Reinforcement Learning Based Test Case Prioritization for Enhancing the Security of Software

Tingting Shi
Xiamen University of Technology
Xiamen, China
timmyshi@163.com

Lei Xiao
Xiamen University of Technology, Shanghai University
Xiamen, China
lxiao@xmut.edu.cn

Keshou Wu
Xiamen University of Technology
Xiamen, China
kswu@xmut.edu.cn

***Abstract*—In order to enhance the security of software, each system update needs to perform regression test. Regression testing in a continuous integration environment requires test cases to meet the needs of rapid feedback. Therefore, it is necessary to enable test cases to be effectively sorted within a certain time range so that more failure data could be discovered and the fault detection rate of testing could be improved. Reinforcement learning algorithms interact with the environment, so it is viable to optimize the sorting problem of test case in the process of continuous integration through a reward mechanism. In the development environment of continuous integration, it has been experimentally proven that the execution history of test cases in the last four cycles has a greater impact on the sorting of test cases in the current cycle. Therefore, a new RHE reward function was put forward by using part of weighted information obtained from historical execution result for enhancing the security of the system. Taking the influence of execution time into account, the multi-target sequencing technology for test case is employed with a view to improving the efficiency of defect discovery. It has been found by applying this sorting method to three industrial testing research that: (1) compared with weighted reward function based on the entire historical execution information, the function based on the four historical execution information had a higher capability of detecting faults; (2) The reward function obtained from the weighted historical results could effectively improve the fault detection rate and reduce the time consumed. (3) The multi-objective sorting methods taking execution time into consideration was able to maximize the number of testing cases that had already discovered faults within the available time.**

***Index Terms*—continuous integration, reinforcement learning, fault detection rate, reward function, historical execution information, multi-objective sorting**

I. Introduction

With the continuous development of the Internet economy such as big data and cloud computing, the scale and complexity of software development has been continuously expanding and growing, resulting in rapid iteration of software products, the security of the system needs to be continuously enhanced, and increasing pressure on regression testing. The defective testing cases could be detected as early as possible by regression testing in a way of selecting and executing related test cases. In order to improve testing efficiency and satisfy testing requirements, test case prioritization [1] was proposed to optimize regression testing. Continuous integration [2](CI) played a key part in ensuring rapid product iteration and maintaining product quality at the same time. Continuous integration was considered as a cost-effective practice of software development. In a continuous integration development environment, developers could frequently (at least one integration daily) submit modified code to a single code base and then perform adaptive build of code that merged into trunk code, including version control, compilation, and adaptive regression testing, so as to quickly find errors and prevent branches from significantly deviating from the trunk, as well as allowing products to iterate quickly and maintain high quality. In order to support frequent code integration and improve the efficiency of continuous integration testing, traditional methods of optimizing test case, including test case selection [2] and test case prioritization, could be utilized to detect failed test cases as early as possible.

Test case selection and prioritization play a crucial role in continuous integration, because it is difficult to run all test cases within a limited time. After all, it will take a large amount of time to execute so many test cases. For example, Google's automated testing platform performs an average of 150 million test operations per day [3], but defects could be detected by only a small part of test cases, resulting in a huge waste of computing resources. Therefore, it is necessary to prioritize and select test cases that had such a superior capability to detect defects than others and reduce the resource requirements by continuous integration without affecting the test quality as much as possible [2]. However, as the software evolves frequently, the number of test cases increases as well, which may cause test cases that can previously detect defects not to be able to detect defects in subsequent cycles, and the degree of defect detection for these test cases to be changed. It is hoped that more test cases capable of detecting defects should be executed. In order to solve the problems that the original deterministic methods of test cases are not suitable for the current usage scenarios, reinforcement learning is employed in continuous integration test case optimization to find the dynamic changing rules of defect detection degree of test case , so that the defect detection de-

978-1-7281-8206-3/20/$31.00 ©2020 IEEE
DOI 10.1109/DSAA49011.2020.00076

gree of test cases can be adaptively found and efficiency could be gradually improved.

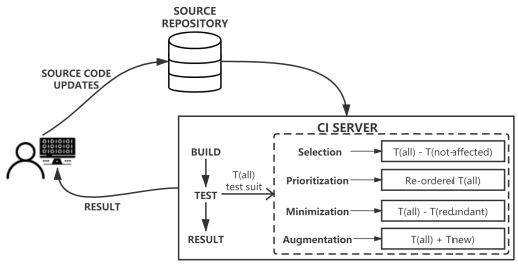


Fig. 1: Continuous Integration Process [4].

Reinforcement learning, as an important branch of machine learning, is a cyclic process in which an agent takes certain action to change its state, obtain a reward, and interact with the environment. In continuous integration testing, test case selection and prioritization could be regarded as sequential decision [5] problems faced by test cases on the basis of software code integration. It was proposed by people that in the continuous integration environment, reinforcement learning was utilized to select and prioritize test cases according to the characteristics of integration code. In 2017, RETECS (reinforced test case selection) method was proposed by Spieker et al. [6] who took the lead in applying reinforcement learning to continuous integration testing. The duration of test cases, the last execution time, and the historical records of failures were regarded as the status of a single test case. According to the policy, the agent would select an action and return feedback to itself after receiving an instant reward. Based on rewards obtained, the policy modified the next action so as to automatically learn the ability of testing cases to detect faults, and also increase the priority of discovering wrong testing cases.

Reinforcement learning could reflect the interaction between the agent and the environment by means of reward functions. A good reward function helps to perceive environment condition and choose favorable actions, thereby accelerating the convergence of reinforcement learning. However, the tcfail [6] reward function mentioned in the RETECS method only rewarded wrong test cases detected in the current integration cycle, with no reward being given to test cases that were inherently flawed, which did not take the impact of historical execution information into account. In the process of test case prioritization, RETECS just sorted according to the predicted probability, but it was hoped that more test cases were able to be executed within the time limit. As a result, overall consideration should be given to the historical execution results of test cases in continuous integration and the execution time of test cases. On the basis of different reward strategies [5] proposed by He Liuliu, a reward function based on weighted historical execution information and an execution-time-bearing multi-target ranking method of test cases were proposed.

The main contributions of this article included:

(1) Based on historical execution information with different length, the weighted rewards were given to four historical execution results and all historical results, thereby the weighted reward functions based on different length of historical result were proposed.

(2) The experimental results showed that the reward functions weighing part of the historical results in reinforcement learning effectively improved the capability of detecting faults in the integration test without increasing the time consumed.

(3) On the basis of the sequencing technology for test cases and consideration of the limit of testing time, a multi-objective sequencing technology was put forward based on execution time which could maximize the number of test cases executed in the available time and increase the number of defects detected.

Paper Outline. The chapters of this article are arranged as follows: Section 2 presents the reward function obtained from the execution results of the weighted history and multi-objective sorting technology. Section 3 applies the method proposed to different data sets for the purpose of comparative analysis. Section 4 introduces the related work and section 5 summarizes the article.

II. Reward Function Based on Weighted Historical Execution Information of Test Cases

In the reward function describing the sorting and selection of test cases by the reinforcement learning method in a continuous integration development environment, either only the execution results were considered in the current integration cycle, or the historical execution results were considered in the entire integration cycle, both of which did not take account of how much historical result information was required to calculate the reward value. Based on the above conditions, reward functions obtained from four historical records and all historical records were proposed and different weights targeted for the execution results of different cycles were given as well. Further, reward functions based on weighted history execution information was put forward.

In the testing process of continuous integration, test cases that had detected faults recently were very likely to detect errors again in the subsequent tests. Therefore, when the reward function was established, a relatively high reward should be given to the test cases that failed recently so as to these cases were able to be executed rapidly in the next integration cycle. The test cases that had detected faults in recent cycle were utilized in reward function got from RETECS method and given single reward, which did not take account of the interval between error detecting cycle and current execution cycle. In consequence, a reward method based on weighted historical execution information of test cases was proposed.

T_c was used to express a set of n test cases $t_1, t_2, ..., t_n$in a CI cycle c, which could be evolved from one cycle to another. Some of these test cases that were selected for execution and detected failures are called TE_c^{fail}. The execution result of each test case in the T_c was recorded as $ER_c = [ER_1, ER_2, ..., ER_n]$ when each case ran to the last

m calculations of C cycle, where $ER_i = [er_1, er_2, ..., er_n]$. A matrix MT with size of $m \times n$ was constructed on the basis of T_c and ER_c.

$$MT_{m,n} = \begin{pmatrix} er_{1,1} & er_{1,2} & \cdots & er_{1,n} \\ er_{2,1} & er_{2,2} & \cdots & er_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ er_{n,1} & er_{n,2} & \cdots & er_{n,n} \end{pmatrix}$$

$$MT[i,j] = \begin{cases} 0 & \text{if } t_j \text{ passed in } (cycle - i) \text{ execution} \\ 1 & \text{otherwise} \end{cases} \quad (1)$$

The execution result of each test case in each integration cycle was reflected in the values of the matrix containing n test cases. Specifically, if the test cases were executed successfully, $MT_{i,j} = 0$; If faults were detected or the test cases were not executed, $MT_{i,j} = 1$. The information of position where faults were detected was crucial in the historical execution records of test cases, so i was expressed as the distance between the initial position of test cases and current position of test cases after being executed for m times ($i = 0$ before any executions, $i = m$ after executions of m times). The distance setting weight function $w(x)$ represented the steps of current execution test after the faults were detected from test cases. According to the calculation of weighted heuristic algorithm, the reward value of each test case in the integration period of c could be obtained:

$$Reward_c(t_j) = \sum_{i=1,\cdots,m} MT[i,j] \times w(i) \quad (2)$$

According to positional distribution of the failed test cases, different weight was set for comparison experiments. The results gained from recent executions of test cases were particularly important for the subsequent tests. The experiments conducted by Spiker et.al has demonstrated that the testing efficiency could be effectively improved when the data containing execution results of four historical length were selected. As a result, the data of four historical length were selected to compare with all historical execution information. According to different reward strategies proposed by He Liuliu, the setting of reward function was divided into overall rewards and partial rewards.

A. ReLu-Weighted Historical Execution Information Reward (RHE Reward for Short) Based on Test Cases

The Relu function was used as the weight function $w(x) = relu(x) = max(0, x), (x \geqslant 0)$. In order to calculate the reward value of each test case, it is necessary to record the execution status ER_c of test case in every cycle with an integration cycle from t to c. The reward value of test cases could be obtained by adding up the product of periodic cycle order regraded as weight and execution results of each integration cycle. In a similar integration cycle, the larger the integration order, the greater the corresponding weight, and the greater the reward value of test cases failed in the similar cycle, which would be conducive to improving the fault detection rate in the next integration cycle. The calculation formula was as follows:

$$RHE_c(t_j) = \sum_{i=1,\cdots,m} MT[i,j] \times relu(i) \quad (3)$$

RHE overall rewards: (m is expressed as the number of historical records selected.)

- The whole history records were selected as the overall rewards of RHE, $i.e. m = all$ execution results.

$$Reward_c^{RHE\_whole\_all}(t_j) = RHE_c(t_j) \quad (4)$$

- The latest four history execution results were chosen as the overall rewards of RHE, $i.e. m = 4$.

$$Reward_c^{RHE\_whole\_four}(t_j) = RHE_c(t_j) \quad (5)$$

RHE partial rewards:

- The whole history records were selected as the overall rewards of RHE, $i.e. m = all$ execution results.

$$Reward_c^{RHE\_part\_all}(t_j) = \begin{cases} RHE_c(t_j) & \text{if } t \in TE_c^{fail} \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

- The latest four history execution results were chosen as the overall rewards of RHE, $i.e. m = 4$.

$$Reward_c^{RHE\_part\_four}(t_j) = \begin{cases} RHE_c(t_j) & \text{if } t \in TE_c^{fail} \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

For example, there were three testing cases in the integration cycle of c, with individual historical execution information of $ER_1 = [1,1,0,0,1,1], ER_2 = [1,0,0,0,1,0], ER_3 = [0,0,1,0,1,1]$ respectively, thereby the MT matrix was constructed as follows:

$$MT_{6,3} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix}$$

$$Reward_c^{RHE\_whole\_all}(t_1) = 1 \times relu(1) + 1 \times relu(2) + 0 \times relu(3) + 0 \times relu(4) + 1 \times relu(5) + 1 \times relu(6)$$

$$Reward_c^{RHE\_whole\_four}(t_3) = 1 \times relu(1) + 0 \times relu(2) + 1 \times relu(3) + 1 \times relu(4)$$

As for the overall rewards, the rewards were given to all testing cases in the current integration cycle, while they were only given to those testing cases that failed in the current integration cycle for partial rewards, with zero reward being given to the rest of cases. Selecting all historical execution records meant that all execution results of test cases were included and counted in the entire continuous integration test, while choosing the latest four execution results meant that

integration testing results obtained from the current cycle and the previous three cycles were taken into account and used as input for the iterations. Similarly, other formulas could be obtained.

B. Reward Function Based on Tanh Weighted Historical Execution Information of Test Cases (THE Reward for Short)

The tanh function served as weight function $w(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}, (0 \leqslant x \leqslant 4)$. It could be known from the graph of the tanh function [7] that when the value of x was greater than or equal to 4, the function value was infinitely close to 1. As a result, the values of $\tanh(x)$ function with the domain of definition between 0 and 4 were selected as the weighted values. The data with a historical length of 4 were selected as the input of the reward function, and the execution result in the latest integration cycle was given a larger weight value, thus a larger reward value could be obtained.

$$THE_c(t_j) = \sum_{i=1,\cdots,m} MT[i,j] \times tanh(i) \tag{8}$$

THE overall reward:

$$Reward_c^{THE\_whole\_four}(t_j) = THE_c(t_j) \tag{9}$$

THE partial reward:

$$Reward_c^{THE\_part\_four}(t_j) = \begin{cases} THE_c(t_j) & \text{if } t \in TE_c^{fail} \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

According to different reward strategies, the overall reward function of THE and the partial reward function of THE were proposed, and four historical execution results were selected for the calculation of reward value.

III. Reinforcement Learning Based on Priority Selection of Multi-Objective Test Cases

As for prioritization technology for testing cases, the coverage and complexity of code and historical defect information of test cases played a vital role in the ranking process. In the process of sorting test cases with reinforcement learning, only the historical execution results of test cases were considered. However, the single-target priority-based ranking technology had certain one-sidedness. As a result, multi-objective priority method targeted at test cases was utilized to in this paper after incorporating influencing factors of testing execution time and comprehensively taking the execution time and selection probability of tests.

Considering the execution order of test cases with multiple optimized objectives in the continuous integration testing could improve the testing efficiency of software. It was hoped that the test cases with high capability to detect faults and the shortest execution time were preferred to be executed, therefore there was also a need to consider the execution time of test cases during sequencing and selecting for the purpose of more test cases being executed within the limited time. As each test case was likely to detect faults, the consideration mentioned above could help to improve the ability to find faults.

The original calculated test cases were sorted in descending order of priority. The test cases with the same priority were classified into one category, and all cases were divided into s categories with interval of 1 between two consecutive categories. Although the test cases in the same category had the same priority, their execution time was different, so it was necessary to recalculate the priority based on the execution time. Assuming that the execution time of each test case in a set of test cases was $Te = \{te_1, te_2, \cdots, te_n\}$, and T_{max} was the maximum time available to be used by the test cases in the cycle. If the execution time of a certain test case was longer than the overall available time, the priority of this case was set to $P = 0$ and the execution time of the rest of test cases was normalized to $[0, 1]$. Consequently, the following algorithm was used to increase the priority of test cases:

$$p_i = \begin{cases} s + 1 & te_i \geqslant T_{max} \\ p_i + (1 - \frac{te_i}{T_{max}}) & \text{otherwise} \end{cases} \tag{11}$$

IV. Experiment Analysis

The RETECS method that firstly applied reinforcement learning to the prioritization problem of test cases was proposed in 2017 by Spiker et al. who regarded the execution time required by test cases, the last execution time, as well as the historical execution results as the state of reinforcement learning. The action referred to the priority of test cases in the current integration cycle. Before the testing cases were selected to be executed and rewarded on the basis of execution results, all test cases were sorted in the descending order according to the priority, after which the rewards were given back to the agent for priority predicting in the next cycle. Therefore, the fault detection rate of test cases could be improved gradually. The flow chart of the RETECS method was shown in Fig. 2.

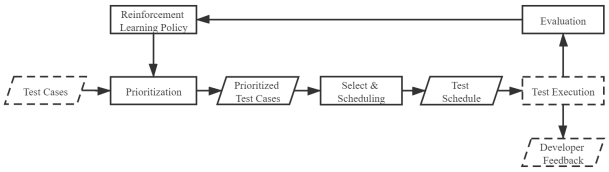


Fig. 2: Testing in a continuous integration environment [6].

In this chapter, the RETECS method proposed by Spiker et al. was employed to verify the pros and cons of the proposed reward function based on the weighted test execution time and the original tcfail reward function. The following questions were mainly raised.

RQ1: Which kinds of rewards functions that included the execution results with history length of four and all history records could improve the error detection rate.

RQ2: As for the reward function based on the weighted historical execution results, how to design the weight would be conducive to quickly detecting the failed test cases.

RQ3: What impacts would be caused by applying the weighted historical record information to the overall reward and the partial rewards respectively.

RQ4: What effect would be imposed on the consumption of execution time by the reward function based on the weighted historical execution results.

RQ5: Whether Multi-objective sorting that considered execution time when sorting test cases would increase the executing number of test cases or not.

A. Experimental Parameters

In the experiment, the artificial neural network was selected as the agent for reinforcement learning, and according to the experiment of Spieker et al. [6] the time threshold was set to 50% of the total execution time of all test cases to obtain relatively high efficiency. In addition, the average value was chosen as the final result after the experiment had been repeated for 30 times.

B. Evaluation Indicators

APFD, as a currently mainstream evaluation index, was first proposed by Rothermal [8]et al. Given the execution order of test cases, this index could give the average cumulative ratio of defects detected during the execution of test cases. The higher the value, the quicker the detection rate. Elbaum [8] gave the calculation formula of APFD. More specific, for a given set of test cases T, there were n test cases and m defects. TF_i represented the order of test case that was the first to have detected the number i defect during the process of execution [9].

$$APFD = 1 - \frac{TF_1 + TF_2 + \cdots + TF_n}{nm} + \frac{1}{2n} \tag{12}$$

Normalized APFD [10] (NAPFD) was an extension of APFD, including the ratio of defects that had been detected in the testing set to the defects that were likely to be detected. Therefore, it was applicable to task selection of test cases, while not all test cases were executed and faults could be detected. If all failures were detected $(p = 1)$, NAPFD was equal to the original formula of APFD.

$$NAPFD(TS_i) = p - \frac{\sum_{t \in TS_i^{fail}} rank(t)}{|TS_i^{fail}| \times |TS_i|} + \frac{p}{2 \times |TS_i|} \tag{13}$$

$$with\ p = \frac{TS_i^{fail}}{TS_i^{total,fail}}$$

Where $|TS_i|$ was used to express the total number of test cases executed in the cycle i, $|TS_i^{fail}|$ was represent the number of failed test cases in TS_i, $|TS_i^{total,fail}|$ indicate the number of all failed test cases in the test set, $rank(t)$ means the position of the t-th failed test case in TS_i. The value of NAPFD was used in this paper as the evaluation index for this experiment, $i.e.$ p was not equal to 1.

C. Experimental Design

In order to solve the question of RQ1, the latest four historical execution results and all historical execution results were selected respectively in this paper for comparison analysis in the process of designing reward function. As for the RQ2, the relu function and the tanh function were selected respectively as the weighted function for further comparison, thus the weight setting that resulted in a higher ability to detect errors could be obtained. For the purpose of solving RQ3, both overall rewards and partial reward were set respectively to determine which reward strategy had a higher capability of detecting faults. RQ4 was mainly to verify the ability of different reward functions to consume time. In addition, the purpose of RQ5 was to determine whether the multi-objective sorting method could effectively increase the error detection number of test cases. In this paper, the reward function was mainly compared with tcfail reward function in the RETEGS method, and the formula was as follows:

$$Reward_c^{tcfail} = \begin{cases} 1 & \text{if } t \in TE_c^{fail} \\ 0 & \text{otherwise} \end{cases} \tag{14}$$

D. Experimental Subject

In order to determine this method had the practical applicability in the industry, the ABB Robotics Norway2 was utilized in this paper to test the industrial data sets of *Paint Control* and *IOF/ROL* for the complex industrial robots, and the Google Open Source Data Set [11] (GSDTSR). These data sets contained the historical execution results and execution time of test cases for more than 300 integration cycles. The basic structure of three industrial data sets were listed in Table. I, including the number of test cases, the number of integration cycles, execution results, and failure rates of each data set.

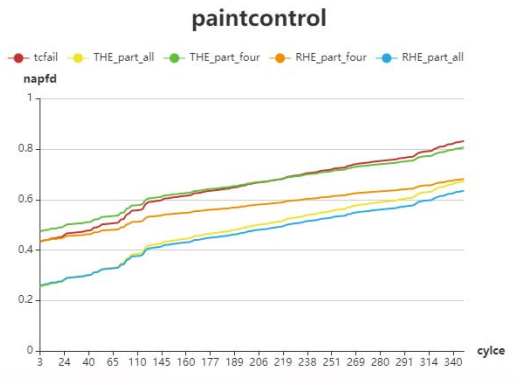
TABLE I: BASIC STRUCTURE OF THREE INDUSTRIAL DATA SETS [6]

Data Set	Test Case	CI Cycles	Verdicts	Failed
Paint Control	114	312	25,594	19.36%
IOF/ROL	2,086	320	30,319	28.43%
GSDTSR	5,555	336	1,260,617	0.25%

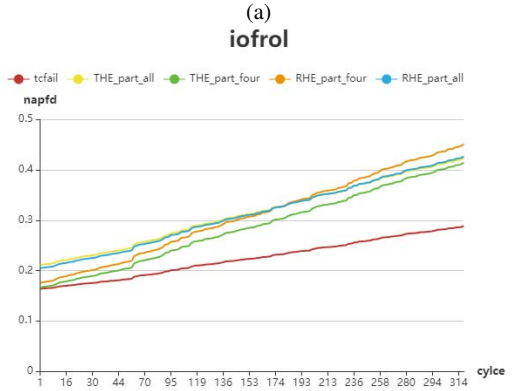
E. Experimental Results and Analysis

For the purpose of better comparison, the RETECS method was reproduced. The Fig. 3 and Fig. 4 showed the experimental results obtained by different reward functions, such as tcfail, THE, RHE, with the abscissa denoting the number of cycles in the integration cycle and the ordinate representing the value of NAPFD. What was more, the fitted graph was adopted to represent the trends of testing results.

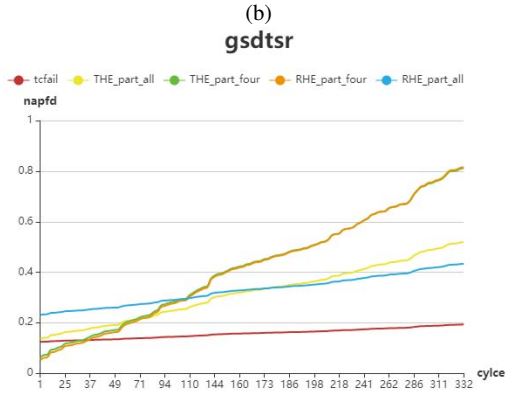
a) Analysis of RQ1: The reward function used in the RETECS method was the partial reward strategy, thereby the comparison experiments targeted at partial reward strategy were performed. The NAPFD fitting curves obtained by executing different reward functions on the data sets with



(a)

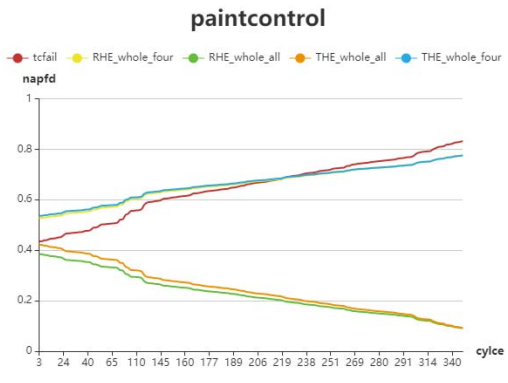


(b)

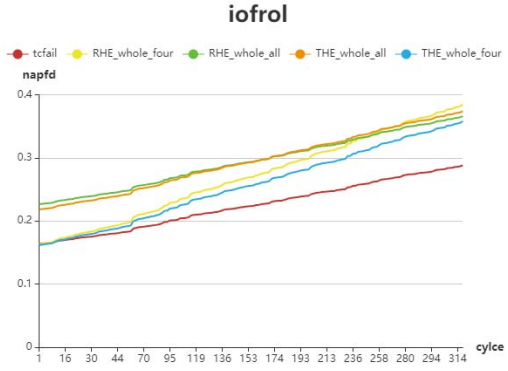


(c)

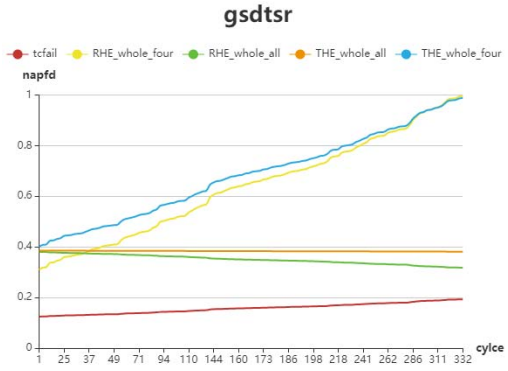
Fig. 3: Napfd under different reward functions in partial rewards.



(a)



(b)



(c)

Fig. 4: Napfd under different reward functions in overall rewards.

different size under the guidance of partial reward strategy was shown in Fig. 3 respectively, where THE and RHE contained four historical records and all historical execution results respectively. The results demonstrated that as for data sets of paint control, It was found that on the paint control data set, the difference between the reward function value of RHE_part_four and tcfail was about 0.15 and the reward function value of RHE_part_all was about 0.2 lower than that of tcfail, while the detection rate of THE_part_four in the first 180 cycles was about 0.05 higher than that of tcfail, and the detection rate of THE_part_all was approximately 0.2 lower than that of tcfail. In terms of infrol, the four reward functions of RHE and THE were at least 0.1 higher than tcfail, indicating that when the failure rate was high, the reward function containing the historical record could effectively detect errors. In the first 170 cycles, RHE_part_all had a slightly higher detection capability than RHE_part_four. In the later period, the detection rate of RHE_part_four reward could reach 0.45. However, the detection capability of THE_part_four is inferior to that of THE_part_all in all cycles. On the google dataset, the

detection rate of tcfail was about 0.2 and that of RHE_part_all and THE_part_all exceeded 0.4 and 0.5 respectively, while the RHE_part_fourand THE_part_four showed the comparable detection ability of 0.8, which indicated that the reward function with four historical records was superior to the reward function without history records or the reward function with all history records.

In summary, in the data set of infrol and gsdtsr, the reward function containing execution results with the historical length of four could effectively improve the error detection rate compared with that of all historical records, because some test cases that failed in the early stage had been repaired and these out-of-date test cases played a marginal impact on the following tests.

Fig. 5: The differences in the four historical execution results between the reward function and tcfail.

b) Analysis of RQ2: A reward function based on four historical execution results was used for comparative analysis of different weights, and the partial reward strategy was adopted. As shown in Fig. 3, the reward function of THE_part_four had a detection rate of 0.8 on the paint cotrol data set, which was approximately equal to that of tcfail, while that of RHE_part_four was about 0.7. As illustrated in Fig. 5, it was clearly demonstrated by the results of infrol data set that the RHE_part_four showed a higher detection ability than the THE_part_four, while In the Google data set, they showed the comparable detection ability. In general, when the failure rate was not high but the data size was very large, using the reward function of the weighted historical result recording of tanh and relu could effectively improve the error detection ability of the test cases. When the error rate was high and the data size was large, using the relu could be helpful for detecting failed test cases much faster.

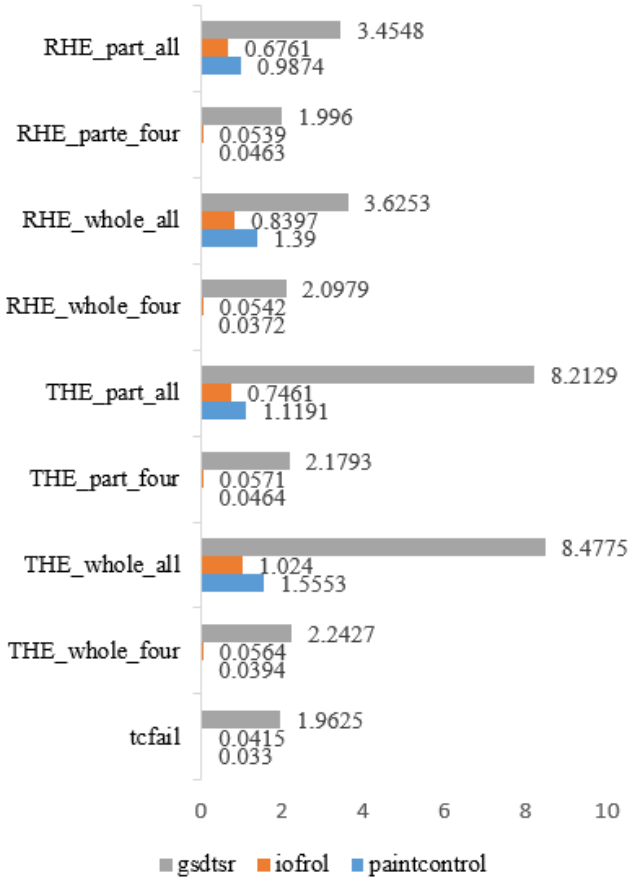


Fig. 6: Time spent in different reward functions (in seconds).

c) Analysis of RQ3: As shown in Fig. 5 and Fig. 6, there were no significant differences in detection rates between the THE_whole_all reward and the RHE_whole_all reward in these three data sets, but they were ineffective in the paint control data set. However, the overall rewards of THE and RHE based on the four historical results played a minor effect on the NAPFD value tested. On the whole, the weighted reward function of the overall reward based on the four historical results had a better effect than that based on the entire historical records. In addition, there was no significant difference between the results of overall rewards and that of

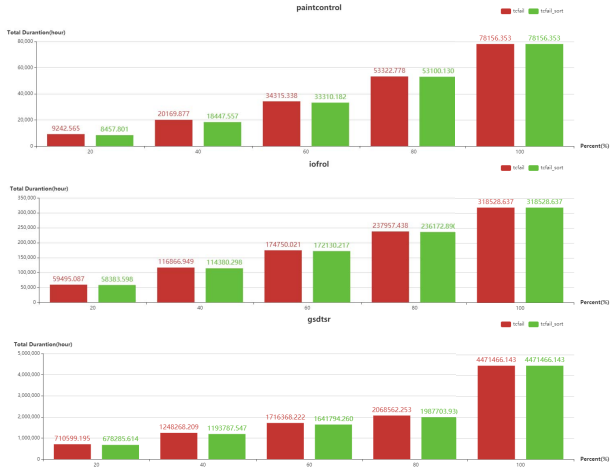
partial rewards when the relu function and tanh function were applied to the weighted reward function.

d) Analysis of RQ4: In this part, the conditions of time consumed were mainly compared among different reward functions. In the experiment, the total time required by selecting and sorting was calculated in each integration cycle, and the experiments were repeated for 30 times. The average value of results obtained from 30 experiments was selected for further comparison, with unit of seconds. As shown in Fig. 6, it could be seen that in the reward functions based on the same history length, the number of test cases for rewards was relatively large, resulting in the overall rewards consuming more time than the partial rewards. However, under the condition of the same reward strategy, the historical information required to be calculated by reward functions based on four historical length was relatively less, leading to the time required by reward functions that contained all history records shorter. There were no significant differences in the time spent by reward functions that were weighted by using relu function and tanh function on the data sets of paintcontrol and infrol, but the difference between the time spent on the paintcontrol and infrol data sets and the time spent on Google shared data set was approximately 0.2. The scale of the test cases in the Google data set was relatively large, and the number of test execution results was also the largest, thus the corresponding time consumed was relatively high. In general, the time consumed by the weighted reward function based on the four historical lengths was roughly equivalent to the time spent by the tcfail reward function. However, in an industrial production environment, it is necessary to reduce the time overhead.

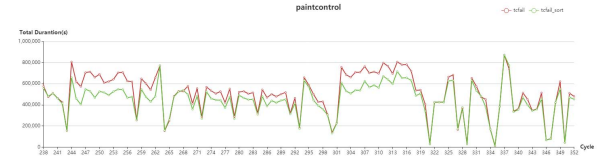
e) Analysis of RQ5: In order to determine the detection capability of the reinforcement learning using multi-objective ranking method, the single variable method was used. The original tcfail was used in design of the reward function, while the contrast experiments were conducted between the multi-objective method and the original RETECS single-objective method in terms of sorting methods Fig. 7 (a) demonstrated the comparison of the execution time required to execute different numbers of test cases on different data sets. In order to visually show the change in test execution time, 40% test cases that had been executed in each cycle were selected and tested to gain the execution time required. The results showed that the execution time required by the multi-objective ranking method was shorter than that needed by the single-objective method, meaning that the more test cases could be executed using multi-objective methods within the same time, thereby improving the overall error detection capability.

V. Related Work

The problem related to the priority of test case was first proposed by Wong et al. [1] In 1997. The detection rate of regression tests was able to be improved by sorting the test cases according to the coverage ability of single test case and executing them in sequence. After that the specific description of TCP issue was given by Elbaum and Rothermel



(a) Comparison of test time required by data of different scales



(b) the execution time required by executing 40% test cases with paintcontrol data sets

Fig. 7: Comparison of test time for different sorting methods.

[8], i.e. the code coverage, code complexity and historical defect information of test cases were needed to considered in the process of sorting.

Consideration from the perspective of demand could provide guidance for sequencing test cases. In combination with demands, error feedback and historical information, Wang Xiaolin [12] concluded that the defect detection rate of test cases could be improved in an agile development environment. The test case priority based on code coverage was first used by Yoo et al. [13] to select the test cases. Yoo et al. [14] also quoted expert knowledge in the clustering algorithm. More specific, the test cases were sorted in intra-cluster according to code coverage, and they were selected in inter-cluster using expert knowledge, which leaded to robust results. Leon et al. [15] and Carlson [16] also conducted similar analysis using code coverage information.

Machine learning has increasingly wide applications in the field of software testing. Busjaeger and Xie et al. [17] gave top priority to test cases in the industrial environment by using machine learning and a variety of heuristics. Chen et al. [18] proposed a prioritization technique based on the execution time of test and coverage range per unit time. The prediction model was trained by using XGBoost algorithm [19], and the results showed that XGBoost's performance was superior to the other ten typical classification algorithms, including Multivariate Bernoulli Naive Bayes [20] (BNB), Multilayer Perceptron [21] (MLP), Support Vector Machine [22] (SVM), Logistic Regression [22] (LR), Ridge Regression [23] (RR),

K-Nearest Neighbors [24] (KNN), Random Forest [25] (RF) , Extra-Trees [26] (ET), Linear Discriminant Analysis [27] (LDA) and Quadratic Discriminant Analysis [28] (QDA). Zhao et al. [29] combined a code-coverage-based clustering method with the Bayesian-based method, which resulted in errors being detected more quickly than the results obtained by using the single Bayesian-based method.

Taking the testing resources and execution time into consideration in a continuous integration environment, Kim and Porter et al. [30] assigned the different weights to the observation values for every test cases at the stage of testing, performed weighted summation, and executed the test cases according to the sequence after cases being sorted. Marijan et al. [31] performed the weighted analysis of historical data that filed, and it was concluded that test cases with short execution time were preferred to be executed, thus more faults could be detected within a limited time. Elbaum et al. [32] performed research on the Google's large data sets and defined a time window to show the execution conditions of test cases. On the basis of diversity (DBTP) and historical problems related to testing priority, Haghighatkhah et al. [33] employed a technique similar to the clustering-based technique proposed by Hemmati et al. [34] and drew the conclusion that good results could be gained without needing large quantities of historical data.

The person who first combined reinforcement learning with continuous integration was Spieker et al. [6] The RETECS method they proposed in 2017 was to reward the prioritization results after they were executed in the previous integration cycle. The reward value given to the wrong test cases was 1, while reward value assigned to the remaining test cases was 0. Then the artificial neural network method was used to train and to adaptively continue optimizing and selecting the test cases that may failed with a higher possibility for further execution. On the basis of RETECS, He Liu Liu [5] proposed different reward strategies, and considered all historical failure results and the distribution of failure in the design of award functions.

VI. Conclusion

This paper focused on the application of reinforcement learning to the design of reward function in a continuous integration environment. Based on the weighted history execution results obtained in different ways, two types of reward functions were proposed, with one based on the weighted historical results of relu reward function and the other one based on the weighted historical results tanh reward function. According to different reward strategies and the execution results of different historical length, eight different reward functions were utilized for comparative studies, which included the overall reward strategies, the partial reward strategies, execution results of cases with historical length of four, and all historical execution results The experiment results illustrated that in three industrial data sets with different scale and failure rate, the reward functions based on the weighted history results could effectively improve the capability of test cases to detect errors. The reward functions containing execution results of four historical length had a stronger ability of detection and required shorter execution time than the reward functions that contained all historical records. This is because some test cases that failed previously were repaired after a certain period of time, and there was no need to consider these repaired test cases in the following tests.

When the data size was large and the error rate was not very high, the weighted method based on the rule function would be favorable for quicker detection of the failed test cases than that based on the tanh function As for the reward function of the THE and RHE proposed in this paper, there was no significant difference in the detection efficiency between the overall reward and partial reward strategies. However, the time spent by executing functions based on the overall rewards was relatively longer than that consumed by the partial rewards, thereby it was recommended that the partial rewards were applied in the experiments.

In the comparative experiments of reinforcement learning using multi-objective ranking method, it could be intuitively seen that under the condition of the same execution time, the multi-objective ranking method was able to execute more test cases, thus improving the error detection rate and reducing time cost.

In the sorting process, the execution time was mainly considered. However, in practice, there were many factors influencing sorting. In the future work, it is expected that the sorting targets will be optimized by considering multiple factors, and the agent algorithm is modified by using some deep learning methods with a view to improving fault detection rate.

References

[1] Wong W E, Horgan J R, London S, et al. A study of effective regression testing in practice[C]. PROCEEDINGS The Eighth International Symposium On Software Reliability Engineering, 1997: 264-274.

[2] Li YL, Wang Q. Test Set Optimization in Continuous Integration:A Systematic Literature Review. Journal of Software, 2018, 29(10): 3021-3050(in Chinese).http://www.jos.org.cn/1000-9825/5613.htm.

[3] Memon, Atif & Gao, Zebao & Nguyen, Bao & Dhanda, Sanjeev & Nickell, Eric & Siemborski, Rob & Micco, John. (2017). Taming Google-Scale Continuous Testing. 233-242. 10.1109/ICSE-SEIP.2017.16.

[4] Alireza Haghighatkhah, Mika M¬ antyl ¬ a, Markku Oivo, Pasi Kuvaja, Test Prioritization in Continuous Integration Environments, The Journal of Systems & Software (2018), doi:https://doi.org/10.1016/j.jss.2018.08.061

[5] He LL, Yang Y, Li Z, Zhao RL. Reward of Reinforcement Learning of Test Optimization for Continuous Integration. Journal of Software, 2019, 30(5): 1438-1449(in Chinese).http://www.jos.org.cn/1000-9825/5714.htm.

[6] Spieker H, Gotlieb A, Marijan D, et al. Reinforcement Learning for Automatic Test Case Prioritization and Selection in Continuous Integration[C]. Acm Sigsoft International Symposium on Software Testing & Analysis, 2017.

[7] Science - Physical Science; Researchers from Zhejiang Science Technical University Detail Findings in Physical Science (The Residual Symmetry and Consistent Tanh Expansion for the Benney System) %J Science Letter[J], 2017.

[8] Elbaum S, Malishevsky A G, Rothermel G. Prioritizing test cases for regression testing[M]. 25. ACM, 2000.

[9] Chen X, Chen JH, Ju XL, Gu Q. Survey of test case prioritization techniques for regression testing. Ruan Jian Xue Bao/Journal of Software, 2013,24(8):1695-1712 (in Chinese). http://www.jos.org.cn/1000-9825/4420.htm.

[10] Qu X, Cohen M B, Woolf K M. Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization[C]. IEEE International Conference on Software Maintenance, 2007.
[11] Duvall M P, Matyas S, Glover A. Continuous Integration: Improving Software Quality and Reducing Risk[M]. 2007.
[12] Wang XL, Zeng HW, Lin WW. Techniques for Regression Testing in Agile Development Environment %J Chinese Journal of Computers [J], 2019: 1-15.
[13] Yoo S, Harman M. Pareto efficient multi-objective test case selection[C]. Proceedings of the 2007 international symposium on Software testing and analysis, 2007: 140-150.
[14] Yoo S, Harman M, Tonella P, et al. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge[C]. Proceedings of the eighteenth international symposium on Software testing and analysis, 2009: 201-212.
[15] Leon D, Podgurski A. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases[C]. 14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003., 2003: 442-453.
[16] Carlson R, Do H, Denton A. A clustering approach to improving test case prioritization: An industrial case study[C]. 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011: 382-391.
[17] Busjaeger B, Xie T. Learning for test prioritization: an industrial case study[C]. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2016: 975-980.
[18] Chen J, Lou Y, Zhang L, et al. Optimizing test prioritization via test distribution analysis[C]. Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018: 656-667.
[19] Chen T, Guestrin C. XGBoost: A Scalable Tree Boosting System[C]. Acm Sigkdd International Conference on Knowledge Discovery & Data Mining, 2016.
[20] Peng F, Schuurmans D. Combining Naive Bayes and ¡Emphasis Type=”Italic”¿n-Gram Language Models for Text Classification[M]. 2003.
[21] Hinton G E J a I. Connectionist learning procedures[J], 1989, 40(1): 185-234.
[22] Volokh A, Neumann G J S P O I W O S E. 372:Comparing the Benefit of Different Dependency Parsers for Textual Entailment Using Syntactic Constraints Only[J], 2010: 308-312.
[23] Hoerl A E, Kennard R W J T. Ridge Regression: Biased Estimation for Nonorthogonal Problems[J], 2000, 12(1): 55-67.
[24] Altman N S J a S. An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression[J], 1992, 46(3): 175-185.
[25] Camgöz N C, Kindiroglu A A, Akarun L. Gesture Recognition Using Template Based Random Forest Classifiers[C]. European Conference on Computer Vision, 2015.
[26] Geurts P, Ernst D, Wehenkel L J M L. Extremely randomized trees[J], 2006, 63(1): 3-42.
[27] Rayens W S J J O T R S S. Discriminant Analysis and Statistical Pattern Recognition[J], 2010, 35(3): 324-326.
[28] Hastie T, Friedman J, Tibshirani R J T. The Elements of Statistical Learning[J], 2010, 45(3): 267-268.
[29] Zhao X, Zan W, Fan X, et al. A Clustering-Bayesian Network Based Approach for Test Case Prioritization[C]. Computer Software & Applications Conference, 2015.
[30] Kim J M, Porter A A. A history-based test prioritization technique for regression testing in resource constrained environments[C]. International Conference on Software Engineering, 2002.
[31] Marijan D, Gotlieb A, Sen S. Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study[C]. IEEE International Conference on Software Maintenance, 2013.
[32] Elbaum S, Rothermel G, Penix J. Techniques for improving regression testing in continuous integration development environments[C]. Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014: 235-245.
[33] Alireza H, Mika M N, Markku O, et al. Test Prioritization in Continuous Integration Environments[J], 2018: S0164121218301730-.
[34] Hemmati H, Fang Z, Mäntylä M V, et al. Prioritizing manual test cases in rapid release environments: Prioritizing manual test cases in rapid release environments[J], 2017, 27(6).