. They also pointed out that it is a good choice to execute new test cases first. In the early stage of the study, the coverage information of the test cases was used the most, and as the study progressed, the history information of the test cases was used more and more. Marijan et al [16] proposed an approach named ROCKET based on the historical error data of test cases and test execution time. They use two pieces of information extracted from the test cases, so they use a weighting function to calculate the priority of the test cases.

B. Test case prioritization in continuous integration testing

In CI environment, fast integration requires regression testing to be fast due to TCP having time constraints in most cases, which may lead to unable to execute all test cases. Spieker et al. [11] define test case prioritization with time constraints as the Time-Constrained Test Case Prioritization Problem (TTCP), which is formally defined as follows:

$$T' \in TS \ s.t. \ (\forall T'' \in TS)\,(T'' \neq T')\,(f\,(T') \geq f\,(T'')) \wedge \left(\sum_{tc_k \in T'} tc_k.duration \leq M \wedge \sum_{tc_k \in T''} tc_k.duration \leq M\right) \quad (2)$$

Where tc_k represents the k^{th} test case in T' or T'' and $tc_k.duration$ represents the duration of tc_k. M represents the maximum execution time available to the test suite of a certain CI cycle.

Equations (1) and (2) show that for each test suite, we can find an optimal solution from the permutations of all test cases. However, for test suites with plenty of test cases, it is difficult to solve it positively due to resource and time constraints. In general, the process of TCP is to give each test case a priority, then rank them according to the priority, and test them

according to the order of the ranked test cases. The key and hard part of TCP is how to determine what kind of priority should give.

Under time and resource constraints, traditional software testing methods cannot meet the requirements of rapid fault detection. With the rise and rapid development of machine learning, more and more machine learning methods are applied to TCPCI. Mirarab et al. [17] proposed an improved ranking model based on a Bayesian network and introduced a feedback mechanism, which has advantages in early fault detection. Carlson et al. [18] used the test code coverage information, code complexity and actual failures of history information, and proposed a clustering method to improve the fault detection rate of test cases and reduce the number of failures that are missed when there has a time constraint. Sharif et al. [19] put forward the DeepOrder network model based on the research of Marijan et al. [16], they added the neural network and achieved better results. Spieker et al. [11] use the history information of test cases and first apply the reinforcement learning framework to TCPCI. Bertolino et al [20] made an experimental comparison of 10 machine learning methods including reinforcement learning to explore the effectiveness of the 10 methods and indicated reinforcement learning is promising. Aman et al [21] tried to use the NLP method based on test case distance to prioritize test cases, which proved that doc2vec is a good choice. Many machine learning methods have been used in test case optimization problems in CI environments, but Pan et al. [8] point out that clustering and ranking models may not be applicable, NLP has good promise but is currently less explored, while reinforcement learning not only has good promise but also has a lot of exploration and success cases.

C. Reinforcement learning for test case prioritization in continuous integration testing

Reinforcement learning as unsupervised learning is an important part of machine learning. Traditional reinforcement learning mainly includes environment, agent, state, action and reward. At every moment, the environment interacts with the agent. For example, at time *t*, there is a state s_t in the environment, the agent makes an action a_t, then we can determine the next state s_{t+1} and compute the reward value r_t according to the reward function.

Spieker et al. [11] applied reinforcement learning to TCPCI and put forward the RETECS framework for the first time. They treated TCPCI as an adaptive problem, in consideration of the fast frequency of CI leading to the fast continuous change of the test case suite. Therefore, a method that can adapt to the fast continuous change of test cases is needed. Reinforcement learning agent constantly learns from historical information to improve itself, which is an adaptive method. Therefore, they tried to apply reinforcement learning to the TCPCI, and let the agent learn the experience information from the previous integration cycle to improve itself, and then prioritize the test case.

The main interaction between the agent and the environment in the RETECS framework is shown in Figure 1. At the moment t, the states s_t is the information of test cases features in the test suite, and then the agent network makes the corresponding action a_t according to s_t (i.e., predicts the priority of test cases), and then sorts the test suite according to the a_t. Finally, the rewards r_t can be calculated based on the sorted test suites and the environment is updated to the next states s_{t+1}.

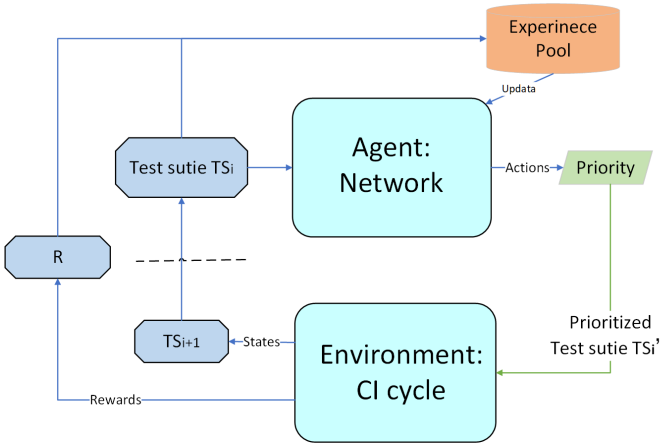

Figure 1. The interaction between agent and environment in the RETECS framework

As shown in Figure 1, RETECS uses an agent network to predict the priority of test cases. The agent network uses the experience replay mechanism [12], which is an important technique in reinforcement learning to break the correlation of sequences and to reuse the collected experience, thus allowing the same performance to be achieved with a smaller number of samples. Therefore, the quantity and quality of historical information are of particular importance.

The reward function is an important element of reinforcement learning, which is often used to guide the network of agents for updating, so the quality of the reward directly affects the learning ability of the agent. Spieker et al. also proposed three reward functions: Failure Count Reward, Test Case Failure Reward and Time-ranked Reward. They define the TCPCI as a TTCP problem, as in Equation (2). So, there will be an execution time limit in each CI cycle, and not all test cases will be in scheduling. Therefore, all three reward functions are designed for scheduled test cases, and unscheduled test cases will not be rewarded (the reward is 0). Among three reward functions, the Test Case Failure Reward (TCF) achieved the best results. TCF only rewards the test cases that are scheduled and failed, the test cases unscheduled and passed are not rewarded.

III. APPROACH

In this section, we focus on the proposed approach of priority assignment and additional reward methods for new test cases.

A. Priority assignment for new test cases

First, we give the definition of the new test cases:

***Definition** 1: New test cases*:

In this paper, new test cases are newly added test cases, i.e., they have never been executed before.

In traditional regression testing, as newly added test cases are more likely to fail, new test cases are usually executed first. RETECS is a reinforcement learning-based framework and uses an agent network to predict the priority of test cases. The agent network works well when it is able to learn features from a sufficient amount of historical interaction information from these test cases. However, for newly added test cases, they don't have any history interaction information for the agent network to learn the features. And newly added test cases are usually for new or modified code fragments, making their features more different from those of previous test cases. Based on the experience with deep neural networks, the network often predicts the results in a 'random' way in the above cases. Therefore, this method of predicting priorities by the agent network is not appropriate for the new test cases.

To execute the newly added test case first in CI testing, we try to combine the traditional method with the RETECS method and propose the priority assignment method for newly added test cases:

***Definition** 2: Priority Assignment (PA)*:

$$PA\left(tc\right) = \begin{cases} 1 & if\ tc\ is\ new \\ RETECS.agent\left(tc\right) & otherwise \end{cases} \tag{3}$$

Where tc means a test case.

The new test cases are directly assigning the highest priority and the old test cases are using the agent network of RETECS to predict the priority.

B. Additional reward for new test cases

The approach of directly assigning the highest priority to the test cases only works until the first execution of the test cases and can not directly affect the update of the agent network. Therefore, the PA method is mainly working for the first execution of the newly added test cases. Since newly added test cases are more likely to fail, they have a strong fault detection capability in the early stage of their life cycle, so we should make the newly added test cases be executed as early as possible within a certain period of time. The update of the agent network is directly guided by the reward, therefore, we considered redesigning the reward to allow the new test cases to be executed first as much as possible. Inspired by Savinov et al. [14], we use the additional reward method. Among the three reward functions used by RETECS, the TCF achieved the best results, so we use this reward as the baseline reward and add an additional reward to it. TCF is defined as follows:

$$TCF(tc) = \begin{cases} tc.verdict & if\ tc \in TS \wedge scheduled \\ 0 & otherwise \end{cases} \tag{4}$$

Where $tc.verdict$ indicates the execution result of test case tc, with 0 means tc passed in the testing, 1 means tc failed in the testing.

TCF only rewards the scheduled and failed test cases because they are the ones that make sense and need to be executed as early as possible. The unscheduled and passed test cases cannot provide valid information to guide the improvement of the quality of test case sequencing, so they are not rewarded. For new test cases, as they are more likely to fail, so they may get a reward. But TCF cannot guarantee that they will be recognized by the agent and prioritized in subsequent executions, due to the rewards being the same for all the failed test cases no matter whether they are new or old test cases. Therefore, we try to add an additional reward for new test cases, so that the agent can distinguish between old and new test cases and prioritize the execution of new test cases. And the unscheduled but newly added test cases also will give additional rewards, because they have strong fault detection capability in their early lifecycle although they are not unscheduled.

TCF focuses on failed test cases while we focus on new test cases. To weigh the importance of failed test cases and new test cases, we considered three reward strategies: new test cases are less, equally or more important than failed test cases. We use three different additional reward values to distinguish the above three reward strategies. Due to the value of TCF reward for failed test cases is 1, and the neural network used by RETECS is a classifier from Scikit-Learn [22] which does not allow fractional inputs. If we use the original TCF, the reward strategy that the new test case is less important than the old test case will generate a fraction, which is not allowed by the network. Therefore, we double the TCF value and then set reward values 1, 2, 3 to represent the new test cases are less, equally or more important than the old test cases, respectively.

Based on the above discussion and analysis, we propose the additional reward method based on TCF, which is defined formally as follows:

***Definition** 3:* Additional Reward Based on Test Case Failure Reward (ATCF):

$$ATCF(tc) = AR(tc) + 2 * TCF(tc) \tag{5}$$

where

$$AR(tc) = \begin{cases} v & if\ tc\ is\ new \\ 0 & otherwise \end{cases} \tag{6}$$

where there are three values of v: 1, 2, 3. We use $ATCF_1$, $ATCF_2$, $ATCF_3$ in the following to represent the three reward strategies respectively.

C. Termination period for additional reward

In the previous subsection, we described the method of giving additional rewards for new test cases. The simplest way is to give additional rewards only for the first execution of a new test case. Since new test cases are more likely to fail and have a strong fault detection capability in their early lifecycle, it may not be optimal to give additional rewards only for the

first execution of a new test case. Therefore, we try to give additional rewards for subsequent executions of new test cases as well. However, the number of rewards cannot be unlimited, and we need to stop the additional rewards at a certain period. We plan to use the number of test case executions as the stopping condition for the additional reward, i.e., stop the additional reward when the new test case executes more than a threshold number of times.

IV. EXPERIMENTS SETUP

In this section, we present the experimental setup of the priority assignment and additional reward methods we proposed based on the RETECS framework. First, we list the research questions to be studied (section IV-A). Then we introduce the result evaluation criteria (section IV-B) and the datasets (section IV-C) we used. Finally, the selection of parameters for the experiment is described in detail (section IV-D).

A. Research questions

We propose three questions as the main exploration of our experiments:

- **RQ1**: Does our proposed PA method improve the fast fault detection capability of the test sequence compared with the priority assignment method used by RETECS?
- **RQ2**: Does the ATCF improve the fault detection performance compared to TCF? Which of the three additional strategies *($ATCF_1$, $ATCF_2$, $ATCF_3$)* is better?
- **RQ3**: When is the best time to terminate additional rewards for new test cases?

RQ1 explores whether the traditional regression testing approach of executing new test cases first can be combined with the RETECS method to obtain better performance. In comparing the two priority assignment methods, the reward function used is the TCF reward, which facilitates the RQ2.

RQ2 explores the effectiveness of the additional reward method and compares the three reward strategies. In fact, RQ2 is proposed on top of the experimental results of RQ1, so the priority assignment method used is the best method from RQ1.

RQ2 only gives the additional reward for the new test cases in their first execution. RQ3 examined when to stop the additional reward for new test cases is better, i.e., the threshold for the number of additional rewards to be given. Therefore, RQ3 is a further exploration based on RQ2, using the best configuration from RQ2.

B. Evaluation criterion

1) NAPFD: Rothermel et al. [15] proposed APFD (Average percentage of failures detected) to evaluate the effectiveness of TCPCI. It is only applicable when all test cases are detected, however, RETECS sets a maximum execution time for each CI cycle, so it cannot execute all test cases and thus we cannot use APFD. Qu et al. [23] proposed the NAPFD (Normalized average percentage of failures detected) evaluation standard, which can be used when not all test cases are detected. So, we use NAPFD as the evaluation standard and the calculation formula of NAPFD is as follows:

$$NAPFD\left(TS_i\right) = p - \frac{\sum_{j \in TS_i^{fail}} rank\left(j\right)}{|TS_i^{fail}| \times |TS_i|} + \frac{p}{2 \times |TS_i|} \quad (7)$$

where

$$p = \frac{|TS_i^{fail}|}{|TS_i^{all,fail}|} \quad (8)$$

Where $|TS_i^{fail}|$ represents the number of detected failed test cases in the test case suite TS_i while $|TS_i^{all,fail}|$ represents the number of all failed test cases in the test case suite TS_i. $|TS_i|$ represents all number of test cases in the test case suite TS_i and $rank\left(j\right)$ represents the location of the jth failed test case in TS_i. The value of NAPFD is between [0,1] and the greater the better.

In addition to using NAPFD as the evaluation standard, we also use RECALL, TTF, and Duration as the evaluation standard [24].

2) RECALL: RECALL is defined as the formula (8), which reflects the average ratio of detected failed test cases to all failed test cases. Obviously, the value of Recall is between [0,1] and the larger the better, because we hope to detect as many failed test cases as possible in a limited execution time.

3) TTF: TTF (Test to Fail) is the average position of the first failed test case in the test suite. The smaller the value the better, as the smaller the value means the more forward the position of the first failed test case, so the error can be detected more quickly.

4) Duration: The duration indicates the average consumed time by our algorithm framework and contains the time spent for reward calculation, the time spent for test suite sorting, the time spent for agent updating, etc. Therefore, the smaller the duration, the better, because more test cases can be executed within the execution time limit. Meanwhile, the above three metrics are meaningful only if the duration of our proposed method is not longer than the duration of RETECS. For example, if method *a* has a NAPFD of 80% but a Duration of 8s, while method *b* has a NAPFD of 50% but a Duration of 0.5s, we cannot directly say that method *a* is better than method *b*. Therefore, we should observe this metric first, and only when the Duration of different methods is basically the same, we can use other metrics to distinguish the advantages and disadvantages of different methods.

C. Datasets

We obtained 14 industrial datasets used by Yang et al. [24]. But according to Bagherzadeh et al. [25], GSDTSR is a Google Shared Dataset with test suites developed using different programming languages, therefore, we discarded this dataset in our experiments as they did. Furthermore, we observed that the Apache Hive dataset has particularly short CI cycles. Considering that reinforcement learning is a method that requires learning in constant interaction, such short cycles are not sufficient to let the agent learn good strategies, therefore, the Apache Hive dataset was also discarded.

Table 1. Information statistics of 12 industrial datasets

Dataset	Test cases	CI cycles	Results	Failure rate
Apache Commons	457	423	332650	0.02%
Apache Drill	556	376	8887	2.28%
Apache Parquet	273	1190	205770	0.15%
Apache Tajo	314	4125	903301	0.30%
Dspace	106	3230	204161	1.82%
Google Auto	46	530	14601	0.19%
Google Closure	360	2257	663470	0.07%
Google Guava	433	772	877466	0.02%
IOF/ROL	1941	320	32260	28.79%
Mybatis	303	899	815598	0.15%
Paintcontrol	89	352	25594	19.36%
Rails	2010	3263	781273	0.62%

The statistical information of the rest 12 datasets is shown in Table 1. The meaning of the column names are as follows: 'Test cases' is the number of unique test cases, 'CI cycles' is the number of CI cycles, 'Results' is the number of test cases execution results, 'Failure rate' is the percentage of failures of all test cases executions.

From Table 1, we can see that most of the datasets have more than 100 test cases, Rails has up to 2010 test cases, while Google Auto has less than 50 test cases. All datasets have more than 300 CI cycles, with four datasets: Apache Parquet, Apache Tajo, Dspace and Rails having more than 1000 CI cycles. All datasets except Apache Drill have more than 10,000 execution results. The average failure rate of test cases is low for most of the datasets, except for Apache Drill, IOF/ROL, and Paintcontrol, where the average failure rate of test cases is greater than 2%. The low failure rate is one of the main issues limiting TCPCI, but the problem is bound to exist again as programmer quality improves. Therefore, we need to extract as much useful feature information as possible from the limited available information to solve the TCPCI problem.

D. Selection of parameters

The main experimental parameters were carried out as Spieker et al. [11] did, such as the execution time limit of every CI cycle, the deep of the agent network and the number of training steps. In consideration of the stochastic nature of reinforcement learning, our experimental results are also the average value of 30 repeated experiments.

In the PA method, the priority of the new test cases is directly set to 1, because the range of the prediction priority of the agent network is [0,1], thus we can ensure that new test cases are prioritized for execution. The reward values of ATCF were set to 1, 2 and 3 to represent the three reward strategies: new test cases are less, equally or more important than failed test cases, respectively.

Before determining the threshold for the number of additional reward times, two definitions are given:

***Definition** 4:* Average Failure Rate of All Executions (AFRAE):

AFRAE is the ratio of the number of failed execution results to the total number of execution results for all test cases in a dataset over their lifetime, i.e. the 'failure rate' column in Table 1. It indicates the average failure rate of a test case over its lifetime.

***Definition** 5:* Average Failure Rate of i^{th} Executions (AFRIE):

AFRIE is the ratio of the number of test cases that failed in the i^{th} execution of all test cases to the total number of test cases in a dataset. It represents the failure rate of a test case in the i^{th} execution.

We counted the AFRIE for the first 10 executions of the new test cases and compared it with AFRAE, the results are shown in Table 2, where the shaded part indicates that AFRIE $>$ AFRAE and means the test case is more likely fail for that execution.

As can be seen from Table 2, AFRIE $>$ AFRAE for the first 6 executions of the test cases on most of the datasets. Therefore, we set the max threshold to 6 as the bound of the number of additional reward times of new test cases in RQ3. That is, we explore the 6 cases with thresholds from 1 to 6. In addition, we note that the first 6 executions of the test cases in the Google Closure dataset are error-free, and thus our proposed approach for the new test cases may not produce good results on this dataset.

V. RESULTS AND ANALYSIS

A. RQ1: Does our proposed PA method improves the fast fault detection capability of the test sequence compared with the priority assignment method used by RETECS?

To compare the fast fault detection capability of the PA and RETECS frameworks, we conducted experiments with the two methods (the reward function used in both methods is TCF), and the average results of 30 repeated experiments for 12 datasets are shown in Table 3.

We need to analyze the Duration metric first, because if our new method adds a significant amount of duration time, then the method is meaningless. As we see in Table 3, our new method PA is surprisingly smaller than the RETECS method in terms of average duration for both the Apache Commons, Apache Drill, Google Guava, IOF/ROL, Mybatis, Paintcontrol datasets as well as the average of 12 datasets. This is different from our subjective guess because we are modifying the RETECS framework by adding some conditional judgments, and the PA method should take more time to be consistent with the subjective inference. However, this objective fact exists because the machine (CPU) may run at different speeds at different moments. From the data in Table 3, it can be calculated that PA takes at most 0.009 seconds less than the RETECS method (in the Apache Commons dataset). In addition, PA takes less time than the RETECS method, indirectly indicating that the PA method is in the same level as RETECS in terms of time consumption, and therefore, the comparison of other metrics is meaningful.

On NAPFD, the PA method has improved on 8 datasets compared with the RETECS method, but the improvement on Apache Parquet and Dspace dataset is smaller, less than 1%; the performance on Apache drill, Apache Tajo, Google Closure, Google Guava 4 datasets has decreased, among which the performance on Google Closure dataset has decreased more than 3%.

Table 2. Average failure rate per execution for the first 10 executions

DATASET	AFRIE										AFRAE
	1	2	3	4	5	6	7	8	9	10	
Apache Commons	0.22%	0.22%	0.22%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.02%
Apache Drill	18.71%	3.42%	1.26%	1.44%	1.08%	1.08%	1.44%	1.98%	1.26%	0.72%	2.28%
Apache Parquet	1.10%	1.10%	1.47%	0.37%	0.73%	0.73%	0.37%	0.73%	1.10%	1.10%	0.15%
Apache Tajo	5.41%	5.10%	1.91%	0.96%	0.32%	0.32%	1.27%	1.59%	0.32%	0.64%	0.30%
Dspace	1.89%	1.89%	2.83%	1.89%	0.94%	0.94%	0.94%	0.94%	1.89%	0.94%	1.82%
Google Auto	2.17%	2.17%	0.00%	0.00%	2.17%	2.17%	2.17%	0.00%	0.00%	0.00%	0.19%
Google Closure	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.28%	0.00%	0.00%	0.28%	0.07%
Google Guava	0.46%	0.46%	0.69%	0.23%	0.23%	0.23%	0.00%	0.00%	0.00%	0.00%	0.02%
IOF/ROL	42.71%	26.53%	24.21%	29.98%	28.13%	21.84%	24.27%	20.66%	22.41%	18.13%	28.79%
Mybatis	1.98%	1.98%	1.98%	1.65%	1.32%	1.32%	0.66%	0.66%	0.66%	0.33%	0.15%
Paintcontrol	26.97%	23.60%	22.47%	22.47%	20.22%	28.09%	24.72%	22.47%	11.24%	14.61%	19.36%
Rails	4.53%	2.79%	1.34%	1.44%	0.85%	0.65%	0.40%	0.40%	0.25%	0.45%	0.62%

Table 3. The Duration, NAPFD, RECALL and TTF value of different priority assignment methods on 12 datasets

Dataset	Duration(s)		NAPFD		RECALL		TTF	
	RETECS	PA	RETECS	PA	RETECS	PA	RETECS	PA
Apache Commons	0.1104	**0.1010**	25.14%	**31.47%**	91.94%	**94.09%**	348.53	**322.28**
Apache Drill	0.0115	**0.0102**	**32.58%**	32.55%	**64.72%**	62.23%	**12.33**	12.52
Apache Parquet	**0.0196**	0.0200	48.18%	**48.21%**	**92.34%**	**92.34%**	48.39	**48.10**
Apache Tajo	**0.0311**	**0.0311**	**28.26%**	26.53%	**70.98%**	70.54%	**85.23**	88.67
Dspace	**0.0107**	0.0108	34.35%	**35.18%**	64.52%	**65.71%**	17.59	**17.36**
Google Auto	**0.0258**	0.0264	20.43%	**21.94%**	51.03%	**53.17%**	7.92	**6.33**
Google Closure	**0.0366**	0.0371	**25.10%**	22.06%	**64.50%**	63.11%	**107.19**	111.15
Google Guava	0.1820	**0.1742**	**66.69%**	65.50%	**93.72%**	93.58%	**237.58**	244.72
IOF/ROL	0.0217	**0.0207**	25.62%	**28.13%**	35.06%	**37.50%**	6.53	**4.42**
Mybatis	0.1452	**0.1425**	66.40%	**69.06%**	85.32%	**86.48%**	135.90	**113.73**
Paintcontrol	0.0150	**0.0149**	48.17%	**50.49%**	59.51%	**61.68%**	5.93	**5.68**
Rails	**0.0605**	0.0613	30.81%	**35.61%**	46.65%	**49.20%**	49.85	**41.04**
Average	0.0558	**0.0542**	37.64%	**38.89%**	68.36%	**69.14%**	88.58	**84.67**

On RECALL, the PA method is comparable to RETECS on the Apache Parquet dataset (because the data in the table is obtained by rounding, the same value means the performance is very close). In addition, the PA method improves over the RETECS method on 7 datasets, but the improvement on the Apache Parquet dataset is smaller, less than 1%; the performance decreases on the Apache Drill, Apache Tajo, Google Closure, Google Guava 4 datasets, of which Apache Drill, Google Closure dataset performance drop is larger, more than 1%.

On TTF, PA method performance is the same as NAPFD and RECALL, both outperforming RETECS on 8 datasets, and underperforming RETECS on 4 datasets. PA method performs best on the Apache Commons and Mybatis datasets, both outperforming RETECS by more than 20 positions. PA method performs worst on the Google Guava dataset, with TTF increasing by 7.14 positions.

In the overall analysis of the three metrics NAPFD, RECALL and TTF, the PA method outperforms the RETECS method on the average of all 12 datasets. But in the four datasets: Apache Drill, Apache Tajo, Google Closure and Google guava, PA method do not perform better than RETECS. Among the four datasets, the gap in the Google Closure dataset between PA and RETECS in the three metrics is the largest because all of the new test cases in the Google Closure dataset are error-free in their first execution. When we execute them first, it will cause the truly failed test cases to be executed after the new but passed test cases, thus leading to a decrease in failure detection efficiency.

Further, we performed a hypothesis test on the results of RQ1 to verify whether there is a significant difference between PA and RETECS. The two hypotheses are:

H0: No significant difference between PA and RETECS.

H1: There is a significant difference between PA and RETECS.

We used the T-Test on NAPFD, RECALL, and TTF to obtain p-values and then compared them with the significance level α (which is usually set very small), and if $p \leq \alpha$, then H0 can be rejected and H1 accepted, which means that PA and RETECS are significantly different. Our sample size is 30 (30 replicate experiments), and the sample is the metrics results of each experiment. Our table of p-values is shown in Table 4.

The common values of α are 0.01, 0.05, 0.1, but unfortunately, PA did not achieve the desired results at the above significance levels. It is when $\alpha = 0.25$ that PA begins to achieve the desirable results, with significant differences in 8 datasets on NAPFD, 7 datasets on RECALL, and 6 datasets on TTF. From this point of view, the improvement of the PA method is limited. This is due to the fact that as CI proceeds,

Table 4. The p-values between PA and RETECS methods on NAPFD, RECALL and TTF in 12 datasets

DATASET	NAPFD	RECALL	TTF
Apache Commons	2.65E-01	1.00E+00	6.67E-01
Apache Drill	9.72E-01	6.99E-01	8.23E-01
Apache Parquet	5.27E-03	2.26E-01	3.61E-01
Apache Tajo	9.88E-02	7.21E-10	8.59E-08
Dspace	9.46E-07	3.02E-07	1.56E-04
Google Auto	1.82E-01	8.76E-01	2.91E-02
Google Closure	9.63E-03	2.63E-01	2.75E-01
Google Guava	2.09E-01	1.82E-01	1.99E-01
IOF/ROL	2.93E-01	2.00E-01	6.44E-01
Mybatis	5.76E-02	9.16E-02	2.05E-01
Paintcontrol	4.19E-01	3.67E-01	7.92E-01
Rails	5.13E-04	2.46E-01	8.49E-02

the agent may learn to prioritize test cases with high error probabilities by giving them a large priority; however, our PA method forces all new test cases to be executed first, which can somewhat undermine fault detection efficiency. But in total, our approach has a small performance improvement over RETECS. Further, we propose additional reward mechanisms to motivate the agent to better handle the new test cases.

B. RQ2:Does the ATCF improve the fault detection performance compared to TCF? Which of the three additional strategies ($ATCF_1$, $ATCF_2$, $ATCF_3$) is better?

Due to the fact that the reward function used to explore the test case priority assignment method in RQ1 was TCF, we conducted 30 repeated experiments in RQ2 using only three reward functions, $ATCF_1$, $ATCF_1$ and $ATCF_1$. The experimental results of TCF were obtained from PA in RQ1. The Duration, NAPFD, RECALL and TTF results are shown in Tables 4, 5, 6 and 7.

We still analyze the Duration metrics first, as we can see in Table 4, the TCF reward fails to take the shortest time on five datasets, that is, there still seems to be a discrepancy between actual and theoretical, for the same reason as we explained in RQ1. At the same time, we can see that the difference in different reward functions is small, so the NAPFD, RECALL and TTF metrics are meaningful.

Table 5. The Duration of three reward functions on 12 datasets

Dataset	TCF	$ATCF_1$	$ATCF_2$	$ATCF_3$
Apache Commons	**0.1010**	0.1036	0.1108	0.1071
Apache Drill	**0.0102**	0.0248	0.0221	0.0240
Apache Parquet	**0.0200**	0.0240	0.0226	0.0244
Apache Tajo	0.0311	0.0307	**0.0300**	0.0300
Dspace	**0.0108**	0.0150	0.0124	0.0156
Google Auto	0.0264	0.0378	**0.0110**	0.0355
Google Closure	0.0371	0.0369	0.0383	**0.0364**
Google Guava	**0.1742**	0.1840	0.1769	0.1829
IOF/ROL	**0.0207**	0.0257	0.0224	0.0261
Mybatis	0.1425	0.1363	**0.1196**	0.1295
Paintcontrol	**0.0149**	0.0171	0.0159	0.0173
Rails	0.0613	0.0613	**0.0605**	0.0609
Average	0.0542	0.0581	**0.0535**	0.0575

From Tables 5, 6 and 7, we can see that the three additional reward functions, $ATCF_1$, $ATCF_1$ and $ATCF_1$, all have an improvement over the TCF reward on the 12 datasets. This indicates that our exploration of the additional reward mechanism is in the right direction, and giving an additional reward to the new test cases can improve the fast failure detection ability of the test suites. We plotted the NAPFD plots for TCF and $ATCF_1$ for all cycles with failures on 12 datasets, as shown in Figure 2. From the figure, we can see that $ATCF_1$ rewards better than TCF for almost all test cycles on the 12 datasets. This is because new test cases are not concentrated in one cycle, many test cycles have new test cases, so the additional rewards can give good guidance to the agent to motivate it to learn how to optimize the new test cases for subsequent execution.

Among the three additional rewards, $ATCF_1$ improved the most on the average of the 12 datasets, having 17.99%, 9.9% and 30.14 positions improvement in NAPFD, RECALL and

Table 6. The NAPFD of three reward functions on 12 datasets

Dataset	TCF	$ATCF_1$	$ATCF_2$	$ATCF_3$
Apache Commons	31.47%	**52.48%**	42.72%	50.52%
Apache Drill	32.55%	56.47%	**56.48%**	42.49%
Apache Parquet	48.21%	**65.19%**	64.87%	63.96%
Apache Tajo	26.53%	**39.73%**	39.25%	39.07%
Dspace	35.18%	53.36%	**54.64%**	51.57%
Google Auto	21.94%	**63.30%**	63.04%	59.38%
Google Closure	22.06%	**45.18%**	43.26%	42.90%
Google Guava	65.50%	**83.01%**	77.43%	82.99%
IOF/ROL	28.13%	**39.87%**	35.95%	30.21%
Mybatis	69.06%	80.94%	**81.64%**	81.55%
Paintcontrol	50.49%	64.65%	66.37%	**67.38%**
Rails	35.61%	**38.39%**	38.33%	38.17%
Average	38.89%	**56.88%**	55.33%	54.18%

Table 7. The RECALL of three reward functions on 12 datasets

Dataset	TCF	$ATCF_1$	$ATCF_2$	$ATCF_3$
Apache Commons	94.09%	98.15%	99.05%	**99.80%**
Apache Drill	62.23%	74.98%	**75.07%**	66.29%
Apache Parquet	92.34%	**93.61%**	92.63%	92.77%
Apache Tajo	70.54%	**83.07%**	81.02%	81.86%
Dspace	65.71%	84.19%	**85.03%**	82.63%
Google Auto	53.17%	**73.97%**	73.97%	72.22%
Google Closure	63.11%	**76.45%**	74.35%	74.67%
Google Guava	93.58%	97.21%	95.45%	**97.45%**
IOF/ROL	37.50%	**49.88%**	45.58%	39.21%
Mybatis	86.48%	93.32%	**93.88%**	93.80%
Paintcontrol	61.68%	73.29%	74.34%	**76.61%**
Rails	49.20%	**50.25%**	50.25%	49.37%
Average	69.14%	**79.03%**	78.39%	77.22%

Table 8. The TTF of three reward functions on 12 datasets

Dataset	TCF	$ATCF_1$	$ATCF_2$	$ATCF_3$
Apache Commons	322.28	**226.11**	281.46	238.77
Apache Drill	12.52	**4.31**	4.40	6.59
Apache Parquet	48.10	27.01	**26.45**	27.36
Apache Tajo	88.67	87.46	**83.31**	86.02
Dspace	17.36	9.23	**8.86**	9.77
Google Auto	6.33	**1.66**	1.79	1.86
Google Closure	111.15	**81.41**	82.47	83.94
Google Guava	244.72	**108.33**	159.65	109.73
IOF/ROL	4.42	1.93	**1.91**	4.28
Mybatis	113.73	68.57	**65.47**	68.56
Paintcontrol	5.68	2.94	**2.67**	3.09
Rails	41.04	35.38	35.51	**34.32**
Average	84.67	**54.53**	62.83	56.19

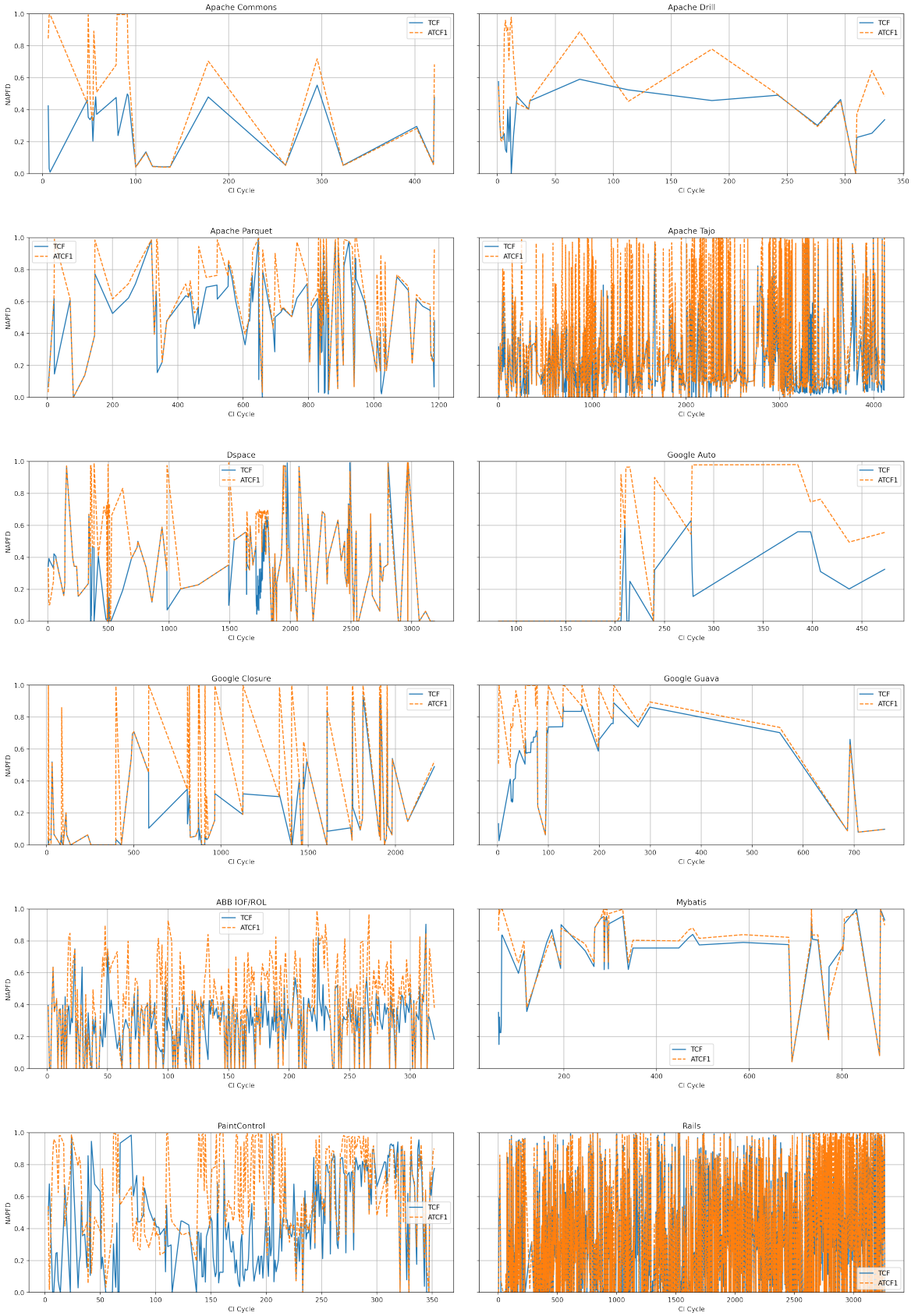

Figure 2. Comparison of reward functions: TCF and $ATCF_1$ in NAPFD on all 12 datasets

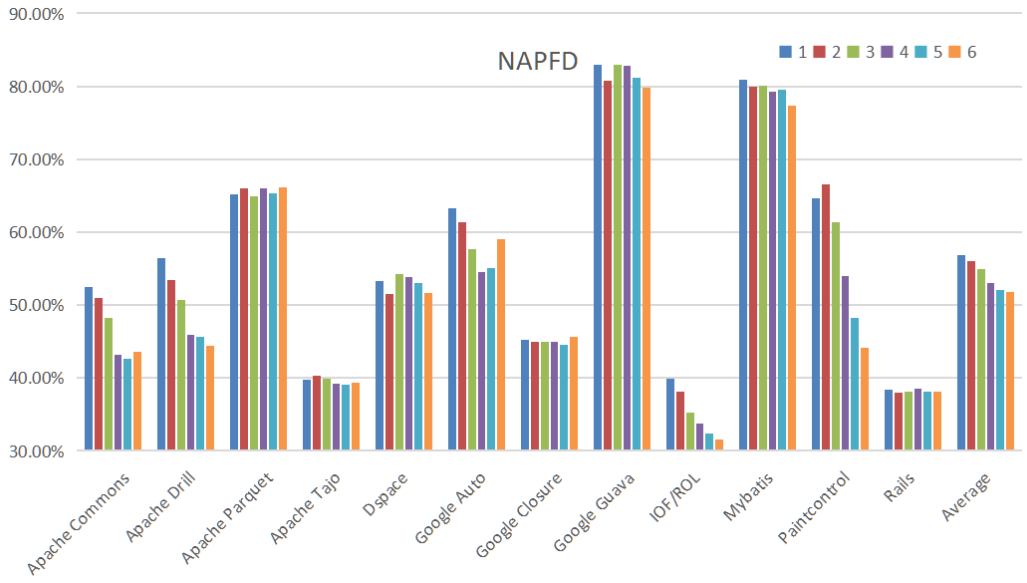

Figure 3. Comparison of the first 1 to 6 executions of $ATCF_1$ in NAPFD.

TTF metrics, respectively. This indicates that the importance of the new test cases should be weaker than the failed test cases in the reward design. This is because most of the new test cases actually do not make failures (as in Table 2), and we are using the total reward strategy in designing the additional rewards, and thus the test cases that do not make failures will receive an additional reward value that may be greater than ($ATCF_3$) or equal to ($ATCF_2$) the reward value of the old but failed test cases, which in turn may prompt the agent to rank these new but passed test cases ahead of the old but failed test cases in the subsequent execution. This situation violates the assumption underlying TCP based on historical information, i.e. the test cases that failed in the past are more likely to fail in the future, so we should execution these test cases first in the new CI cycle [26].

In summary, additional rewards are very effective, and the $ATCF_1$ reward works best with giving a larger reward value to the failed test cases. It may reflect that failed test cases should be more important than new test cases in the reward design.

C. RQ3: When is the best time to terminate additional rewards for new test cases?

We set the threshold of the number of times of additional rewards from 1 to 6 executions based on failure information in 12 datasets and conducted experiments using the best configurations in RQ2. In RQ3, we no longer focus on the Duration metric because at the code level, and based on the experience of RQ1 and RQ2, the difference in Duration will be small. The comparison results on the three metrics NAPFD, RECALL, and TTF are shown in Figures 3, 4 and 5. In Figures 3 and 4, the taller the bar, the better; in Figure 5, the shorter the bar, the better.

As can be seen from Figure 3, on 6 datasets (Apache Commons, Apache Drill, Google Auto, IOF/ROL, Mybatis, Paintcontrol), the results of NAPFD gradually worse as the number of additional reward times increases. In the other 6 datasets, the results remained roughly the same with the increasing number of additional rewards times.

From Figure 4, we can see that, on 7 datasets (Apache Commons, Apache Drill, Apache Parquet, Apache Tajo, Google Auto, IOF/ROL, Mybatis and Paintcontrol), the results of RECALL become progressively worse as the number of additional reward times increases. In the Rails dataset, the results improve as the number of reward times increases, and in the other four datasets, the results remain basically the same as the number of additional reward times increases.

In Figure 5, the low height of the bar chart means better TTF results. Only on the Apache Commons dataset, the TTF shows a trend of getting better as the number of additional reward times increases. And the range of variation in TTF on the Apache Commons dataset is wider than others, this can largely influence the average TTF of 12 datasets and decay its significance.

Taking into account the above analysis and the trend presented by the average results of the 12 datasets, only giving an additional reward to the first execution of the test case works best. The reason for this result is that the new test case is more likely to fail only at the early stage of its existence (about the first six executions), and giving an additional reward to the first execution of the test case can influence the agent network to give a higher priority to its subsequent executions, i.e. the agent network have the ability of short-term memory. If additional rewards are also given to the subsequent executions, this effect is passed to remote executions, although by then the new test case has become an old test case, which then reduces the fast fault detection capability of the test suite.

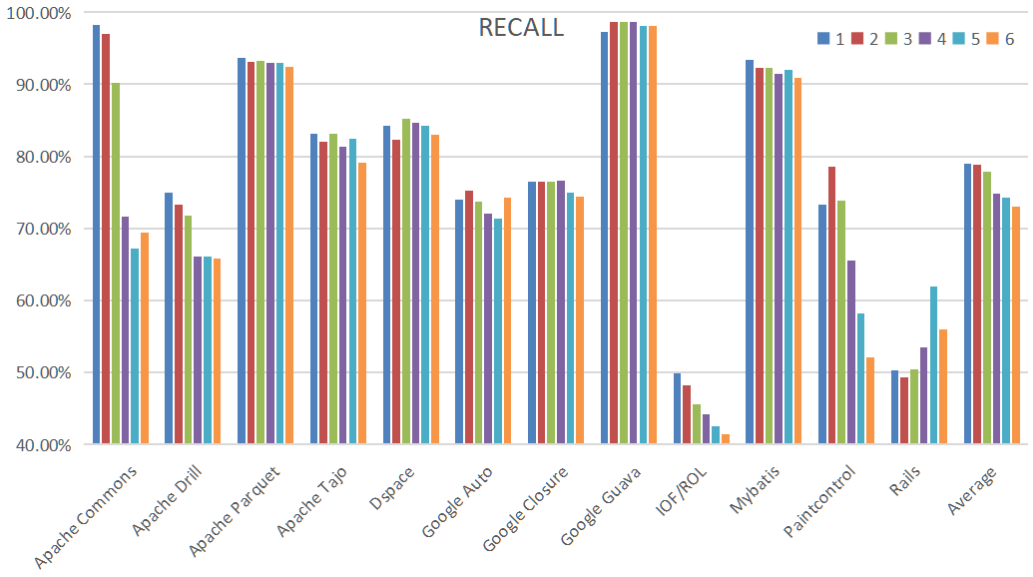

Figure 4. Comparison of the first 1 to 6 executions of $ATCF_1$ in RECALL.

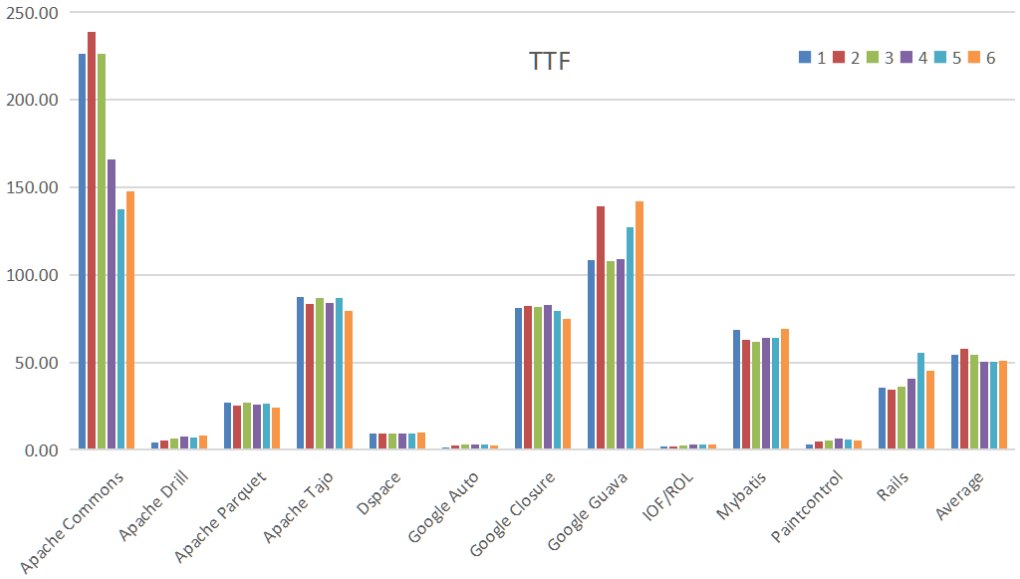

Figure 5. Comparison of the first 1 to 6 executions of $ATCF_1$ in TTF.

VI. Conclusion

The reinforcement learning-based RETECS framework has been successfully applied to the TCPCI problem. Priority assignment and reward functions are the main components of the RETECS framework and should take into account the characteristics of the test cases when designing. Based on previous studies and our statistical information, new test cases tend to be more likely to fail than old ones, and if we can handle new test cases well, it will have a positive effect on the overall testing. Therefore, in this paper, our research focuses on the new test cases.

We propose the PA and ATCF methods based on the RETECS framework and considered the features of newly added test cases. PA is an improvement of the priority assignment method of the RETECS framework, and we directly assign a high priority to new test cases so that they are executed first. ATCF is an additional reward based on TCF, due to the TCF reward being designed for failed test cases while we focus on new test cases, thus we compare the importance of new test cases with failed test cases and propose three specific reward functions. The experiments show that ATCF can significantly improve the fast fault detection capability and $ATCF_1$ achieved the best results. We also explore the best termination time of additional rewards by setting the

threshold of the number of additional rewards times, and the experimental results show that it is enough to only give the additional reward to the first execution of new test cases.

Although our proposed PA and ATCF methods improve the performance of the RETECS framework, our research is limited to priority assignment methods and the reward function proposed by Spieker et al. Due to RETECS being an application of reinforcement learning, the design of the agent is also very important, and the network structure used in RETECS is relatively simple, so the study of the network structure of the agent is also very necessary. Therefore, our future research will focus on (1) more extensive experiments on the additional reward function mechanism to investigate whether it is generalizable for different reward functions and (2) research the agent network part in terms of feature input, network structure, hyperparameter setting, policy update, etc.

ACKNOWLEDGMENT

The work described in this paper is supported by the National Natural Science Foundation of China under Grant No.61872026, 61902015 and 62077003.

REFERENCES

[1] P. Duvall, S. Matyas, and A. Glover, *Continuous integration: Improving software quality and reducing risk*, 2007.

[2] Y. Yu, B. Vasilescu, H. Wang, V. Filkov, and P. Devanbu, "Initial and eventual software quality relating to continuous integration in github," *ArXiv.*, June 2016.

[3] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Test. Verif. Reliab.*, vol. 22, pp 67-120, March 2012.

[4] A. A. Saifan, "Test Case Reduction Using Data Mining Classifier Techniques," *J. Softw.*, vol. 11, no. 7, pp. 656–663, 2016.

[5] R. Kazmi and Dayang N. A. Jawawi and R. Mohamad and I. Ghani, "Effective Regression Test Case Selection: A Systematic Literature Review," *ACM Comput. Surv.*, vol. 50, no. 2, pp. 29:1–29:32, 2017.

[6] T. P. Jacob and T. A. Ravi, "Optimization of Test Cases by prioritization," *J. Comput. Sci.*, vol. 9, no. 8, pp. 972–980, 2013.

[7] P. Lima, A. Jackson, Vergilio, R. Silvia," Test case prioritization in continuous integration environments: A systematic mapping study," *Inf. Softw. Technol.*, vol. 121, May 2020.

[8] R. Pan and Bagherzadeh, Mojtaba and Ghaleb, A. Taher and L. Briand," Test case selection and prioritization using machine learning: A systematic literature review," *Empirical Softw. Engg.*, vol. 27, Mar 2022.

[9] J. X. Chen," The Evolution of Computing: AlphaGo," *Comput. Sci. Eng.*, vol. 18, no. 4, pp. 4–7, 2016.

[10] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, 1988.

[11] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis.*, pp 12-22, 2017.

[12] P. Wawrzynski, "Real-time reinforcement learning by sequential Actor-Critics and experience replay," *Neural Networks.*, vol. 22, no. 10, pp 1484–1497, 2009.

[13] Z. Li and M. Harman and R. M. Hierons, "Search Algorithms for Regression Test Case Prioritization," *IEEE Trans. Software Eng.*, vol.33, no. 4, pp 225–237, 2007.

[14] N. Savinov, A. Raichuk, D. Vincent, R. Marinier, M. Pollefeys, T. P. Lillicrap, and S. Gelly. "Episodic curiosity through reachability." *International Conference on Learning Representations.*, Jul 2019.

[15] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. "Test case prioritization: an empirical study," *International Conference on Software Maintenance.*, pp 179-188, September 1999.

[16] D. Marijan, A. Gotlieb, and S. Sen, "Test case prioritization for continuous regression testing: An industrial case study," *International Conference on Software Maintenance.*, pp 540-543, 2013.

[17] S. Mirarab and L. Tahvildari. "An empirical study on bayesian network-based approach for test case prioritization," *International Conference on Software Testing, Verification, and Validation.*, pp 278–287, 2008.

[18] R. Carlson, H. Do, and A. Denton. "A clustering approach to improving test case prioritization: An industrial case study," *International Conference on Software Maintenance.*, pp 382-391, 2011.

[19] A. Sharif, D. Marijan, and M. Liaaen, "Deeporder: Deep learning for test case prioritization in continuous integration testing," *International Conference on Software Maintenance and Evolution.*, pp 525–534, 2022.

[20] 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," *International Conference on Software Engineering.*, pp 1-12, 2020.

[21] H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "A comparative study of vectorization-based static test case prioritization methods," *Euromicro Conference on Software Engineering and Advanced Applications.*, pp 80-88, 2020.

[22] F. Pedregosa and G. Varoquaux and A. Gramfort and V. Michel and B. Thirion and O. Grisel and M. Blondel and P. Prettenhofer and R. Weiss and V. Dubourg and J. VanderPlas and A. Passos and D. Cournapeau and M. Brucher and M. Perrot and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp 2825–2830, 2011.

[23] X. Qu, M. B. Cohen, and K. M. Woolf, "Combinatorial interaction regression testing: A study of test case generation and prioritization," *International Conference on Software Maintenance.*, pp 255–264, 2007.

[24] Y. Yang, C. Yue Pan, Z. Li, and R. L. Zhao, "Adaptive reward computation in reinforcement learning-based continuous integration testing,"*IEEE Access.*, vol. 9, pp 36674-36688, 2021.

[25] M. Bagherzadeh and N. Kahani and L. C. Briand, "Reinforcement Learning for Test Case Prioritization,"*CoRR.*, 2020.

[26] D. Marijan and A. Gotlieb and S. Sen, "Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study,"*IEEE Computer Society.*, pp 540–543, 2013.