The Journal of Systems & Software 170 (2020) 110787

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A systematic study of reward for reinforcement learning based continuous integration testing

Yang Yang, Zheng Li*, Liuliu He, Ruilian Zhao

College of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, PR China

ARTICLE INFO

Article history:
Received 16 September 2019
Received in revised form 8 August 2020
Accepted 13 August 2020
Available online 22 August 2020

MSC:
00-01
99-00

Keywords:
Continuous integration
Test case prioritization
Reinforcement learning
Reward policy

ABSTRACT

Continuous integration(CI) testing is characterized by continually changing test cases, limited execution time, and fast feedback, where the classical test prioritization approaches are no longer suitable. Based on the essence of continuous decision mechanism, reinforcement learning(RL) is suggested for prioritizing test cases in CI testing, in which the reward plays a crucial role. In this paper, we conducted a systematic study of the reward function and reward strategy in CI testing. In terms of reward function, the whole historical execution information of test cases is used with the consideration of the failure times and failure distribution. Further considering the validity of historical information, partial historical information is used by proposing a time-window based approach. In terms of reward strategy which means how to reward, three strategies are introduced, i.e., total reward, partial reward, and fuzzy reward. The empirical study is conducted on four industrial-level programs, and the results reveal that using the reward function with historical information improves the Recall by on average 13.21% when compared with existing TF(Test Case Failure) reward function, and the fuzzy reward strategy is more flexible and improve the NAPFD(Normalized Average Percentage of Faults Detected) by on average 3.43% when compared with the other two strategies.

1. Introduction

Continuous Integration (CI) is a software development practice designed for real-time deployment, which merges all developers' working copies to a shared mainline,[1] including version control, software configuration management, automated build, and regression testing (Vasilescu et al., 2015). While the released software need to remain potentially publishable, CI also requires developers to integrate their work frequently, in general, once a day, sometimes even multiple times a day. Each integration is verified by an automated build and regression testing to detect faults as quickly as possible. Compared to the traditional development pattern of integrating over a long period of time, CI has a high frequency of integration. Therefore, it is necessary to find faults and provide feedback as soon as possible, which helps developers to understand the software integration and modify the unsuccessful integration in time to improve the efficiency and quality of software development. But with scaling up the size of software and the increasing number of test cases, it is challenging for the existing prioritization techniques of regression testing to satisfy the requirement for rapid feedback in CI testing.

To improve the efficiency of regression testing in CI environment, the test cases, that can potentially detect faults, need to be executed earlier. Traditional regression testing prioritization techniques are based on a variety of code coverage metrics mainly, which is time-consuming. Zhang et al. (2009) proposed a prioritization method based on call path, Jiang et al. (2009) used the differences of the test cases to do test case prioritization(TCP), which select the least similar test case as the next to be sorted, and Memon et al. (2017) leveraged correlations among code, test cases, developers, programming languages, code changes, and test execution frequency to improve CI and development processes, which could help find more test cases that are prone to failure. However, the effectiveness of these TCP techniques depends on the program, test cases, and modifications, and TCP techniques are not applicable in different programs or modified versions (Elbaum et al., 2004), which makes traditional test methods unsuitable for TCP in CI testing.

In recent years, learning-based methods are tended to be used for solving TCP problems. Spieker et al. (2017) first applied reinforcement learning (RL) into CI test optimization in 2017, and proposed an approach for test case prioritization and selection based on RL. The continuous integration system itself is continuously modified and updated, which requires continuous testing in a fast time frame. Accordingly, the prioritization of CI test cases is a continuous decision-making process, in which the execution sequence of test cases will be reordered in each

* Corresponding author.
E-mail address: lizheng@mail.buct.edu.cn (Z. Li).

[1] http://wikipedia.moesalih.com/Continuous_integration

https://doi.org/10.1016/j.jss.2020.110787
0164-1212/© 2020 Elsevier Inc. All rights reserved.

integration, which is usually determined based on their execution results in previous combination synthesis. RL is a learning-based approach for the continuous decision problem, which can be used in TCP of CI. RL uses the learning of historical processes to develop a set of strategies to achieve maximum effectiveness. The combination of RL and CI enables it to prioritize test cases without specific system, modification and test case set, which is an adjustment framework with adaptive learning, where agent and reward are the two main components. The reward evaluates the effectiveness of a test case in previous CI integration, and the agent can determine the execution order of a test case based on the reward.

The prioritization effect of test cases based on RL depends directly on the feedback of learning results, that is, the measure of reward function. The good reward method will continuously optimize the sorting effect, while the bad reward method will continuously learn from the bad direction to reduce the sorting effect. Therefore, the reward plays a vital role to study the combination that should be designed according to the requirements of the CI testing. In this paper, the reward is systematically studied in the combination framework of RL and CI, with the consideration of what to reward and how to reward respectively.

In terms of what to reward, which is referred to as reward function, we consider to include historical information of the test case execution into the computation of reward function. We first include the whole historical information of a test case in the reward function, in which, a metric of the Average Percentage of Historical Failure (*APHF*) is proposed with the consideration of the distribution of the execution results and the execution timing of test cases in CI. Compared with the existing RETECS (Reinforced Test Case Selection) (Spieker et al., 2017) framework, where the reward function only considers the current execution information, *APHF* is more effective by using the whole history information of the test cases. However, a significant computation cost increase comes from the high frequency of CI integration and the large accumulation of historical information of the test case execution. This paper further proposes a partial historical information calculation method, which adopts the technology of time window to select the recent useful historical information for the computation of reward function, to improve the efficiency with retaining the quality.

In the view of how to reward, three reward strategies are proposed, i.e., total reward, partial reward, and adaptive fuzzy reward. In total reward strategy, all test cases are treated equally and can receive a reward, although the value for a single test case may be different based on the evaluation of reward function. However, every test case has its execution result in the current cycle, passed or failed. It is general to consider that a failed test case is more important than a passed, as it detects faults in the current cycle and then has high possibility to detect faults in the following cycles. Consequently, the partial reward strategy is then proposed, in which only the failed test cases receive a reward. A further investigation reveals that the ability to fault detection of a test case is not only correlated with the faults detected in last execution but also strongly associated with the test requirements (Campos et al., 2014). Therefore, the fuzzy reward strategy is further proposed, in which some passed test cases in the current cycle are adaptively rewarded based on their historical execution results.

The main contributions of this paper include:

- A systematic research on the reward of reinforcement learning is conducted in this paper for continuous integration testing.
- Two types of reward function are defined with the consideration of total and time-window based historical execution information of test cases.
- Three reward strategies are further proposed on the reward of reinforcement learning for continuous integration testing.
- Empirical research is carried out on real-world industrial programs to compare the variety of reward functions and reward strategies.

The paper is organized as follows: Section 2 summarizes the related work and leads to the motivation of this paper. Section 3 presents the reward with reward function and reward strategy. Section 4 empirically verifies the validity of the rewards proposed in this paper. Section 5 concludes the article.

2. Background and related work

Test Case Prioritization (TCP) is a problem of time series. During the testing, test cases are executed one by one, and the testers hope that the test case with a high probability of detecting faults could be executed preferential, to identify and fix faults earlier and reduce losses. TCP technique is used to evaluate test cases individually and reschedule them base on the evaluation. Wong et al. (1997) first proposed it in 1997 to find the optimal execution sequence of the original test cases that met the test criteria. Rothermel et al. (2000) gave a formal description of TCP in 2000 and conducted a series of empirical studies. Then a variety of TCP techniques were proposed (Yoo and Harman, 2015), including coverage-based (Chi et al., 2020), modification-based (Jahan et al., 2019), fault-based (Mahdieh et al., 2020), requirement-based (Srikanth et al., 2016), and history-based (Cho et al., 2016) TCP techniques. However, whether the Time Consumption of a test case is taken into account, the rate of code coverage and faults detection will not be changed in TCP (You et al., 2011).

Search-based approaches can solve complex prioritization problems effectively. Li et al. firstly proposed search-based TCP in software regression testing (Li et al., 2007). They further studied search algorithms for multi-objective TCP problems and used GPU parallel technology to improve its efficiency (Li et al., 2013; Bian et al., 2017). Souza et al. (2011) and Yu et al. (2010) used search-based approach to solve TCP problems based on the similarity among test cases.

In fact, the above TCP technologies do not take into account the test environment and system cycle. And the methods occupy a large amount of time overhead in configuration (Do and Rothermel, 2006). However, with the continuous integration of CI systems, there are environment and configuration changing, and the corresponding needs to add new test cases. Experiments have shown that in traditional method test suite augmentation significantly hampers their effectiveness, whereas source code changes alone do not influence their effectiveness much (Lu et al., 2016). So new approaches are needed for test prioritization in CI.

Many TCP techniques had been presented for CI testing. Marijan et al. (2013) proposed TCP techniques for CI, followed by multiple prioritization objectives and methods (Noor and Hemmati, 2016; Elbaum et al., 2014). When faced with multiple prioritization objectives, Ammar et al. (2017) prioritized test cases by adjusting weights of different prioritization objectives. Strandberg et al. (2016) analyzed and combined execution time and its related factors, such as fault detection result, a time interval of last execution, and the modification information of the code under test, and test cases were selected by assigning and merging priorities to the test case sequences. Haghighatkhah et al. (2018) sorted CI test cases based on historical information and diversity, which was proved more effective by Henard et al. (2016) with the black-box test approach. But, such approaches were based on the dynamical analysis for each CI cycle, which may cause a lot of time overhead (Luo et al., 2016). To address the problems, the combination of RL and CI (Spieker et al., 2017) was proposed to order test cases through CI learning.

RL is an essential branch of machine learning, which is often used to solve continuous decision problems. It is based on the continuous learning of the historical process, through continuous exploration to find a strategy with the greatest future expectations. Groce et al. combined RL and ABP (ASP.NET Boilerplate Project) to test software APIs (Groce et al., 2013). Reichstaller et al. applied RL to risk-based interoperability testing and conducted risk assessment through RL to achieve test case generation (Reichstaller et al., 2016).

CI testing, as a branch of regression testing, is more stringent in terms of feedback time, in addition to program complexity than regression testing. It continuously tests the submitted commits to ensure that their integration into the backbone works properly. The key to continuous software integration is that newly committed code must pass the automated regression testing before it is integrated into the backbone, that is, in each integration, it is necessary to detect whether new faults are introduced. With the continuous integration of the CI system, real-time testing is required after each integration. In order to find faults as early as possible, test cases are prioritized. After each integration of the continuous integration, corresponding test cases need to be reprioritized each time, which is a continuous decision process.

The unsupervised exploration mode of RL, through the double-feedback mechanism of observation and decision-making, enables it to quickly learn an optimized strategy. In RL, a strategy is to perform a specific step in a particular state and the agent generates a good strategy through continuous exploration and learning (Sutton and Barto, 1999). It should be noted that the agent interacts with the environment directly instead of relying on supervisory mechanisms or complete environmental models. Then RL makes decisions based on environmental feedback to maximize the benefit. Consequently, RL can be used to solve continuous decision problem, and the feedback is an important factor in decision making.

CI testing is a constant decision problem, in which test cases are sorted in each integration. When RL is applied into CI testing (Fig. 1) (Spieker et al., 2017), the state is the test cases submitted in each cycle, the action is the prioritization of the test case set, and the strategy is how to sort the test cases. The determination of the strategy is based on the interaction between the environment and the agent, where the agent optimizes next behavior of the state according to the feedback of history behavior, which is measured with reward. So the quality of reward directly affects the quality of RL.

With the application of RL in CI, the agent prioritizes the test cases in the current integration cycle based on the feedback of their behaviors in previous integrations. A metric should be defined to evaluate the effectiveness of the prioritization strategy in RL for the feedback, which can improve reinforcement learning policy. For TCP in CI, this consists of two aspects: what is used to reward a test case, and whether a test case should be rewarded. The reward intensity is computed by the reward function, and the reward strategy decides which one should be rewarded. The reward function and reward strategy are the two main subjects of our study. It should be noted that there is only reward but without punishment in the environment, because compared with the rewarded test cases, unrewarded test cases are the penalized.

Fig. 2 presents the testing process in RETECS proposed by Spieker et al. (2017). For the test cases provided in each cycle, based on the RL policy, which is generated with historical learning, the test cases are got the prioritization and then prioritized, so as to generate test sequences for test execution. For the test results, the evaluation and developer feedback are used to optimize the learning strategy. RETECS evaluates the test execution result through RL after each integration and gives rewards to test cases based on the evaluation. There are three reward functions used in RETECS. Failure Count Reward (FC) rewards every test case with the total failed number of test sequence in each cycle. Test Case Failure Reward (TF) only rewards the failed test case in current cycle. Time-Ranked Reward (TR) rewards the test case with the failed information of the test sequence and the position of the test case. For failed test cases, TR is the same as FC. For passed test cases, TR is further decreased by the number of failed test cases ranked after the passed test case to penalize scheduling passing test cases early. Spieker's experimental results indicated that the test sequence obtained by TF performed better than the other two reward functions.

RL is based on the reward hypothesis, which is the idea that each goal can be described as the maximization of the rewards (Dewey, 2014). For a sequence of test cases in CI, two aspects of reward hypothesis are considered, i.e., what and how to reward. The former focuses on the reward intensity of a test case, that is the study of the reward function. While, the latter focuses on which test case should to be rewarded, that is the study of reward strategy. The three reward functions of RETECS are only based on the execution information of the current CI cycle. It has been widely recognized that the historical execution information of test cases in the entire integration process is more valuable for the TCP (Khatibsyarbini et al., 2018). Based on RETECS, this paper further considers whole historical execution information of test cases, specifically the impact of historical failure count and failure distribution on prioritization. We propose reward functions based on the historical execution information of test cases, and further, with the consideration of the timeliness of historical information (Wu et al., 2019), we propose the time window based reward function. Finally, combined with different reward strategies, an efficient RL method for CI test case prioritization is formed.

3. Reward for reinforcement learning in CI testing

In this section, three history-based reward functions are first proposed to measure the fault-detection capability of a test case in CI testing. Firstly, Historical Failure Count (*HFC*) reward is presented with the consideration of the historical failure count of a test case. Then, the Average Percentage of Historical Failure (*APHF*) is defined to measure the historical failure distribution of a test case, based on which the reward function is proposed. With further consideration of the timeliness of historical information, a time-window based reward function is finally proposed.

How to reward the test cases is another research point, in which two kinds of reward strategy are first presented, total reward and partial reward, and then a fuzzy reward strategy is proposed to overcome the drawback of the previous two. The total reward strategy rewards all test cases, while partial strategy only rewards the failed test cases in the current execution. However, it is tough to determine whether a test case should be rewarded only based on the result of current execution result, because the historical results are more valuable to evaluate the capability of fault detection for a test case. Therefore, it is unreasonable to give a discrete classification to reward a test case based on the current execution result. This inspires us to include the historical execution information of test cases and adopt a fuzzy classification approach as a reward strategy. Fuzzy classification is based on fuzzy logic, which is a mathematical logic that analyzes variables as continuous values between 0 and 1, during the classical or digital logic operates on discrete values of either 1 or 0 (true or false) (Larsen, 1980), respectively.

The reward function and reward strategy together constitute the reward of RL, which enables us to obtain a better test case prioritization strategy. The roadmap of the history-based reward proposed in this paper is presented in Fig. 3.

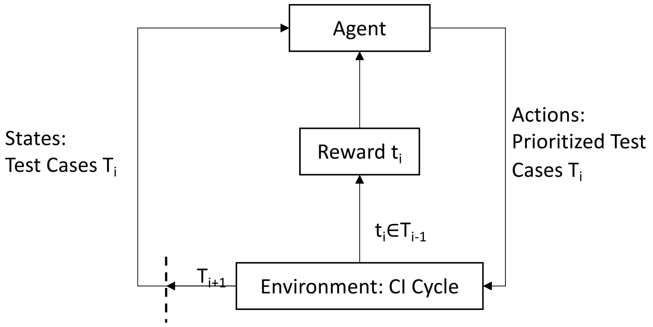


Fig. 1. Reinforcement learning used in continuous integration.

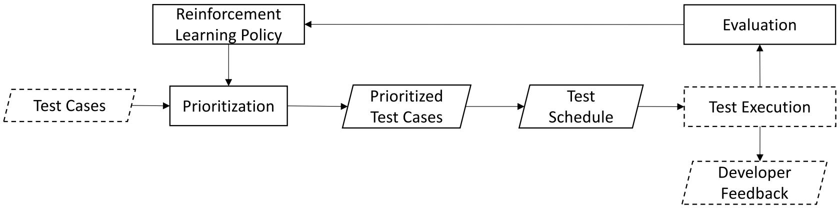


Fig. 2. Testing in CI process: RETECS uses test execution results for learning test cases prioritization (solid boxes: Included in RETECS, dashed boxes: Interfaces to the CI environment).

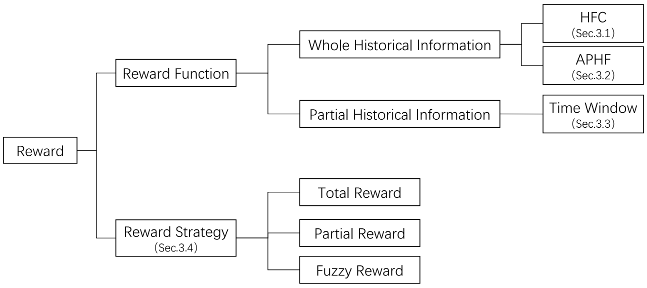


Fig. 3. The frame of history-based reward.

3.1. HFC reward function

In the process of CI testing, the historical execution information of a test case is the embodiment of its fault detection ability. The more faults a test case detected, the more possibility it detects a fault in the next integration. So the number of failures of a test case can be used to measure the fault detection ability of the test case, which can be used as a reward.

In this paper, we use r_j to record the execution result of the test case in its jth execution. The values of r_j is 1 or 0, where 1 indicates failure and 0 indicates pass. The reward function based on historical failure count, which is called Historical Failure Count(*HFC*) reward function, is defined as follows:

$$HFC_i(t) = \sum_{j=1}^{n} r_j \tag{1}$$

where i denotes the ith integration, n denotes the number of historical executions of the test case t. It should be noted that a test case is not necessarily executed in every integration, so the jth execution does not represent the jth CI cycle.

HFC rewards a test case according to its failed number of historical execution. To a certain extent, it can well reflect the fault detection ability of a test case. However, for test cases with the same number of failures, the *HFC* rewards have the same fault detection capability, while test cases with more recent failures are more likely to detect faults.

3.2. APHF reward function

It has been shown that *HFC* cannot distinguish the test cases with the same failure count but occurred in different CI integrations, that is, the historical failure distribution is also very important for evaluating the fault detection capability of a test case. In this section, a new evaluation metric, Average Percentage of Historical Failure (*APHF*), is further proposed and defined as follows:

$$APHF_i(t) = 1 - \frac{\sum_{j=1}^{m} Rank_j}{n \times m} + \frac{1}{2n} \tag{2}$$

where i denotes the ith integration, n denotes the number of historical executions of the test case t, and m denotes the failure count of t in n executions. $Rank_j$ denotes the countdown order of the last jth failure of t.

The range of *APHF* is from 0 to 1. The higher the *APHF* value is, the greater the failure probability of test case is. Thus, the test case with a higher *APHF* value is more likely to detect a fault in the next execution.

For instant, test cases t_1 and t_2 are both executed n times, and their historical execution results are represented as [1, 0, . . ., 0] and [0, . . ., 0, 1]. It can be seen that the failure count of t_1 and t_2 have the same value of 1 in terms of *HFC* reward, but t_1 fails in the recent execution, and t_2 fails in the first execution. Using the *APHF* reward function, the reward values of t_1 and t_2 are $1 - \frac{1}{2n}$ and $\frac{1}{2n}$, respectively. Although t_1 and t_2 have the same failure count, t_1 detects a fault in the last execution that has a higher *APHF* value, i.e., a higher probability of detecting faults in the next integration. Further inspection reveals that with the increasing of n, the value of $APHF(t_2)$ gets infinitely closer to 0. Thus, with the increase of the CI cycles, the failure information of a test case tends to have less impact on evaluating the fault detection capability. The *APHF* reward function can represent this trend properly.

3.3. Time-window based reward function

The reward function based on the historical information can reflect more details about the test case, which can help increase the ability of fault-detection of the test case. But in CI testing, the feedback time should be as short as possible. Compared with the reward function using current information, it takes more time in the calculation of the reward function with historical information. Besides, some historical information, which has not been used for a long time, weakens the ability of fault detection with the delay of the cycles. That is to say, there is redundant information in the historical information of the test case. However, the reward function based on historical information is more effective than that with current information. Too long-unused historical information, which is calculated in the reward function, overstates the fault detection abilities of the test case and incorrectly increases its ordering position. For example, one test case, which failed in the early CI, and never failed in the following CI process, has been fixed an error associated with it. This test case may has no-fault detection capability, until it detects a new fault. The effect of a test case to detect a fault is certain timeliness. According to its timeliness, we introduce a time window method to combine the best of the two methods (Wu et al., 2019).

Based on the definition of *HFC* and *APHF*, the time-window based reward function is defined in Eqs. (3) and (4), called HFC^{tw} and $APHF^{tw}$, respectively. HFC^{tw} calculates the number of historical failures for test case t in previous w executions, in which w is the time window size. The definition is as follows:

$$HFC_i^{tw}(t) = \sum_{j=1}^{w} r_j \tag{3}$$

The value of HFC^{tw} is not higher than the value of *HFC*, owing to the reduction in the number of historical failure. But HFC^{tw} focuses on the impact of recent failure information of test cases on fault detection.

Similarly, $APHF^{tw}$ measures the average percentage of historical failure for test case t in the previous w executions, and the definition is as follows:

$$APHF_i^{tw}(t) = 1 - \frac{\sum_{j=1}^{m} Rank_j}{win(n,w) \times m} + \frac{1}{2 \times win(n,w)} \tag{4}$$

where $win(n, w)$ is the number of historical executions of test case t in the time window w, and m denotes the number of failed executions of $win(n, w)$.

$$win(n,w) = \begin{cases} n & n < w \\ w & n \geq w \end{cases} \tag{5}$$

$APHF^{tw}$ ranges from 0 to 1. Since $APHF^{tw}$ also focus on the impact of the recent time window failures rather than the whole historical executions, the computation cost can drop down compared with *APHF*. Note that, for a test case that is not failed in the time window w executions, the reward is down to 0, which may have a slight impact on the evaluation on the test case. However, the test case, that failed in very early executions but not failed in recent executions, is also unlikely to detect faults in the next execution. Further more, in the RL framework, the order of test cases is based on the expectation of the cumulative discount reward of the reward value of test cases, that is, the reward value in the historical process always exists in the calculation process of the priority of test cases, but the influence is attenuated with a particular factor.

3.4. Reward strategies

In CI cycles, test cases usually have different performance on fault detection. Therefore, which test case should be rewarded in the sequence is another research question. In this section, we present three different reward strategies, total reward, partial reward and fuzzy reward, respectively.

In each CI cycle, a series of test cases are used to test the commits newly submitted. We first introduce the total reward strategy, which is defined in Definition 3.1. In this strategy, each test case in the cycle is treated fairly, i.e., each test case will be rewarded with a reward value to evaluate its contribution in the current cycle.

Definition 3.1 (*Total Reward Strategy*). For $T_i = \{t_1, t_2, \ldots, t_j, \ldots, t_n\}$, every t_j in T_i will be rewarded.

However, for the test cases with the same reward value, which have different execution results currently, in total reward strategy, they will be rewarded with the equal reward value. The test case failed currently is more important than the test case passed. We further put forward the partial reward strategy. With the consideration of current execution results, only the failed test cases are rewarded. The partial strategy is defined as follows:

Definition 3.2 (*Partial Reward Strategy*). For $T_i = \{t_1, t_2, \ldots, t_j, \ldots, t_n\}$, t_j will be rewarded only if t_j fails; others will not be rewarded.

With the partial reward strategy, the more frequently a test case fails, the better its fault-detection capability is.

The above two reward strategies are implemented based on the current execution results of test cases. In the total reward strategy, all test cases are rewarded but may have different reward values. In the partial reward strategy, we emphasize the distinction between the failed test case and the passed test case, where only failed test cases are rewarded. Both reward strategies use the current execution information to determine whether to reward a test case. In the feature of the CI, in each cycle, every incoming test case is related to a submitted commit for the current cycle, which meets the test requirements (Campos et al., 2014). Thus, the more test requirements a test case meets, the more frequency it occurs. It can be seen that the factor of test case's frequency is not considered in the reward strategies based on the current execution information. A fuzzy reward strategy is further proposed to improve the incentives for passed test cases with the consideration of the frequency, where besides of the failed test cases, some passed test cases could be rewarded with fuzzy rules. For example, if a test case is related to the commits often, it appears in many cycles but rarely fails. In the partial reward strategy, the reward value is usually 0 as the test case passes, which keeps lowering the discount expectation in RL for CI, and results in a low ranking priority. But considering its frequent appearance, the test case, which greatly conforms to the requirements of the test (Campos et al., 2014), should be rewarded, which will increase its priority appropriately.

The fuzzy rule is defined according to the environment of CI, in which the proportion of failed frequency of a test case t is calculated as follows:

$$\mathcal{P}_i(t) = \frac{N_f}{N_{total}} \tag{6}$$

Where, i is the current cycle number, n_f is the number of failure count in historical execution, N_{total} is the total number of historical execution count. Again, i is not equal to N, because a test case may not appear in every cycle.

A lower value of the $\mathcal{P}$ usually means that the test case is often associated with some commits, but rarely failed. For such test case, even if it does not fail, it would be better to reward it for improving its ordering, owing to its high frequency, which can be regarded as a vital point to the CI system. The strategy for determining some passed test cases to be rewarded, is defined as the fuzzy reward strategy, which is defined as follows:

Table 1
An example of three test cases in twelve CI cycles.

Test case	c_{12}	c_{11}	c_{10}	c_9	c_8	c_7	c_6	c_5	c_4	c_3	c_2	c_1
t_1	1	–	0	–	0	0	1	0	1	1	–	0
t_2	1	0	0	0	0	1	–	0	0	0	0	–
t_3	0	0	0	0	1	0	0	0	0	0	0	1

Table 2
The comparison of five reward functions for the example presented in Table 1.

Test case	*TF*	*HFC*	*APHF*	HFC^{tw5}	$APHF^{tw5}$
t_1	1	4	17/36	2	1/2
t_2	1	2	7/10	1	9/10
t_3	0	2	1/3	1	1/10

Definition 3.3 (*Fuzzy Reward Strategy*). For $T_i = \{t_1, t_2, \ldots, t_j, \ldots, t_n\}$, if t_j is failed, t_j will be rewarded; if t_j is passed and $\mathcal{P}_i(t)$ meets the threshold, t_j will be rewarded; others will not be rewarded.

The expression can be described by the following formula (7).

$$Reward_i^{fuzzy}(t) = \begin{cases} RewardValue & r = 1 \\ 0 & r = 0 \ \wedge \ \mathcal{P}_i(t) > m \\ RewardValue & r = 0 \ \wedge \ \mathcal{P}_i(t) \leq m \end{cases} \quad (7)$$

where, r is the execution result of the test case t in current cycle, 1 means the test case is failed currently and 0 is passed; m is the threshold value in $\mathcal{P}$, which is defined by the CI environment. In fuzzy reward way, two types of test cases are rewarded, currently failed and currently passed but often appears and occasionally failed in history.

3.5. An example to illustrate the different rewards

In the previous section, three reward functions and three reward strategies based on historical information are introduced. In this section, an example is further presented to illustrate the difference between reward functions based on the current information and historical information.

Table 1 presents three test cases with twelve executions in CI cycles, c_1 to c_{12} respectively. 1 is failed, 0 is passed and '–' is not committed in current cycle. Five reward functions are used in the comparison, including *TF* reward function proposed by Spieker et al. (2017) based on current information, *HFC* and *APHF* proposed in this paper based on historical information, HFC^{tw5} and $APHF^{tw5}$ which are further improved with time-window approach. The corresponding results are presented in Table 2.

With *TF* reward, it can be seen that both t_1 and t_2 are better than t_3, but there is no difference between t_1 and t_2, i.e., the *TF* reward cannot distinguish t_1 and t_2, since it is only based on current execution. Further inspecting the results by all other reward functions that based on historical executions, the results are different.

With *HFC* and HFC^{tw5} reward, t_1 is better than both t_2 and t_3, but t_2 and t_3 are severally equal, that means they have the same fault detection capability of the test cases throughout the historical execution so far, even if it behaves differently in the CI cycle.

Both *APHF* and $APHF^{tw5}$ rewards can distinctly distinguish the test cases' fault-detection ability as different reward values are presented in Table 2 for all three test cases. Compared with *APHF* and $APHF^{tw5}$, the ranking of the test cases is same, which are t_2-t_1-t_3, showing a same trend of evaluation based on the calculation of historical information. Since the time-window based reward $APHF^{tw5}$ uses small amounts of historical information, the computation cost will be lower than *APHF*.

The history-based reward functions proposed in this paper have three advantages: (1) *HFC* contains historical execution results of test cases, rather than the current execution. In the history-based TCP, we believe that if a test case detects a fault in the past, it is more likely to cover the defective code in the next test. And the related research of defect prediction shows that code with faults in the past is likely to have another defect, especially when this part of the code is modified (Noor and Hemmati, 2015; Zimmermann et al., 2007). Therefore, it is necessary to consider the complete historical execution result rather than just the result of the current execution. (2) *APHF* measures the historical failure distribution of the test case, reflecting its timeliness, that is, recently failed test cases will obtain a greater measurement, which is more in line with the sequential decision characteristics of RL. Besides the consideration of the execution results of test cases, we should pay more attention to the failure distribution information, recently failed test cases are more contributive to the next fault detection. (3) The reward value based on the time window of historical information, get the similar reward trend to that with whole historical information, by reducing the amount of information, and the reward value, considering the most recent and frequently failed information, extends the reward value to a certain extent, which will improve its sort position in the sequence.

4. Experiments

In order to verify the validity of the history-based reward functions proposed in this paper, four large-scale industrial datasets, such as Paint Control, IOF/ROL, GSDTSR and Rails, are used as experimental subjects, in which the first three subjects are used in the literature Spieker et al. (2017), and the last two are used in the literature Liang et al. (2018).

CI testing has a strict time limit, so not all the test cases can be executed during the test. In the experiments, according to the research of Spieker et al. (2017), the time threshold is set to be half of the total execution time of all test cases, that is, the test cases will be executed in descending order of their priority until the threshold is reached. We repeat the experiments 60 times to eliminate the effect of random factors on the experimental results. On the one hand, we used the source program provided by the author to carry out repeated experiments of the original method, and on the basis of the source program, we used the reward functions and reward strategies designed by us to conduct the experiments. This section is organized as follows: firstly, we introduce the evaluation metric used in the experiments (Section 4.1) and then the experimental setup with the subjects (Section 4.2), next, Section 4.3 introduces the research questions and Section 4.4 analyzes the experimental results.

4.1. Evaluation metric

To simulate the rapid feedback mechanism of continuous integration, the half of the total execution time is allowed (Spieker et al., 2017), thus only the test cases implemented in the first half time are used in the experimental evaluation. In this paper, four evaluation metrics, NAPFD, TTF, Recall and Time Consumption, are used to compare the effect of different rewards on TCP of CI testing under the RL framework.

NAPFD (Normalized Average Percentage of Faults Detected) (Qu et al., 2007) is adopted as the evaluation metric, which is defined as follows:

$$NAPFD(TS_i) = p - \frac{\sum_{j \epsilon TS_i^{fail}} rank(j)}{\left|TS_i^{fail}\right| \times |TS_i|} + \frac{p}{2 \times |TS_i|} \quad (8)$$

With $p = \frac{\left|TS_i^{fail}\right|}{\left|TS_i^{total,fail}\right|}$, $rank(j)$ represents the position of the j_{th} failed test case in the test sequence TS_i, $\left|TS_i^{fail}\right|$ indicates the total number of test cases failed in TS_i, $|TS_i|$ indicates the total number of test cases in TS_i, and $\left|TS_i^{total,fail}\right|$ indicates the total number of test cases failed in the TS.

Rothermel et al. (2000) firstly proposed Average Percentage of Faults Detected (APFD) to evaluate the effectiveness of TCP techniques. APFD evaluates a test sequence based on the index of test cases failed in the test sequence. However, with the combination of TCP and test case selection, where not all test cases are executed, not all faults can be detected. But APFD assumes that all faults are detected, which is only fit for the situation where there is no test case selection. NAPFD (Qu et al., 2007) is an extension of APFD, which reflects the proportion of faults detected by the test case to all faults, and is suitable for the existence of test case selection. If all faults are detected, NAPFD is the same as APFD (p = 1). When calculating the NAPFD value in this experiment, we assume that the fault corresponding to each failure is different.

The goal of TCP is to get the test cases that can detect faults as far ahead as possible, so the earlier the faults are detected, the better the prioritization. Based on this, we use Test to Fail(TTF) as another evaluation index, and TTF is the first location where the fault is found. The earlier the fault is found, the better the sort, and the smaller the TTF value.

In order to evaluate the sorting effect, Recall is introduced to evaluate the proportion of faults detected in each cycle. It calculates the proportion of faults found in the first half execution time to total faults in each cycle.

Time Consumption represents the average time cost of RETECS. It includes time spent calculating reward function, sorting test cases, and reinforcement learning.

4.2. Experimental setup

Four subjects are used in the experiments, in which Paint Control and IOF/ROL are from ABB Robotics Norway for testing sophisticated industrial robots, GSDTST and Rails are open source datasets shared by researchers. The test case is a unique identifier, which will be used to test different commits in different cycles. These datasets contain historical information such as test case execution results in more than 300 integration cycles.

Table 3 lists the statistics of the four datasets, including the size of the test cases, the number of integration cycles, etc. 'Results' is the number of execution results in the table, which refers to the total number of executions of all test cases during the entire integration process, the 'Failure Rate' represents the proportion of failures in the total number of execution results, and the 'Frequency' is the frequency of each test case in each cycle. It can be seen that Paint Control dataset is small-scale, the test case failure rate is nearly 20%, and each test case appears at the highest frequency of 0.82 per cycle. The IOF/ROL dataset is of medium size but has the highest failure rate in the four datasets with the lowest frequency. As for Google Open Source Dataset GSDTSR, it is the largest, but the failure rate is only 0.25%. For the open source Rails, even though its test case size is medium, its CI cycles are maximum. The failure rate in Rails is only 0.62%. CI test prioritization is complicated when only a small number of test cases failed in a large dataset.

In CI, a test case is often used in different cycles according to the commit, and the execution results may be different for different cycles. We analyze the frequency of fault detection of a test case, that is how often the test case will detect a fault. The statistical results are shown in Table 4, in which the count number of test cases between the continuous failure intervals of (0, 5], (5, 10] and (10, maximum] are listed. For all the subjects, more than 60% of the test cases are continuously failed in the interval of (0, 5]. Thus, in the time-window based reward functions, the size of window is set to 5.

Table 3
The statistics of the four datasets.

Dataset	Test cases	CI cycles	Results	Failure rate	Frequency
Paint Control	89	352	25,594	19.36%	0.82
IOF/ROL	1941	320	32,260	28.79%	0.05
GSDTSR	5555	336	1,260,618	0.25%	0.68
Rails	2010	3263	781,273	0.62%	0.12

Table 4
The interval statistics of the continuous failure distribution.

Paint Control		IOF/ROL		GSDTSR		Rails	
Interval	Count	Interval	Count	Interval	Count	Interval	Count
(0, 5]	4030	(0, 5]	2408	(0, 5]	2409	(0, 5]	4363
(5, 10]	276	(5, 10]	340	(5, 10]	283	(5, 10]	97
(10, 106]	737	(10, 17]	237	(10, 929]	1047	(10, 2494]	492

We further analyze the historical fault detection efficiency, which is defined in Eq. (6). The detailed data is shown in Table 5. The failure rate is 0, which means the test case has not found a fault in historical execution. The low failure rate means a test case frequently occurs during the CI cycle but seldom finds a fault. In the datasets of GSDTSR and Rails, for the low failure rate test cases, there is a considerable proportion, which is larger than that with the failure rate of other test cases. For the datasets of Paint Control and IOF/ROL, most of the failure rates are below to 0.5. In the four experimental subjects, there are test cases with relatively high frequency and low fault detection rate. Test cases are meant to test the submitted commits, and the higher the frequency, the more critical the test case is related to the trunk. The importance of the test case is obtained by analyzing all historical execution information, especially for the low failure test case. For large failure test case, the adjacent failure information is a good measure of its importance.

4.3. Research questions

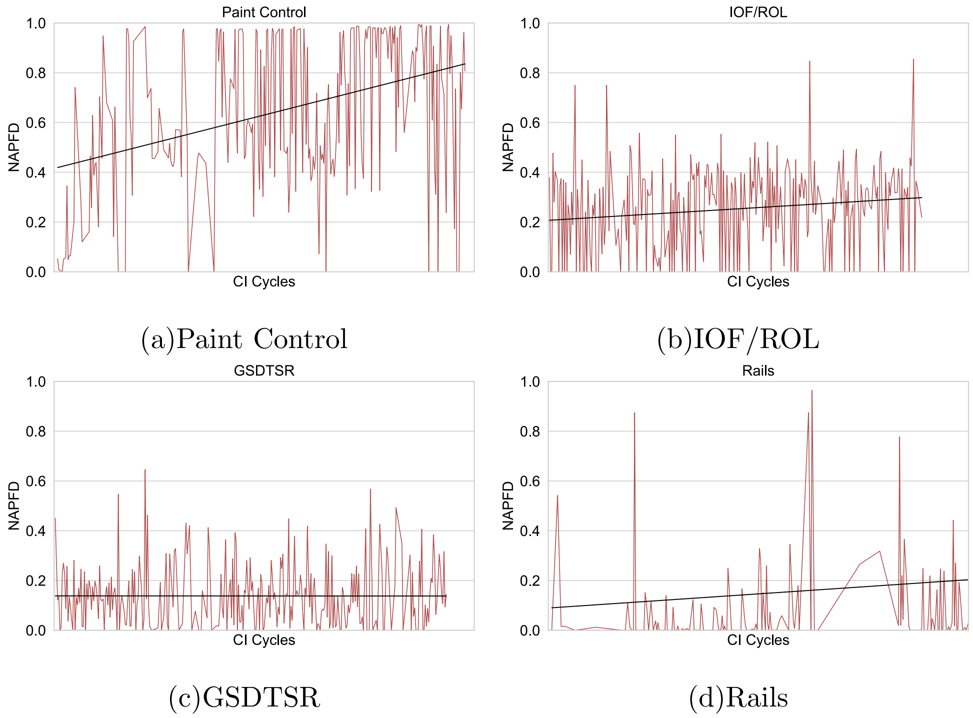
In this section, the proposed three reward functions and three reward strategies are empirically compared. We conduct experiments and analysis based on the following four research questions.

RQ1: Is the history-based reward functions more effective to prioritize test cases than the existing reward function?
RQ2: Does the time-window based reward function has further improvements in CI testing?
RQ3: Does the total reward strategy perform better than the partial reward strategy?
RQ4: Does the fuzzy reward strategy is better than the other reward strategies?

The first two RQs are related with the proposed reward functions. **RQ1** is set to verify the validity of the reward functions that consider the whole historical execution information of test cases. In this paper, we propose two history-based reward functions, namely *HFC* reward and *APHF* reward. *HFC* reward is based on the failure count of the test case in the historical execution process, *APHF* further considers the distribution of failures. Therefore, to answer the **RQ1**, NAPFD, TTF, Recall and Time Consumption are used to compare the proposed two history-based reward functions with the existing TF reward function. **RQ2** is set to verify the impact of the time-window based reward function, in

Table 5
Historical fault detection efficiency.

Failure rate	Paint Control		IOF/ROL		GSDTSR		Rails	
	Count	Percentage	Count	Percentage	Count	Percentage	Count	Percentage
1	132	0.51%	2511	7.22%	198	0.02%	439	0.06%
[0.5, 1)	440	1.71%	6558	18.86%	1425	0.11%	2985	0.38%
[0.1, 0.5)	2025	78.54%	13257	38.13%	4299	0.34%	7933	1.02%
[0.05, 0.1)	2941	11.43%	2802	8.06%	5211	0.41%	6248	0.80%
[0.01, 0.05)	935	3.63%	206	0.59%	26850	2.13%	18340	2.35%
[0.005, 0.01)	16	0.06%	0	0.00%	17665	1.40%	7783	0.99%
[0.001,0.005)	0	0.00%	0	0.00%	9	0.00%	16170	2.07%
[0.00001.0.001)	0	0.00%	0	0.00%	40597	3.22%	33839	4.33%
0	1056	4.10%	9436	27.14%	1164561	92.34%	687974	88.01%



(a)Paint Control

(b)IOF/ROL

(c)GSDTSR

(d)Rails

Fig. 4. The NAPFD results of TF reward.

terms of the effectiveness of the test case sequence and the time cost. **RQ3** studies the effect of different reward mechanisms on prioritization results by comparing total reward strategy (reward all test cases in the current cycle) and partial reward strategy (only reward the failed test cases in the current cycle). **RQ4** is set to verify the effectiveness of the fuzzy reward strategy.

We reproduce Test Case Failure (TF) Reward following the steps from the literature Spieker et al. (2017) for comparison, and implement the RL algorithm based on *HFC* reward and *APHF* reward proposed in this paper. And then, we conduct the experiments of the time-window based the reward functions and the three different reward strategies respectively, total reward strategy, partial reward strategy, and fuzzy reward strategy. The network-based agent is used to represent states and actions consistently.

4.4. Experimental results

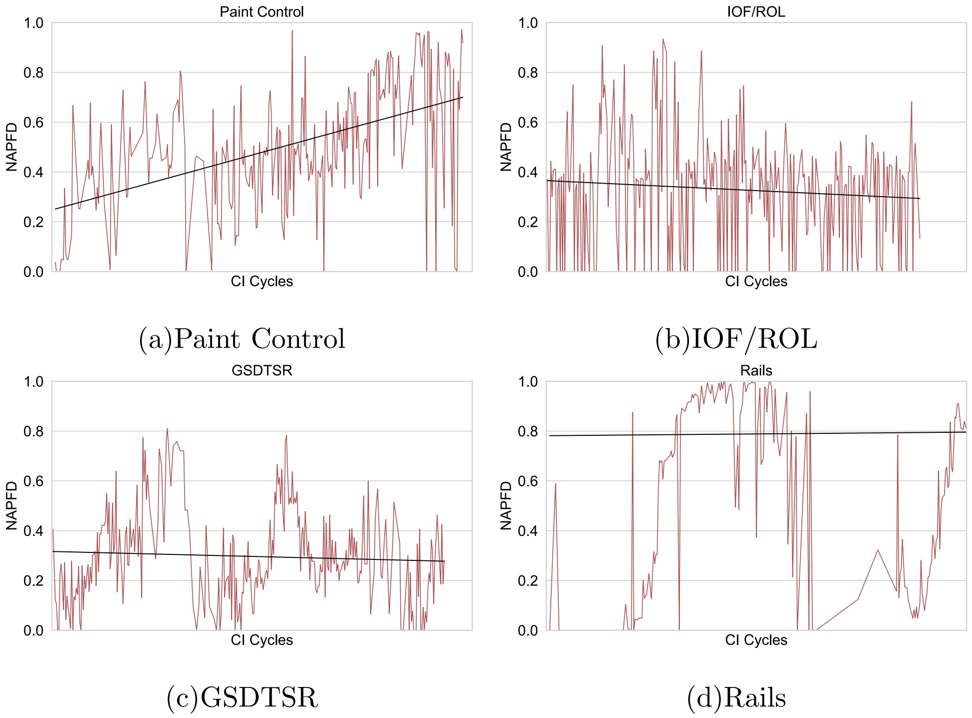
In this section, the experimental results are presented to answer each research question, respectively.

4.4.1. Analysis of RQ1

To answer RQ1, we compare fault detection ability of test sequences generated by the history-based reward functions and TF reward function, where NAPFD, TTF, Recall and Time Consumption are used in the evaluation. Since the TF reward function only rewards the failed test case, which is a partial reward strategy in our definition, only results of partial reward strategy are used for comparison.

The NAPFD results of three reward functions are shown in Figs. 4–6, respectively. The horizontal axis is the number of integration cycles, and the vertical axis is the corresponding NAPFD value of the test sequence. The higher the NAPFD value, the stronger the ability of the test sequence to detect faults. The red curve in the figure is the average of NAPFD value of the test sequence generated by RL method for each integration, and the black line is an unary linear regression fitting line calculated by the standard algorithm, which can reflect the learning trend of RL with different reward functions.

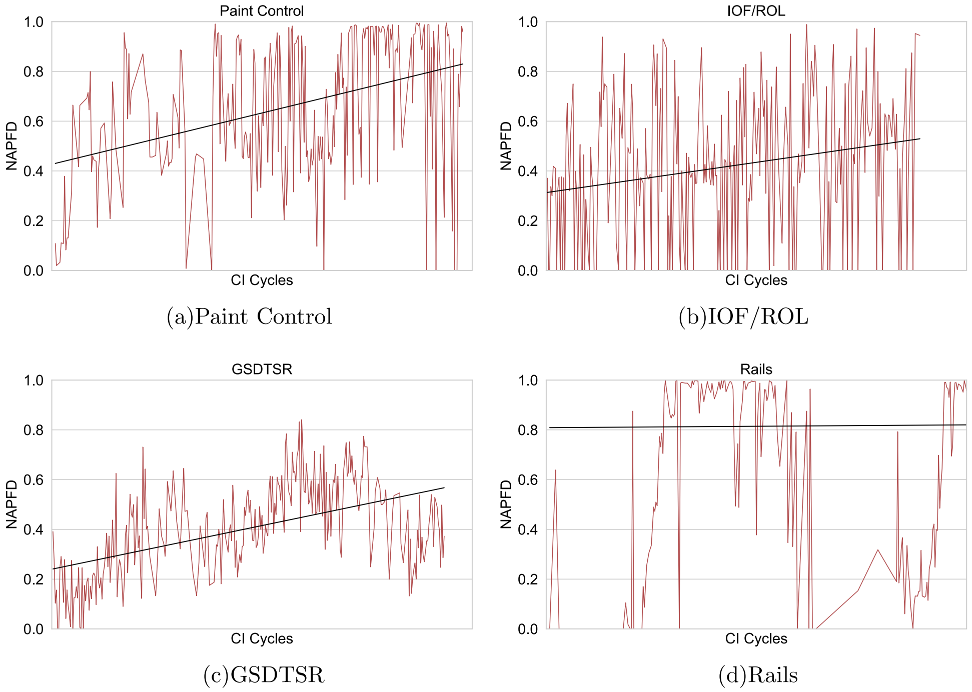
Fig. 4 presents the result of the TF reward function as the baseline of the comparison, and Figs. 5 and 6 present the results of the *HFC*-Partial and the *APHF*-Partial reward function with the partial reward strategy, respectively. It can be seen that for Paint Control dataset, the learning trend of the NAPFD value of the test sequence obtained with the *APHF*-Partial reward is roughly equal to that with TF reward, which has the similar starting and ending values. In ending values, the learning trend of the NAPFD value of the test sequence using *HFC*-Partial reward is



(a)Paint Control (b)IOF/ROL

(c)GSDTSR (d)Rails

Fig. 5. The NAPFD results of *HFC*-partial reward.

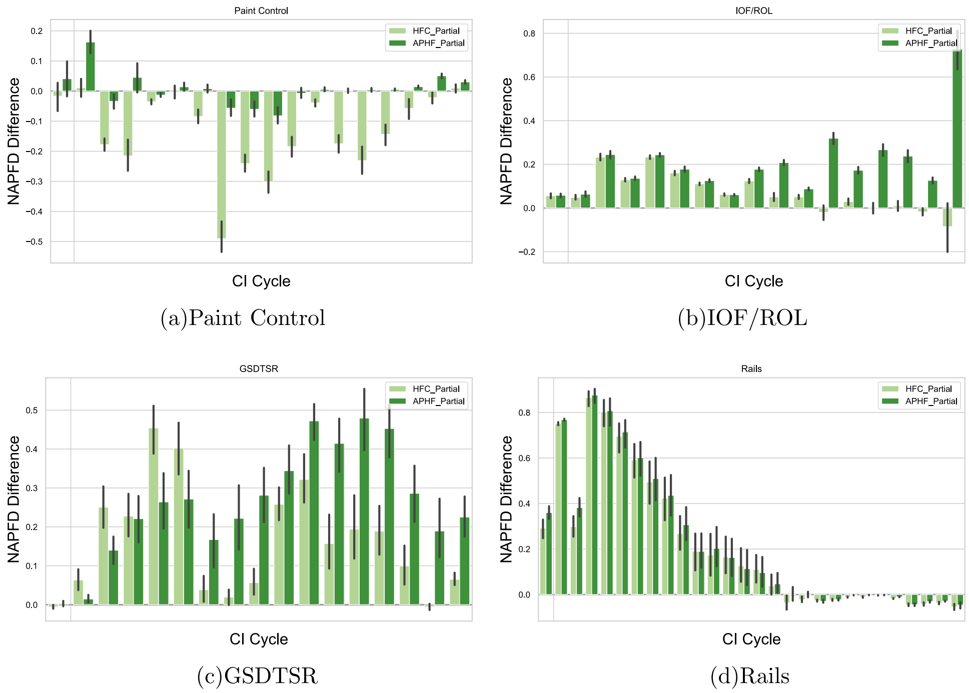


(a)Paint Control (b)IOF/ROL

(c)GSDTSR (d)Rails

Fig. 6. The NAPFD results of *APHF*-partial reward.

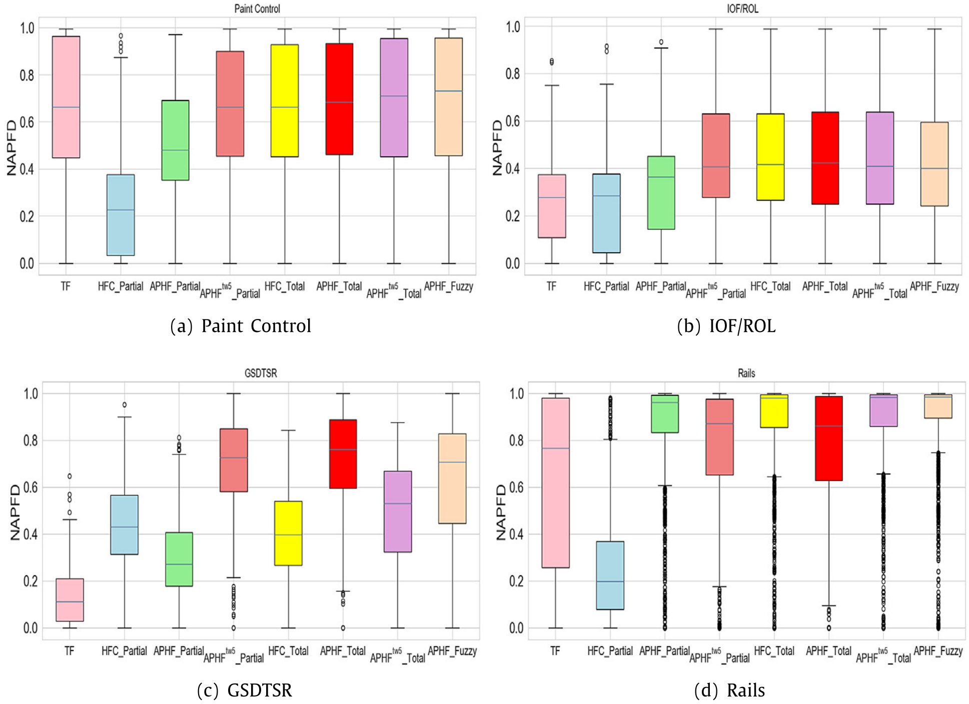
less than 0.7, which is slightly lower than that with the *APHF*-Partial reward and the TF reward (more than 0.8). Besides, there are higher starting values in TF and *APHF*-Partial. For the IOF/ROL dataset, in the ending values, the NAPFD value trend of the test sequence obtained with the *APHF*-Partial reward is greater than 0.5, the NAPFD values of *HFC*-Partial reward and TF rewards are all around 0.3, but with the slope, the *APHF*-Partial rewards are significantly better than the other two. In GSDTSR and Rails datasets, the history-based reward function is obviously better than TF reward function. As for GSDTSR dataset, in the ending values, the NAPFD value trend in the *APHF*-Partial reward and *HFC*-Partial reward are about 0.6 and 0.3 respectively, and the NAPFD value in the TF reward is less than 0.2. In Rails, the NAPFD value trend in ending values increases from 0.2(in TF reward) to about 0.8(in both history-based reward functions). Therefore, the learning trend of the history-based reward function is better, in other words, with continuous integration, historical information based reward functions tend to rank better than then TF in the later stages of CI.



(a)Paint Control (b)IOF/ROL

(c)GSDTSR (d)Rails

Fig. 7. Performance difference between TF reward and history-based reward.



(a) Paint Control (b) IOF/ROL

(c) GSDTSR (d) Rails

Fig. 8. Boxplot of NAPFD produced by different reward functions.

Table 6
Average NAPFD for different reward functions with different reward strategies.

	Partial				Total			Fuzzy
	TF	HFC	APHF	APHF^{tw5}	HFC	APHF	APHF^{tw5}	APHF
Paint Control	60.15%	50.16%	65.31%	67.09%	25.13%	64.82%	69.08%	66.14%
IOF/ROL	25.22%	32.99%	42.02%	42.00%	24.80%	42.30%	40.01%	42.30%
GSDTSR	13.80%	29.72%	40.22%	54.85%	44.11%	68.29%	75.30%	71.92%
Rails	58.64%	84.34%	86.21%	87.10%	22.25%	78.01%	88.17%	76.95%
Average	40.70%	49.30%	58.44%	62.76%	29.07%	63.35%	68.14%	64.33%

Table 7
Average TTF for different reward functions with different reward strategies.

	Partial				Total			Fuzzy
	TF	HFC	APHF	APHF^{tw5}	HFC	APHF	APHF^{tw5}	APHF
Paint Control	3.74	6.74	4.18	3.66	7.07	4.25	3.25	3.81
IOF/ROL	6.62	4.02	1.39	1.33	7.28	1.31	1.80	1.27
GSDTSR	492.47	393.60	284.12	178.55	151.64	61.63	30.01	48.04
Rails	14.12	6.14	4.94	4.41	7.45	11.52	4.05	11.87
Average	129.24	102.63	73.66	46.99	43.36	19.68	9.78	16.25

Table 8
Average Recall of different reward functions with different strategies.

	Partial				Total			Fuzzy
	TF	HFC	APHF	APHF^{tw5}	HFC	APHF	APHF^{tw5}	APHF
Paint Control	76.18%	62.27%	77.21%	78.59%	33.67%	77.16%	80.20%	78.15%
IOF/ROL	34.71%	42.98%	51.58%	51.56%	33.57%	51.99%	49.51%	51.86%
GSDTSR	18.10%	34.84%	45.35%	59.67%	48.87%	73.84%	79.57%	76.44%
Rails	64.80%	88.82%	90.19%	90.89%	28.33%	83.43%	91.69%	82.40%
Average	48.45%	57.23%	66.08%	70.18%	36.11%	71.35%	75.21%	72.21%

Fig. 7 further illustrates the difference of NAPFD between *TF* with *HFC*-Partial and with *AFPH*-Partial, respectively. If *HFC*-Partial or *AFPH*-Partial is better than *TF*, the NAPFD value is positive, otherwise is negative. In the datasets of IOF/ROL, GSDTSR, and Rails, *HFC*-Partial reward and *APHF*-Partial reward are significantly better than *TF* reward, which is consistent with the previous analysis. But in Paint Control, in the early and late stage, the performance of *APHF*-Partial is superior, but there is no obvious advantage in the intermediate process, especially *HFC*-Partial has obvious disadvantage in the intermediate process. Further considering the difference for the four datasets, where Paint Control is the only program with both high failure and high frequency based on the data presented in Table 3, which makes it not only to strengthen its sorting effect, but also due to the historical experience playback of reinforcement learning framework itself, enlarge its own fault detection ability, which causes false priority. Although IOF/ROL has a high failure, its low frequency would not lead to the negative impact on the learning progress.

The boxplots in Fig. 8 present the distribution of NAPFD values produced by various reward functions with different strategies, while the average values of NAPFD are listed in Table 6. It can be seen that the same conclusion as previous is drawn that the history-based reward functions are significantly superior to the TF reward function without using history information of test cases.

TTF, which indicates the first location of the fault, is further analyzed and the average TTF for different reward functions with different strategies are presented in Table 7. Since the smaller TTF value means the earlier detection of the fault, it can be seen that except for Paint Control, the TTF with history-based reward functions is significantly improved.

Table 8 lists the average Recall of different reward functions. Again, in the columns of Partial, with the exception of Paint Control, the history-based reward functions perform significantly superior to *TF* method, especially the *APHF* methods.

Since the history-based reward functions are on the cost of the calculation of the historical information, which makes the result not worse than that with *TF* reward function, we compare the time cost of our proposed method in Table 9. The increased time is in seconds, which is an acceptable range.

Table 9
Average Time Consumption (seconds) for different reward functions with different reward strategies.

	Partial				Total			Fuzzy
	TF	HFC	APHF	APHF^{tw5}	HFC	APHF	APHF^{tw5}	APHF
Paint Control	0.01	0.39	0.21	0.04	0.48	0.36	0.02	0.33
IOF/ROL	0.01	0.23	0.37	0.05	0.29	0.36	0.02	0.68
GSDTSR	1.48	1.65	1.65	1.81	1.77	1.76	1.68	3.34
Rails	0.04	0.11	0.07	0.05	0.42	0.11	0.06	0.14
Average	0.39	0.60	0.58	0.49	0.74	0.65	0.45	1.12

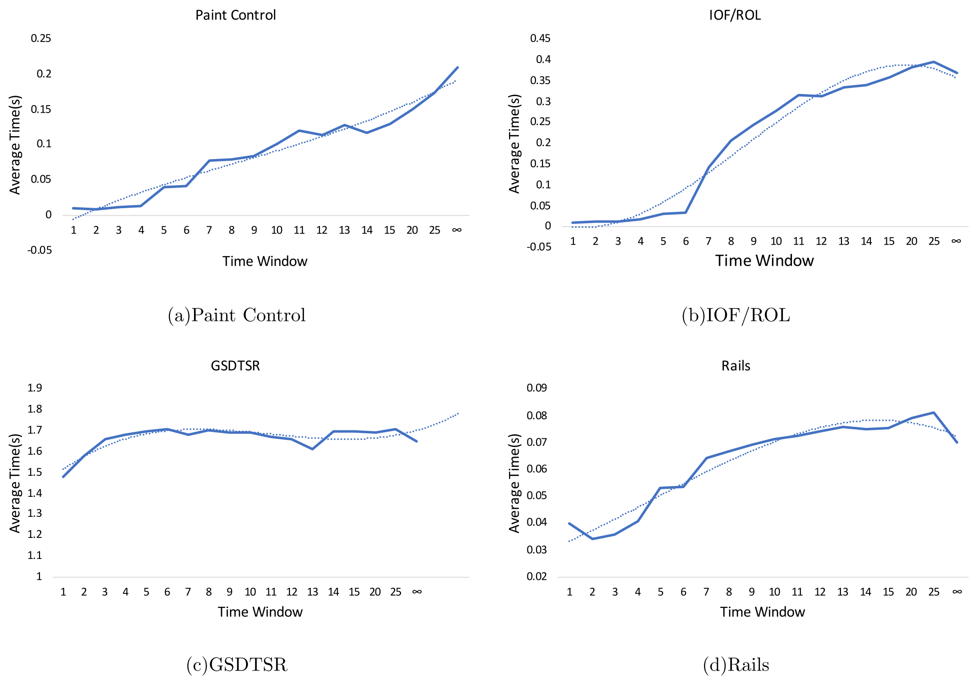
Above all, the *APHF*-Partial reward outperforms *TF* reward in most datasets; for *HFC*-Partial reward, the performance of the *HFC*-Partial reward is better than *TF* reward except for Paint Control dataset; the *APHF*-Partial method is obviously superior to *HFC*-Partial method. In general, the reward function with historical information is better than the reward function with current information, especially for the large-scale GSDTSR and Rails datasets.

4.4.2. Analysis of RQ2

According to the analysis of RQ1, the history-based reward functions are better than the reward function *TF*, especially the *APHF* methods. To some extent, the history-based reward functions increase the rate of fault detection at the expense of Time Consumption. The time-window based reward function only considers the most recent history. With different size of the time window, the reward value of the reward function is different. We analyze the impact of average NAPFD and Time Consumption under different size of time window in Figs. 10 and 9. The X-coordinate is the size of the time window. When the time window is 1, the reward function is *TF*. When the time window is ∞, the reward function is the original *APHF* reward function with the whole historical information. In Fig. 9, the Y-coordinate is the Time Consumption for the time window based reward function in different size. The ordinate is the specific value under the window. The solid line is the value line, and the dotted line is the trend line. In Fig. 10, the Y-coordinate is the average NAPFD of the history-based reward function in different size of time window.

In Fig. 9, although the Time Consumption fluctuates with the increase of the time window, the over trend is evident, where time increases with the rise of the size of the time window. With the use of the time window method, the Time Consumption will decrease by reducing the calculation of historical information.

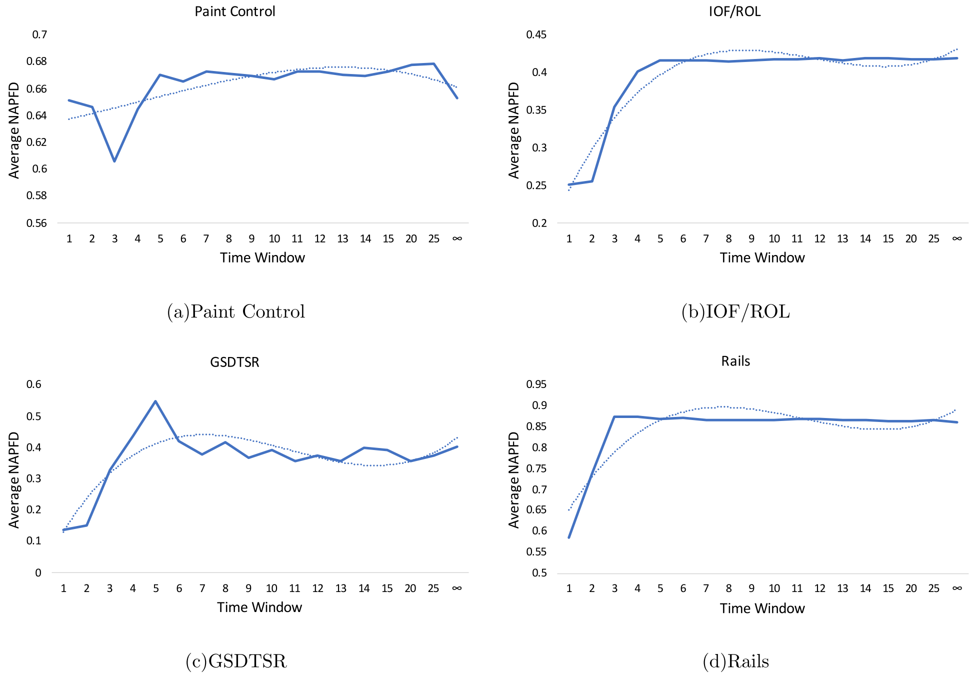
In Fig. 10, the average NAPFD also changes with the increasing of the size of the time window. In addition to Paint Control, the average NAPFD reaches a certain threshold with the increase of time window, and the region becomes stable or fluctuates slightly. For the datasets with low failure rate, it tends to be



(a)Paint Control (b)IOF/ROL

(c)GSDTSR (d)Rails

Fig. 9. Time consumption in different time window.



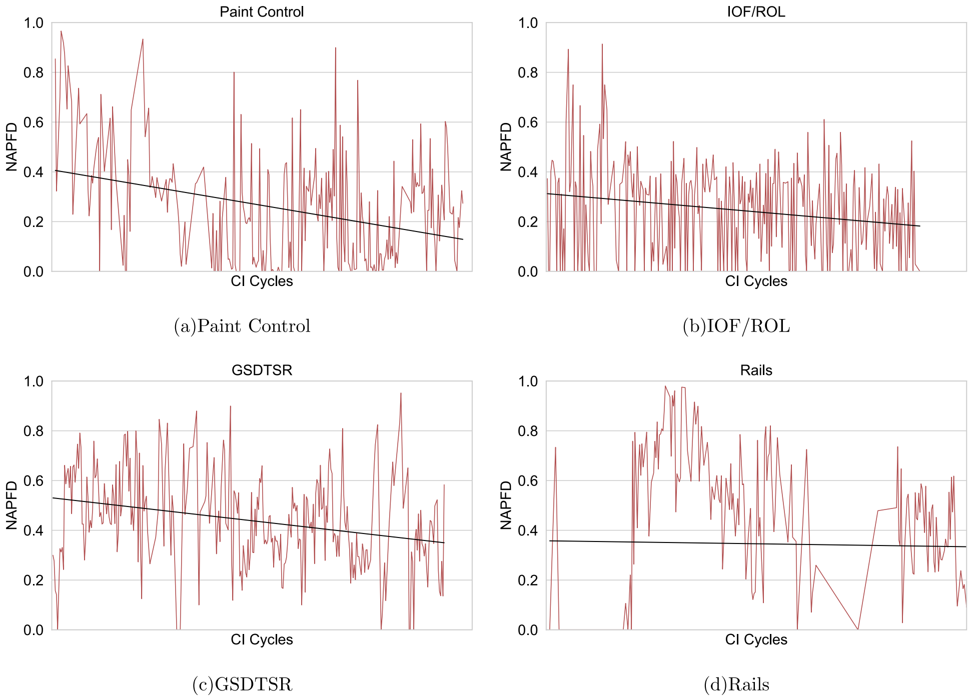
(a)Paint Control (b)IOF/ROL

(c)GSDTSR (d)Rails

Fig. 10. The average NAPFD in different time window.

stable after the threshold value. For the datasets with high failure rate, the fluctuation is obvious after the threshold. Especially in Paint Control, it has an obvious downward trend with the increase of the time window. Further analysis reveals that the frequency of Paint Control is 0.82, namely the possibility that a test case appears in each cycle is 0.82. Combined with the dense continuous failure distribution of the test cases, in the reward function based on historical information, a large number of failed historical information have a great negative impact on the learning efficiency of reinforcement learning. This is because reinforcement learning techniques tend to look for a strategy to maximize long-term future rewards. Even if the rewards with a large number of historical information decline gradually with the cycles, their historical effects are hard to ignore, leading to the prediction of the estimation of the current cycle's fault detection capacity. The more cycles, the greater the impact. Similarly, for high-failure data IOF/ROL, the frequency is only 0.05, i.e., the frequency of each test case in each cycle is very low. However, because of its low continuous failure density, the later fluctuation is not stable.

According to the analysis of RQ1, the history-based reward function, especially the distribution-based reward function *APHF*, is more effective. Therefore, we use the reward function of *APHF* to carry out experimental verification on four datasets with the



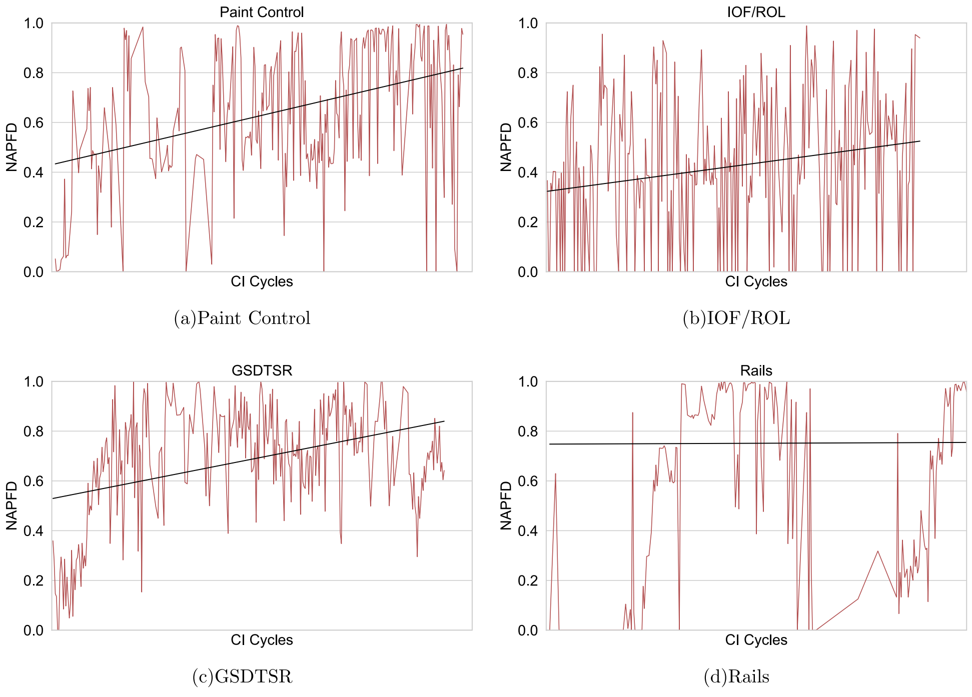
(a)Paint Control

(b)IOF/ROL

(c)GSDTSR

(d)Rails

Fig. 11. The NAPFD results of *HFC*-total reward.



(a)Paint Control

(b)IOF/ROL

(c)GSDTSR

(d)Rails

Fig. 12. The NAPFD results of *APHF*-total reward.

reward function of time window in the size of 5, which is a universal appropriate size time window. Based on the internal comparison of column Partial and column Total in Tables 6, 7, and 8 respectively, under the same reward strategy, on the whole, the reward function based on the time window gets better sorting effect.

In conclusion, with the overall trend presented in Figs. 9 and 10, a properly sized time window can effectively maximize sorting effect and reduce time overhead. With the use of $APHF^{tw5}$, the average NAPFD can improve about 4.55%.

4.4.3. Analysis of RQ3

From the analysis of RQ1, the history-based reward function is superior to the *TF* reward function, and *APHF*-Partial reward function is better in most cases. In RQ3, we further verify which reward strategy(total or partial reward strategy) is better for both reward functions. As the time window-based reward function, with different window sizes, performances different on the datasets, it is not further discussed in terms of relevant strategy discussion.

Figs. 11 and 12 are the experimental results of *HFC* reward and *APHF* reward with total reward strategy, i.e., *HFC*-Total and

Table 10
Comparison of the cycle ratio between the reward strategies of *APHF*-Total and *APHF*-Partial.

	Paint Control	IOF/ROL	GSDTSR	Rails
Equal	7.00%	25.09%	7.04%	26.38%
Partial	55.64%	33.21%	4.23%	54.37%
Total	37.35%	41.70%	95.07%	19.25%

APHF-Total. There is an obvious downward trend along the CI cycles (indicated by the black straight line) with the *HFC*-Total in all datasets, while there is a upward trend for *APHF*-Total. Thus, the *APHF*-Total is far superior to the *HFC*-Total. Similarly, we can see that *APHF*-Partial is greater than that of *HFC*-Partial based on the results presented in Figs. 5 and 6. Since *APHF* reward function is better than *HFC* reward function with both strategies, the following discussion is based on the *APHF* reward function to distinguish the total and partial reward strategies.

The results of different reward strategies in *APHF* reward function are shown in Figs. 6 and 12. From the trend of learning, in Paint Control and IOF/ROL, there are similar trend line, with the same start point and end point. While in GSDTSR, the total reward strategy is distinctly better than the partial reward strategy from the start point and the end point. While in Rails, the partial reward strategy is distinctly better than the total reward strategy from the start point and the end point. For the datasets with a low frequency, the total reward strategy cannot tell the difference among the test cases very well.

Table 10 presents the comparison for each cycle between *APHF*-Total and *APHF*-Partial for all datasets, using the statistic of the ratio of the number of CI cycles (cycle ratio). In Table 10, 'Equal' means the cycle ratio where *APHF*-Total and *APHF*-Partial have the same NAPFD value, 'Partial' means the cycle ratio where *APHF*-Partial has a higher NAPFD value than *APHF*-Total and 'Total' means the cycle ratio where *APHF*-Total has a higher NAPFD value than *APHF*-Partial. For subjects of Pain Control and IOF/ROL, although the learning curve is approximate from Figs. 6 and 12, there is a better percentage of the partial reward in Paint Control and there is a better percentage of the total reward in IOF/ROL. For GSDTSR, although the comparison results are the same, the total score is much higher than the partial score, reaching 95.07%, which is due to GSDTSR's high frequency, where the test cases occur frequently. For Rails, the percentage of the partial reward is higher than that of the total reward, which is same as the conclusion from Figs. 6 and 12.

We further analyzed the first failure location, that is, the size of TTF. From Table 7, with the columns comparison of APHF in Partial and APHF in Total, there are the same phenomenon with the comparison results. For small datasets, the TTF are very similar, and for large datasets, especially GSDTSR, the total reward strategy improves 222.49 than the partial reward strategy, with a high frequency. For Rails, with a low frequency, the partial reward strategy performs better. From the Table 8, with the comparison of *APHF*-Partial and *APHF*-Total, we can see that in small datasets, the Recalls are very close. In GSDTSR, the total reward strategy is superior to the partial reward strategy. In Rails, the Recall with the partial reward is much higher, reaching 90.19%

Based on the above analysis, for small applications, the advantages of the two strategies are not obvious. For large-scale programs, if the reuse rate of test cases is high (a high frequency), it is more appropriate to use total reward strategy, and for the subject with low reuse rate (a low frequency), it is more appropriate to use partial reward strategies.

Table 11
The comparison of average loss rate for different strategies.

Dataset	TF	*APHF*-Total	*APHF*-Partial	*APHF*-Fuzzy
Paint Control	3.89%	2.72%	2.33%	2.72%
IOF/ROL	20.29%	20.29%	20.29%	20.29%
GSDTSR	15.49%	0.70%	1.76%	0.70%
Rails	7.23%	3.62%	3.27%	3.62%
Average	11.72%	6.83%	6.91%	6.83%

4.4.4. Analysis of RQ4

In RQ3, we analyze the advantages and disadvantages of the two different reward strategies. Different reward strategies reward the various objects for enlarging the reward values gap of the test cases to sort. In total reward strategy, every test case will be rewarded with reward function. In partial reward strategy, we widen the gap by not rewarding passed test cases, ignoring the contribution of the test case in history. However, the test cases that passed, based on their historical usage frequency, have some fault detection capability of their own. So we introduce fuzzy reward strategy to increase the reward of the passed test case, which is often used but rarely fails. In this part, we make the experimental comparison of the fuzzy reward strategy and other two reward strategies.

The threshold value of each experimental object is set 0.01, 0.05, 0.0005 and 0.0005, respectively. We compare the average NAPFD for different reward functions in columns of Partial, Total and Fuzzy with *APHF* in Table 6. With the comparison of the columns in Partial and Fuzzy, except for Rails, the average NAPFD of *APHF*-Fuzzy is higher than that of *APHF*-Partial. With the comparison of the columns in Total and Fuzzy, in Paint Control and GSDTSR, the value of the average NAPFD also has increased, with high frequency. For IOF/ROL, the NAPFD of *APHF*-Fuzzy is between that of *APHF*-Partial and *APHF*-Total, where there is not a big difference. But for Rails, there is a decrease of 1.06% in *APHF*-Fuzzy, with a low failure rate. With the comparison of TTF in columns of Partial, Total and Fuzzy in Table 7, in addition to Rails, the TTF in reward strategy is higher than that in *APHF*-Partial and *APHF*-Total. That is because we do not consider the effect of the recently failed test cases. Besides, for the boxplot in Fig. 8, the fuzzy reward function is always the top three award. That means the fuzzy reward strategy is more effective.

We further analyze whether the rate of omission by improving the reward value of the low failure rate test cases. Loss rate means the proportion of cycles with undetected faults to the total number of cycles with faults. With the comparison of the columns in Table 11, with fuzzy strategy, the loss rate of *APHF*-Partial is improved to be equal to the value in *APHF*-Total. Only in GSDTSR, the loss rate of *APHF*-Fuzzy is reduced, due to its high frequency and low failure rate.

In general, the fuzzy reward strategy can improve the partial reward strategy to some extent, without reducing the average NAPFD.

The TCP tries to put the test case that can detect faults as far as possible to the front, and the reward function evaluates the fault detection ability of the test case with the reward value. From the evaluation metric NAPFD, which is used to evaluate the prioritized test cases, we analyze the sorting performance of different reward functions.

For CI systems that fail frequently, history-based rewards are not always superior to the reward that only considers current information. For the frequent failure system, the test cases, which execute failed frequently, lead to a large number of historical failure information. Compared with the continuous failure information and the current failure information, the current failure could be more timely response the test case fault detection ability. In other words, too much history information of test case will reduce the fault detection capability of test case, which can be seen redundant, and the influence of the recent failure information make more help for the test cases.

For the low failure system, which in the CI testing cycles often performs passed, its history information of the test case execution result is usually through state. The reward function only based on the current information, which will usually be 0, because the TCP is unable to provide effective information, namely its effective fault detection ability cannot be measured. But based on the history evaluation methods, through the introduction of historical failure situation, we can effectively measure the fault detection ability of test case, and the test case prioritization can be better.

Combining rewards based on current information and historical information, the time-window based reward function, consider the historical information of the recent few executions. Through the influence of recent failure information on test cases, this avoids the inadaptability of system characteristics that the current information cannot provide effective information and too long historical information brings redundancy.

In general, the history-based reward function proposed in this paper is more suitable for low-failure systems, which is more in line with the actual situation of software development. The reward function with current information is more applicable when software development often fails early on. For the system maladjustment, the time-window based reward function combines the advantages of both reward functions and achieves the best experimental results in all datasets.

4.4.5. Threats to validity

In the context, validity threats are classified into three distinct categories, including internal validity, external validity and construct validity.

Internal. The primary threat is agent stochastic decision-making, especially the stochastic approach used in the initial exploration phase of reinforcement learning. To avoid this threat, we repeated the experiment 60 times and reported the results on average.

In the study of machine learning, its parameters are sensitive to different environments, that is, different parameter configurations are required for different environments. In our experiments, no parameter adjustments are made to compare the results with those of Spieker et al. (2017). In the actual operating environment, the parameters are adjusted for the specific environment.

For the evaluation index, APFD is based on the assumption that all faults are found, and NAPFD is based on actual detected faults. For a finite time simulation of CI testing, we selected only the test cases that ranked in the first 50% of the order. In fact, these test cases did not contain all the faults, so using NAPFD is more effective.

External. The four datasets we used are provided by the researchers. Not only are the three datasets provided by Spieker et al. (2017) repeated, but the corresponding dataset 'Rails' are constructed based on the CI testing logs shared online[2] (Liang et al., 2018). If further datasets are obtained, additional experiments are needed to verify the impact of test case's frequency and failure rate in the TCP of CI testing.

Construct. A threats to construct validity is the assumption, that the more the test cases fail, the better their fault detection ability will be. But that is not always true. Therefore, different reward strategies are introduced to distinguish the fault-detecting ability of test cases.

5. Conclusion and future work

This paper focuses on RL in CI test prioritization and studies the reward mechanism of RL from two aspects: reward function design and reward strategy design. The key contribution is that the historical execution information of test cases is considered in the reward function design for RL used in TCP of CI. Two reward functions are proposed, i.e., *HFC* reward function based on historical failure count and *APHF* reward function based on historical failure distribution of a test case. A time-window based reward function is further proposed to reduce the amount of credit information and Time Consumption. With the aspect of reward strategy, total reward strategy, partial reward strategy, and fuzzy reward strategy are introduced, respectively. Experiments are conducted on four large industrial datasets and the results indicate that: (1) the fault detection ability of test sequence using reward functions with historical execution information has a significant improvement; (2) historical failure distribution information helps to prioritize test cases with potential to detect faults; (3)the time-window based reward function can not only save the Time Consumption, but also improve the average NAPFD; (4) the results of total reward strategy and partial reward strategy are influenced by the characteristics of the program under test, while the fuzzy reward function not only improve the partial reward strategy, but also get better sorting results than that with the total reward strategy.

For future work, we consider: (1) multi-objective test case prioritization based on RL; (2) using more information of test cases, such as coverage information, modification information, similarity information, etc.; (3) combining large-scale neural network and deep learning to optimize the agent of RL.

CRediT authorship contribution statement

Yang Yang: Methodology, Investigation, Writing - original draft. **Zheng Li:** Conceptualization, Writing - review & editing, Supervision. **Liuliu He:** Software, Data curation. **Ruilian Zhao:** Validation, Formal analysis.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The work described in this paper is supported by the National Natural Science Foundation of China under Grant No. 61872026, 61672085 and 61702029, and the Fundamental Research Funds for the Central Universities (XK1802-4).

References

Ammar, A., Baharom, S., Ghani, A.A.A., Din, J., 2017. Enhanced weighted method for test case prioritization in regression testing using unique priority value. In: International Conference on Information Science and Security. pp. 1–6.

Bian, Y., Li, Z., Zhao, R., Gong, D., 2017. Epistasis based ACO for regression test case prioritization. IEEE Trans. Emerg. Top. Comput. Intell. 1 (3), 213–223.

Campos, J., Arcuri, A., Fraser, G., Abreu, R., 2014. Continuous test generation: enhancing continuous integration with automated test generation. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 55–66.

Chi, J., Qu, Y., Zheng, Q., Yang, Z., Liu, T., 2020. Relation-based test case prioritization for regression testing. J. Syst. Softw. 163, 110539.

Cho, Y., Kim, J., Lee, E., 2016. History-Based test case prioritization for failure information. In: 2016 23rd Asia-Pacific Software Engineering Conference, APSEC. pp. 385–388.

[2] http://gihub.com/elbum/CI-Datasets.git.

Dewey, D., 2014. Reinforcement learning and the reward engineering principle. In: National Conference on Artificial Intelligence. pp. 13–16.

Do, H., Rothermel, G., 2006. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models. In: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 141–151.

Elbaum, S., Rothermel, G., Kanduri, S., Malishevsky, A.G., 2004. Selecting a cost-effective test case prioritization technique. Softw. Qual. J. 12 (3), 185–210.

Elbaum, S., Rothermel, G., Penix, J., 2014. Techniques for improving regression testing in continuous integration development environments. In: ACM SIGSOFT International Symposium. pp. 235–245.

Groce, A., Fern, A., Pinto, J., Bauer, T., Alipour, A., Erwig, M., Lopez, C., 2013. Lightweight automated testing with adaptation-based programming. In: IEEE International Symposium on Software Reliability Engineering. pp. 161–170.

Haghighatkhah, A., Mäntylä, M., Oivo, M., et al., 2018. Test prioritization in continuous integration environments. J. Syst. Softw. 80–98.

Henard, C., Papadakis, M., Harman, M., Jia, Y., Traon, Y.L., 2016. Comparing white-box and black-box test prioritization. In: International Conference on Software Engineering. pp. 523–534.

Jahan, H., Feng, Z., Mahmud, S.M.H., Dong, P., 2019. Version specific test case prioritization approach based on artificial neural network. J. Intell. Fuzzy Systems 36 (6), 6181–6194.

Jiang, B., Zhang, Z., Chan, W., Tse, T., 2009. Adaptive random test case prioritization. In: Automated Software Engineering. pp. 233–244.

Khatibsyarbini, M., AdhamIs, M., Jawawi, D.N., Tumeng, R., 2018. Test case prioritization approaches in regression testing: A systematic literature review. In: Information and Software Technology. pp. 74–93.

Larsen, P.M., 1980. Industrial applications of fuzzy logic control. Int. J. Man-Mach. Stud. 12 (1), 3–10.

Li, Z., Bian, Y., Zhao, R., Cheng, J., 2013. A fine-grained parallel multi-objective test case prioritization on GPU. In: International Symposium on Search Based Software Engineering. pp. 111–125.

Li, Z., Harman, M., Hierons, R.M., 2007. Search algorithms for regression test case prioritization. IEEE Trans. Softw. Eng. 33 (4), 225–237.

Liang, J., Sebastian, E., Gregg, R., 2018. Redefining prioritization: continuous prioritization for continuous integration. In: Proceedings of the 40th International Conference on Software Engineering. pp. 688–698.

Lu, Y., Lou, Y., Cheng, S., Zhang, L., Hao, D., Zhou, Y., Zhang, L., 2016. How does regression test prioritization perform in real-world software evolution? In: International Conference on Software Engineering. pp. 535–546.

Luo, Q., Moran, K., Poshyvanyk, D., 2016. A large-scale empirical comparison of static and dynamic test case prioritization techniques. In: Foundations of Software Engineering. pp. 559–570.

Mahdieh, M., Mirian-Hosseinabadi, S.-H., Etemadi, K., Nosrati, A., Jalali, S., 2020. Incorporating fault-proneness estimations into coverage-based test case prioritization methods. Inf. Softw. Technol. 106269.

Marijan, D., Gotlieb, A., Sen, S., 2013. Test case prioritization for continuous regression testing: An industrial case study. In: IEEE International Conference on Software Maintenance. pp. 540–543.

Memon, A., Gao, Z., Bao, N., Dhanda, S., Micco, J., 2017. Taming Google-Scale continuous testing. In: IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track. pp. 233–242.

Noor, T., Hemmati, H., 2015. Test case analytics: Mining test case traces to improve risk-driven testing. In: IEEE International Workshop on Software Analytics. pp. 13–16.

Noor, T.B., Hemmati, H., 2016. A similarity-based approach for test case prioritization using historical failure data. In: IEEE International Symposium on Software Reliability Engineering. pp. 58–68.

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

Reichstaller, A., Eberhardinger, B., Knapp, A., Reif, W., Gehlen, M., 2016. Risk-Based interoperability testing using reinforcement learning. In: International Conference on Testing Software and Systems. pp. 52–69.

Rothermel, G., Untch, R.J., Chu, C., 2000. Prioritizing test cases for regression testing. IEEE Trans. Softw. Eng. 27, 929–948.

Souza, L.S.D., Miranda, P.B.C.D., Prudencio, R.B.C., Barros, F.D.A., 2011. A multi-objective particle swarm optimization for test case selection based on functional requirements coverage and execution effort. In: IEEE International Conference on TOOLS with Artificial Intelligence. pp. 245–252.

Spieker, H., Gotlieb, A., Marijan, D., Mossige, M., 2017. Reinforcement learning for automatic test case prioritization and selection in continuous integration. In: ACM Sigsoft International Symposium on Software Testing and Analysis. pp. 12–22.

Srikanth, H., Hettiarachchi, C., Do, H., 2016. Requirements based test prioritization using risk factors. Inf. Softw. Technol. 69, 71–83.

Strandberg, P.E., Sundmark, D., Afzal, W., Ostrand, T.J., Weyuker, E.J., 2016. Experience report: Automated system level regression test prioritization using multiple factors. In: IEEE International Symposium on Software Reliability Engineering. pp. 12–23.

Sutton, R.S., Barto, A.G., 1999. Reinforcement learning: An introduction. In: Neural Information Processing Systems.

Vasilescu, B., Yue, Y., Wang, H., Devanbu, P., Filkov, V., 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In: Joint Meeting on Foundations of Software Engineering. pp. 805–816.

Wong, W.E., Horgan, J.R., London, S., Bellcore, H.A., 1997. A study of effective regression testing in practice. In: The Eighth International Symposium on Software Reliability Engineering, 1997, Proceedings. pp. 264–274.

Wu, Z., Yang, Y., Li, Z., Zhao, R., 2019. A time window based reinforcement learning reward for test case prioritization in continuous integration. In: Asia Pacific Symposium on Internetware. pp. 1–6.

Yoo, S., Harman, M., 2015. Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. 22 (2), 67–120.

You, D., Chen, Z., Xu, B., Luo, B., Zhang, C., 2011. An empirical study on the effectiveness of time-aware test case prioritization techniques. In: ACM Symposium on Applied Computing. pp. 1451–1456.

Yu, L., Xu, L., Tsai, W.T., 2010. Time-constrained test selection for regression testing. In: Advanced Data Mining and Applications. pp. 221–232.

Zhang, L., Zhou, J., Hao, D., Zhang, L., Mei, H., 2009. Prioritizing JUnit test cases in absence of coverage information. In: International Conference on Software Maintenance. pp. 19–28.

Zimmermann, T., Premraj, R., Zeller, A., 2007. Predicting defects for eclipse. In: International Workshop on Predictor MODELS in Software Engineering. 2007. Promise'07: ICSE Workshops. pp. 76–82.

Yang Yang is a Ph.D candidate of Computer Science in the department of computer science at Beijing University of Chemical Technology.

Zheng Li is a full professor of Computer Science in the department of computer science at Beijing University of Chemical Technology. He obtained his Ph.D. from King's College London, CREST centre in 2009 under the supervision of Mark Harman. He is the author of over 50 publications, co-guest editor for 3 journal special issues and has served on 20 programme committees. He has worked on program testing, source code analysis and manipulation. More recently he is interested in search-based software engineering.

Liuliu He is a master student of Computer Science in the department of computer science at Beijing University of Chemical Technology.

Ruilian Zhao is s a professor of Computer Science in the department of computer science at Beijing University of Chemical Technology. She has worked on software testing and test data generation.