The Journal of Systems & Software 205 (2023) 111823

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

Integration test order generation based on reinforcement learning considering class importance ☆

Yanru Ding [a,b], Yanmei Zhang [a,*], Guan Yuan [a], Shujuan Jiang [a], Wei Dai [b], Yinghui Zhang [a]

[a] School of Computer Science and Technology, China University of Mining and Technology, Xuzhou, 221116, Jiangsu, China
[b] Artificial Intelligence Research Institute, China University of Mining and Technology, Xuzhou, 221116, Jiangsu, China

ARTICLE INFO

Article history:
Received 3 November 2022
Received in revised form 26 July 2023
Accepted 7 August 2023
Available online 12 August 2023

Keywords:
Integration testing
Class integration test order
Class importance
Reinforcement learning
Stubbing cost

ABSTRACT

The task of ordering classes reasonably in the context of integration testing has been discussed by many researchers. Existing methods regard the class integration test order with the minimum stubbing cost as the optimal result. However, they ignore that class importance can also affect the class integration test order. This paper presents a design of a new algorithm, which considered the class importance, and its evaluation using computational experiments. Specifically, two novel reinforcement learning-based methods to generate class integration test orders are proposed, which aim to consider the class importance and minimize the stubbing cost. First, we advance the concept of class importance and optimize its measurement method. Then, we refine the calculation of stubbing complexity, which is the evaluation indicator of stubbing cost. After that, we combine both class importance and the stubbing complexity into the reinforcement learning algorithm to guide the agent to explore. Finally, we evaluate the proposed methods using computational experiments on five benchmark programs and four open-source programs. The experimental results show that our proposed methods can significantly reduce the stubbing cost while prioritizing the classes of high importance.

1. Introduction

Integration testing is an important way to ensure software quality (Xie and Li, 2018). During integration testing, if the currently tested class requires the services provided by other classes that have not been integrated, it is necessary to construct test stubs to simulate the corresponding methods or properties of these classes (Kung et al., 1995b). For example, a program to be integrated contains *n* classes, and class A depends on class B. If testers need to integrate class A before class B, they should first construct a stub that simulates class B for class A.

Before constructing a stub, testers need to understand the semantics of a function called by another class that this stub needs to simulate. Although some member functions are tiny, on average, it takes about 0.79 man-hours to prepare a stub for each function (Kung et al., 1995b). Some tools (e.g., Mockito and JUnit) have been developed to create stubs automatically (Zhang et al., 2021). However, the generation technology still needs to be improved, and the stubs constructed can be error-prone (Beizer, 2003). Under such circumstances, we need to arrange the order of integration for classes reasonably to minimize the stubbing cost (Binder, 1999). That is, we need to solve the class integration test order (CITO) problem.

There are two existing evaluation indicators to measure the stubbing cost of CITOs: the number of stubs (Kung et al., 1995a) and the overall stubbing complexity (Briand et al., 2002). In the beginning, Kung et al. (1995a) found that the stubbing cost was relatively high in generating CITOs, and the number of stubs that simulate classes could be used as the evaluation indicator of CITOs. Then, Briand et al. (2002) found that breaking two dependencies may be cheaper than breaking one dependency sometimes. For this situation, they proposed using the stubbing complexity, calculated relying on inter-class coupling information, as the evaluation indicator. Usually, the overall stubbing complexity is measured by attribute and method complexity, which can be obtained from static analysis.

One of the major goals of integration testing is to improve software quality by detecting as many faults as possible in a program under test (Huo et al., 2021; Tahvili et al., 2018; Borner and Paech, 2009). A software defect is a potential risk associated with the high complexity of software program architectures (Wang et al., 2016b). Effective testing that finds bugs earlier can lead to higher quality software products sooner (Tahvili et al., 2016). Some researchers have found that the scope of defects a class can cause is closely related to its importance (Rana et al., 2016, 2014), and finding these important classes and testing them first

☆ Editor: A. Bertolino.

* Corresponding author.
E-mail addresses: yrding@cumt.edu.cn (Y. Ding), ymzhang@cumt.edu.cn (Y. Zhang), yuanguan@cumt.edu.cn (G. Yuan), shjjiang@cumt.edu.cn (S. Jiang), weidai@cumt.edu.cn (W. Dai), zyh_halo@cumt.edu.cn (Y. Zhang).

https://doi.org/10.1016/j.jss.2023.111823
0164-1212/

can significantly improve the stability of the software (Meyer et al., 2015; Li et al., 2018). The important classes have a significant impact on the generation of CITOs, based on the following considerations: (1) important classes have a greater impact on a program than independent classes, as important classes tend to involve more dependencies; (2) constructing stubs for important classes is more expensive, as it requires simulating more inter-class calling relationships; (3) important classes are more likely to cause errors and have a broader range of error propagation, as an important class has a greater chance of calling a faulty class and is more likely to be affected by it (Zhou and Leung, 2006).

Although stubbing important classes may prevent faults from spreading to them, in reality, the actual classes are ultimately integrated to complete testing. Therefore, by prioritizing the integration and test order of important classes with a high potential impact on the software system, we can provide additional assurance for software quality assessment. Zaidman and Demeyer (2008) applied the Hyperlink-Induced Topic Search (HITS) algorithm (Kleinberg, 1999) to measure class importance for web mining, Pan et al. (2018) used a weighted k-kernel decomposition method to calculate class importance, and Wang et al. (2016a) proposed a Class-HITS method to calculate the class importance by regarding the complexity and influence of the class as the authority scores and hub scores in the HITS algorithm. These methods demonstrated the value of studying class importance, yet few CITO generation methods consider this factor.

Reviewing existing methods for solving the CITO problem, we can categorize them into three types: graph-based, search-based, and Reinforcement learning-based (RL-based) methods (Zhang et al., 2018b). Graph-based methods use Object Relation Diagrams (ORD) to illustrate program dependencies and generate class orders by reverse topological sorting. Search-based methods use various intelligent and non-intelligent algorithms to tackle the CITO problem. RL-based methods use empirical data collected from an agent's interactions with the environment to optimize CITO generation strategies. RL is a popular artificial intelligence technology in machine learning that is effective in solving sequential decision-making problems. It has been successfully applied in simulation (Huang et al., 2021), games (Silver et al., 2016), test case generation (Yang et al., 2021), and CITO generation (Czibula et al., 2018; Ding et al., 2022), and has shown good performance.

The CITO problem is an NP-complete problem, and finding an exact solution is challenging. All three types of methods mentioned above can help minimize testing costs and predict stubbing costs as accurately as possible. However, RL algorithms are dynamic optimization methods, not static optimization functions. In practice, RL can continuously improve test strategies and adjust them based on actual test results. The intelligent agent in RL learns the best test strategy by interacting with the system, selecting the next test order based on previous test results. This approach enables us to find the best test strategy and apply RL to automate testing, save time and resources, and enhance the accuracy and reliability of testing. Therefore, RL-based CITOs generation can significantly improve testing efficiency and accuracy, offering software engineers better testing tools and methods to ensure software quality and reliability.

In this paper, we propose two novel CITO generation methods based on RL considering the class importance, which is a further expansion of our original research (Ding et al., 2022). The proposed CITO generation methods are called CITO_CRL (**C**lass **I**ntegration **T**est **O**rder Considering **C**lass Importance Based on **R**einforcement **L**earning) and CITO_CRL', which are RL methods considering class importance combined with the stubbing cost and the test profit (Zhang et al., 2017), respectively. Specifically, CITO_CRL and CITO_CRL' first identify the calling relationship between classes through static analysis. After getting the out-degree and in-degree of each class, we can obtain the importance of a class by measuring its impact and complexity. Second, CITO_CRL and CITO_CRL' use a new weight calculation method, which is the entropy weight method, to optimize the calculation of the stubbing complexity. Finally, CITO_CRL and CITO_CRL' respectively incorporate the importance of classes into the design of RL algorithms to generate CITOs. We evaluate CITO_CRL and CITO_CRL' by conducting experiments on nine programs. The results show that the ranks of important classes could affect the stubbing cost of CITOs, and our methods could generate CITOs with minimal stubbing complexity while considering class importance. The research ideas of these two methods are basically the same, but in detail, different strategies combining class importance are proposed to guide the agent to conduct exploratory actions. To our best knowledge, the contributions of this paper are as follows:

- An improved strategy for measuring class importance in object-oriented programs is proposed, and the class importance value is derived from inter-class dependencies.
- A weight optimization method is designed to calculate the stubbing complexity. For the complexity of attributes and methods, the entropy weight method is used to calculate their weights.
- Two novel RL-based methods incorporating class importance are proposed to guide the agent's exploration. To the best of our knowledge, these methods have never been used to solve the CITO problem.
- Computational experiments evaluate and confirm the effectiveness of our methods on benchmark programs and open-source programs in terms of the number of stubs, the optimized overall stubbing complexity, permutation position of important classes, and other metrics.
- A more flexible solution for the CITO problem is provided to testers and developers. Based on our approach, testers and developers can prioritize the integration and testing of important classes while minimizing the stubbing cost.

The rest of this paper is organized as follows. Section 2 reviews related work on CITO generation. Section 3 presents the research motivation. Section 4 introduces the approach. Experiments are carried out in Section 5. Section 6 concludes this paper.

2. Related works

Traditional methods of generating CITOs are always classified into graph-based and search-based methods. Recently, with the rapid development of machine learning technologies, RL-based methods to generate CITOs have been proven highly efficient.

2.1. Graph-based methods

Graphs can be divided into two types: acyclic and cyclic graphs. Cycles in a graph make it difficult to sort classes topologically. Therefore, for cyclic graphs, Kung et al. (1995a) proposed a cycle-breaking solution for generating CITOs based on ORD. ORD is a digraph that uses vertices to represent classes and edges to represent inter-class relationships. Based on ORD, Kung et al. (1995a) described a strategy for generating CITOs by prioritizing the deletion of weakly connected relationships, such as associations. In this way, they reduced the cost of constructing stubs.

Later, researchers proposed several schemes for assigning weights to the edges of cyclic graphs, and they broke cycles based on that. Tai and Daniels (1997) assigned two different levels for inter-class relationships: the minor-level maps to associations and the major-level maps to inheritances and aggregations.

They took the sum of the number of incoming and outgoing dependencies of each edge as the weight.

Tarjan (1972) proposed a method of distinguishing strongly connected components based on a depth-first search. Le Traon et al. (2000) made improvements to that. They performed a depth-first search in ORD and assigned weights to each node. They deleted the entry edge of the node with the largest weight until there was no cycle in the graph.

Briand et al. (2003) made further improvements based on Traon et al.'s algorithm (Le Traon et al., 2000). They divided stubs into specific and generic stubs according to their simulation part, and they redesigned the weight of each edge. Moreover, they only deleted associated edges to break cycles.

By analyzing the inter-class relationships, Briand et al. (2001) distinguished method calls and attribute dependencies to evaluate stubbing cost. Under the consideration of transitive dependencies, Jiang et al. (2021) proposed a new coupling measurement combined with the control coupling. This method further reduced the overall stubbing complexity by deleting edges that involved more cycles.

The multi-level feedback strategy and the method considering the similarity of classes proposed by Zhang et al. (2017, 2021) further expanded the breadth and depth of solving the CITO problem through graph-based approaches.

2.2. Search-based methods

Search can be divided into uninformed search and informed search. Informed search, also known as heuristic search, has been studied extensively by scholars.

Briand et al. (2002) and Le Hanh et al. (2001) both used the genetic algorithm (GA) to address the CITO problem, and they redesigned the fitness function by the overall stubbing complexity. After several iterations, a suitable CITO can be obtained. Wang et al. (2011) applied an random interaction algorithm (RIA) to solve the CITO problem. This method combined GA and simulated annealing (SA) algorithm. Besides this, They designed a new inter-class coupling information measurement to break the inter-class dependencies and generated CITOs based on the principle of minimizing stubbing complexity. Zhang et al. (2018a) proposed a method using the particle swarm optimization (PSO) algorithm by mapping CITOs to particles in a one-dimensional search space. In this optimization method, each CITO was treated as a particle, and an iterative process was used to find the optimal particle position, which corresponds to the CITO with the lowest stubbing complexity.

Zhang et al. (2019) broke cycles with a new heuristic method, which was inspired by the exploitation and exploration of evolutionary algorithms. When given a set of deleted dependencies, their heuristic method can find a near-ideal set of class dependencies that have the same or more cycles as the initialized dependencies but cost less to construct stubs.

A hyper-heuristic algorithm provides a high-level strategy that manipulates or manages a set of low-level heuristics to get new heuristics. Guizzo et al. (2015), Mariani et al. (2016) first proposed a hyper-heuristic algorithm to address the CITO problem. This algorithm filtered out the optimal crossover and mutation probabilities for each evolutionary iteration until a satisfactory CITO was obtained.

2.3. RL-based methods

Researchers have found that applying RL to solve the CITO problem can get better results. Czibula et al. (2018) were the first to address the CITO problem by RL. They trained the agent with the Q-learning algorithm, then used generic stubs and specific stubs as indicators to evaluate the pros and cons of the method. They recorded the agent's behavior paths and treated them as CITOs. This method could further reduce the number of stubs constructed when generating optimal CITOs. On this basis, we (Ding et al., 2022) proposed a method for generating CITOs with the optimization goal of reducing the overall stubbing complexity. This method decreased the stubbing cost by constructing less expensive stubs, and it confirmed the feasibility of applying RL to the problem of CITO generation.

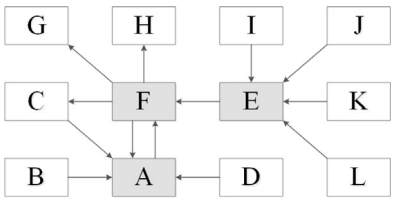


Fig. 1. ORD of the sample program.

3. Motivation

In this section, we explain how important classes affect the stability of software programs with a sample program whose ORD is presented in Fig. 1.

The sample program contains 12 classes, from class A to class L. For convenience, the inter-class dependencies are all set to association relationships. Classes A, E, and F have more dependencies than other classes. We can say that these three classes are more complex than others in the program. The more complex a class is, the more significant impact it has. Therefore, to avoid a faulty class having an excessive impact on the program, we need to evaluate each class and give priority to the high-complexity class in integration testing. To evaluate each class, we need to examine the dependencies a class is involved in, and the most obvious characteristic of dependencies is the in and out-degree.

From the perspective of in-degree, both classes A and E have the largest value, which is four. Classes B, C, D, and F depend on class A. Classes I, J, K, and L depend on class E. It can be considered that a class would have a greater impact when it has a larger value of in-degree. If two classes have the same value for in-degree, we need further analysis. Both classes A and E have the same in-degree, whose value is 4. But class F, which depends on class A, involves more complex dependencies than classes that depend on class E. If class A is a faulty class, it will affect more classes than class E, and therefore the impact of class A is much greater than class E.

From the perspective of out-degree, class F depends on four classes: A, C, G, and H. If one of these four classes is faulty, class F has a high probability of error. That is, the error-proneness of class F is the highest among the 12 classes. Therefore, we need to improve the test priority of class F in integration testing.

The above analysis only involves simple inter-class dependencies. Further research should be conducted when other strong dependencies exist, such as inheritance and aggregation. In short, we can conclude that classes with more dependencies are likely to affect the stability of the software, and we need to evaluate classes to prioritize the important ones in integration testing.

A usual way to get a CITO is: first, by analyzing the inter-class relationships, we find the strongly connected components (cycles) in a program; then, we break these cycles and perform a reverse topological sort on the classes to get CITOs. Through the ORD of Fig. 1, it can be found that there are two cycles in the program: A-F-A and A-F-C-A. We can either delete edge A-F or delete both F-A and F-C to break cycles. Without considering the

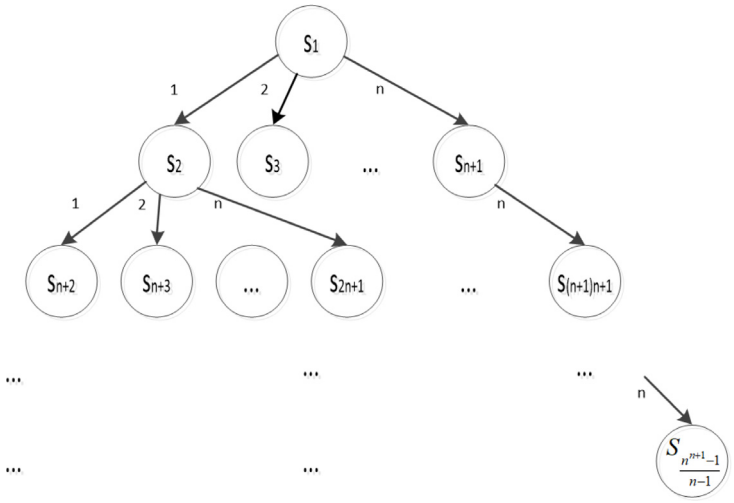


Fig. 2. Unfolding of the state-transition system of Reinforcement Learning.

difference of dependencies between classes, the fewer test stubs are constructed, the lower the stub cost will be taken. Therefore, we choose to delete edge A-F. A possible CITO [A, G, H, C, F, B, D, E, I, J, K, L] is obtained by reversing topological sorting.

Classes A, F, and E with higher class importance are ranked in the 1st, 5th, and 8th positions, respectively, for testing. After removing the edge A-F, the out-degrees of classes A, G, and H are all 0. Class A can be integrated first, and this step releases classes B, C, D, and F that can be integrated later; further analysis of classes B, C, D, and F shows that class F depends on class C. Therefore, it is appropriate for C to be tested before class F to avoid constructing redundant test stubs and reduce the stubbing cost. Then, class E needs to be integrated after classes F, and classes I, J, K, and L needs to be integrated after class E. This order conforms to a condition that prioritizes the integration testing of important classes while keeping the stubbing cost as low as possible. Other CITOs may also satisfy this situation, and this paper aims to find the optimal CITO that meets this need.

4. Approach

The core idea of our algorithm is to generate CITOs and reduce the stubbing cost while considering the importance of classes. Therefore, we redesign the Q-learning algorithm in RL to direct the agent to explore outcomes that match our constraint conditions. Section 4.1 describes the fundamentals of RL. Sections 4.2 and 4.3 introduce the calculation process of the stubbing complexity and the class importance separately. Details on devising reward functions follow in Section 4.4 and a sample program is shown in Section 4.5.

4.1. Reinforcement learning

Reinforcement learning (RL) is an important branch of machine learning, and it is based on the Markov decision process (MDP) that guides the agent to learn the best solution in interactions with the environment (Sutton and Barto, 2018). First, the agent adopts a tentative action a_t at time t, and the current state of the environment is s_t. Then, the agent can get a feedback reward r for choosing this action according to the mapping between states and actions. If the feedback is positive, the trend of taking this action will be strengthened; otherwise, it will be weakened. After that, the environment turns into the next state s_{t+1}, and the agent continues to learn. Finally, the agent stops exploring after it has learned to achieve the maximum profit through interactions with the environment by adopting tentative actions.

Assuming that a program has n classes, for each state, there are n actions that can be selected. Different actions map to different states, which means that n actions can be selected for each state, corresponding to n possible next states, and the possible maximum number of states can be $(n^{n+1}-1)/(n-1)$. Observing Fig. 2 (Sutton and Barto, 2018), suppose that the agent has visited $n+1$ states in total from the first state to the fth state and n actions have been selected. The fth state s_f can be considered as the final state. In this procedure, the state transition function is shown in Eq. (1) (Czibula et al., 2018), where $\delta(s_i)$ represents the union of all possible states in the ith group, and $s' \in \delta(s)$ is called a neighbor of s.

$$\delta(s_i) = \bigcup_{k=1}^{n} \{s_{n\times i-n+1+k}\} \quad \forall i \in \left\{1, \ldots, \frac{n^n-1}{n-1}\right\} \quad \forall k \in \{1, \ldots, n-1\} \tag{1}$$

We use $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_n)$ to represent the state path from the initial state to the final state s_f. In this case, $\sigma_0 = s_1$, and $\sigma_{k+1} \in \delta(\sigma_k)\ \forall \{k \in 1, \ldots, n-1\}$. The order of n actions performed by the agent according to the path can be expressed as a_σ, that is, the action history corresponding to the state path. If the path does not contain repeated actions, it can be considered that a_σ is an alternative class order to be integrated and tested.

The reward and punishment mechanism in RL is the core of guiding the exploration of the agent (Sutton and Barto, 2018). When any situation that does not meet the requirements occurs in the learning process, the environment will give the agent a minimum reward, which can be seen as a penalty value. For example, if any action class appears twice, this path will be given a minimum reward $-\infty$, which means the agent should avoid this situation when continuing to explore. When the final state is reached and no duplicate class appears, it can be considered that an alternative action path has been found, and its overall stubbing cost would be calculated. If the stubbing cost of the currently obtained CITO is less than that of all the previously obtained orders, this currently obtained CITO will get a higher reward. Integrating in the order of this action history can minimize the overall stubbing complexity of the generated CITO and save the stubbing cost to the greatest extent.

Because our next research aims to serve practical engineering, we did not use the RL framework but built the network directly.

And this paper does not emphasize the RL method's theoretical research but explores its application to generating CITOs. Q-learning is one of the simplest and most practical RL techniques, and it is envisioned as a promising solution (Cui et al., 2022). Therefore, this paper adopts the Q-learning algorithm in RL (Sutton and Barto, 2018) to address the CITO problem as preliminary exploratory work.

The Q-learning algorithm aims to obtain the optimal Q-value by solving the Bellman equation (Gass and Fu, 2013), which requires selecting the optimal behavior. Therefore, the strategy of Q-Learning is to choose the behavior with the highest Q-value in the optimal state, and through incremental improvements, the algorithm can achieve policy improvement and ensure that the updated Q-value is at least as good as the original value. This approach allows for learning from any state and ultimately results in convergence to the optimal value function, which aligns with our objective.

As a RL algorithm based on empirical learning, the Q-learning algorithm can incorporate the idea of dynamic programming by using the Bellman equation (Gass and Fu, 2013) to estimate the optimal Q-value function. In this paper, we use the Q-learning algorithm to model the problem of generating the optimal CITO. Specifically, the state space is defined as the set of all possible states that an agent can be in, which comprises states after selecting both tested and untested classes. The action space, on the other hand, refers to the set of possible combinations of classes to be tested next. It represents the available choices for selecting which classes to include in the next testing round. The reward function is defined as the reduction in testing cost achieved by testing a particular combination of classes. The Q-value function is used to estimate the optimal policy for selecting the next combination of classes to be tested. The Q-value is updated using the Bellman equation, which incorporates the immediate reward obtained by testing a particular combination of classes and the expected future rewards based on the next state. The calculation process is shown in Eq. (2), where α represents the learning rate, r represents the reward obtained by testing a particular combination of classes, and γ represents the discount factor. By iteratively updating the Q-value function, the Q-learning algorithm can learn the optimal order for testing classes and generate a test sequence that minimizes the overall testing time and resources.

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma maxQ(s',a') - Q(s,a)) \tag{2}$$

The improved Q-learning algorithm (Czibula et al., 2018) in this paper trains the agent as follows.

Algorithm. The Q-learning algorithm

Input: action a, learning rate α, discount factor γ
Output: an action order
1. repeat(for each episode):
2. initialize s;
3. repeat(for each step of the episode):
4. select action a using policy $\epsilon - Greedy$;
5. perform action a, get reward r, get state s';
6. update $Q - values$:
7. $Q(s,a) = Q(s,a) + \alpha(r + \gamma maxQ(s',a') - Q(s,a))$
8. $s \leftarrow s'$;
9. until s is terminal;
10. end

"One of the challenges that arise in reinforcement learning, and not in other kinds of learning, is the trade-off between exploration and exploitation". Sutton and Barto (2018) To prevent the agent from falling into the problem of local optima, we choose to increase the proportion of exploration and change the ϵ-Greedy mechanism (Czibula et al., 2018) as follows:

(1) Action a corresponding to the maximum Q-value is selected with the probability of $1 - \epsilon$.

(2) Action a corresponding to the smallest ratio n_1/n_2 is selected with the probability of ϵ^2, and n_1 represents the number of stubs required to add action a to the order while n_2 represents the number of stubs that can be avoided from being constructed.

(3) Randomly select action.

If an infinite number of state–action pairs are explored, the Q-value learned by the agent will converge to the optimal value (Watkins and Dayan, 1992). Accordingly, the action history associated with the path learned by the agent will converge to a certain extent, and the optimal CITO will be obtained.

4.2. The stubbing complexity

In this paper, the stubbing complexity is measured by combining the inter-class attribute coupling and method coupling (Briand et al., 2002).

Attribute coupling (Briand et al., 2002): If class i (the source class) depends on class j (the target class), there are references to locally declared attributes of class j in the declaration list of certain methods of class i, or there are instances of class j with pointers in class i, then classes i and j have attribute coupling. Attribute coupling generally includes the following three types: (1) class i accesses the member variable (attribute) of class j; (2) class i takes instances or attributes of class j as its method parameters; (3) class i takes an instance or attribute of class j as the return value of its method.

Method coupling (Briand et al., 2002): If class i (the source class) depends on class j (the target class) and class i calls a method of class j, then classes i and j have method coupling. There are three cases of method coupling: (1) calling the constructor of class j in class i creates an instance of class j; (2) a statement in class i calls a method of class j without a return value; (3) a statement in class i calls a method of class j and assigns the return value of the method to a member variable in class i.

The stubbing complexity is denoted by $SCplx(i,j)$, which represents the stubbing complexity of the stub constructed for class i to simulate the behavior of class j. We use $A(i,j)$ to represent the complexity of attribute coupling between classes i and j, and $M(i,j)$ to represent the complexity of method coupling. Before calculating $SCplx(i,j)$, both $A(i,j)$ and $M(i,j)$ are normalized to ensure their values fall between 0 and 1. The normalization process is shown in Eqs. (3) and (4).

$$\overline{A(i,j)} = \frac{A(i,j) - A_{min}}{A_{max} - A_{min}} \tag{3}$$

$$\overline{M(i,j)} = \frac{M(i,j) - M_{min}}{M_{max} - M_{min}} \tag{4}$$

The weights W_A and W_M are respectively assigned to attribute and method complexity to calculate the stubbing complexity $SCplx(i,j)$, which is shown in Eq. (5) (Briand et al., 2002):

$$SCplx(i,j) = \left[W_A \times \overline{A(i,j)}^2 + W_M \times \overline{M(i,j)}^2 \right]^{\frac{1}{2}} \tag{5}$$

The higher the value of $SCplx(i,j)$, the greater the stubbing cost. Typical approaches set the same weight when measuring stubbing complexity, which is not accurate enough (Briand et al., 2002; Jiang et al., 2021). There are many weighting methods: the entropy weighting method, the analytic hierarchy process method, the Delphi method, the weighted least squares method, principle element analysis, and multiple objective programming, etc. He et al. (2016). Because of its outstanding performance, the entropy weighting method is an important and widely-used method to determine the weight of different indicators (He et al., 2016). Therefore, we use the entropy weight method (Cheng, 2010) to calculate the weights of the attribute

and method complexity to make the measurement more accurate, and the calculation steps are as follows:

- **Step1:** Normalizing indicators. First, we normalize the $A(i,j)$ and $M(i,j)$ to make the results between 0 and 1. Then, the complexity of attribute coupling is denoted as $\overline{A(i,j)}$, and the complexity of method coupling is denoted as $\overline{M(i,j)}$. The processes of normalization of the two indicators are shown in Eqs. (3) and (4).
- **Step2:** Establishing an evaluation matrix. Suppose there is a system that contains m classes, and we use the notation "R" to denote a matrix of size $m \times 2$, where the first column corresponds to the attribute complexity of m classes and the second column corresponds to the method complexity of m classes. For example, r_{i1} represents the ith class's attribute complexity, and r_{i2} represents the ith class's method complexity. The calculation process is given in Eq. (6).

$$R = (r_{ij})_{m\times 2}(i = 1, 2, \ldots, m; j = 1, 2) \tag{6}$$

$$R = \begin{bmatrix} r_{11} & r_{12} \\ r_{21} & r_{22} \\ \cdots & \cdots \\ r_{m1} & r_{m2} \end{bmatrix}$$

- **Step3:** Calculating the information entropy. Before calculating the information entropy e_j, we need to measure the proportion of evaluation values. P_{ij} represents the proportion of the evaluation value of class i corresponding to the jth indicator, as shown in Eq. (7). Then we calculate the e_j for the jth indicator according to the obtained P_{ij}, as it shows in Eq. (8), where K is a constant and can be calculated by Eq. (9).

$$p_{ij} = \frac{r_{ij}}{\sum_{i=1}^{m} r_{ij}}(i = 1, 2, \ldots, m; j = 1, 2) \tag{7}$$

$$e_j = -K\sum_{i=1}^{m} p_{ij} \ln p_{ij}(i = 1, 2, \ldots, m; j = 1, 2) \tag{8}$$

$$K = \frac{1}{\ln m} \tag{9}$$

- **Step4:** Calculating the weights. We denote the weight of the jth indicator as W_j, where $j = 1$ represents the weight of attribute complexity (W_A in Eq. (5)), and $j = 2$ represents the weight of method complexity (W_M in Eq. (5)). The calculation is shown in Eq. (10):

$$W_j = \frac{1 - e_j}{\sum_{j=1}^{2}(1 - e_j)}(j = 1, 2) \tag{10}$$

After obtaining the weights of attribute and method complexity, the stubbing complexity $SCplx(i,j)$ is calculated by applying Eq. (5). We use o to represent an entire CITO, $OCplx(o)$ to represent the overall stubbing complexity, and N to represent the set of stubs to be constructed. The overall stubbing complexity can be estimated with Eq. (11) (Briand et al., 2001).

$$OCplx(o) = \sum_{(i,j)\in N} SCplx(i,j) \tag{11}$$

4.3. The class importance

The HITS algorithm is based on the following assumptions (Kleinberg, 1999): a high-quality authority page will be pointed to by high-quality hub pages, and a high-quality hub page will point to high-quality authority pages. Specifically, the authority value of a page is determined by the hub value of other pages that point to it. If a page is pointed to by multiple pages with high hub values, then the page has a higher authority value. The hub value of a page is characterized by the authority value of the pages it points to. If a page points to multiple pages with high authority values, then the page has a high hub value.

Wang et al. (2016a) found that the metrics of the HITS algorithm and two characteristics that characterize class importance, the impact (IC) and complexity (CC) of classes, are very similar. Therefore, they proposed the Class HITS algorithm by replacing the page authority and hub values in the above theory with IC and CC, respectively, to calculate the class importance. IC represents the in-degree of the class, where the more classes that depend on the target class, the greater the influence of the target class. CC represents the out-degree of the class, where the more classes that the target class depends on, the higher the complexity of the target class.

In addition to the two external characteristics mentioned above, a class also possesses its own complexity, which is denoted by C. Previous studies found that weighted methods per class (WMC), response for a class (RFC), and coupling between object classes (CBO) are statistically significant for the error propensity of a class and would largely affect its complexity (Zhou and Leung, 2006). WMC represents the number of methods implemented within a class to provide an indication of the internal complexity of a class. RFC counts the methods of a class and the unique methods that are called from that class to provide a deeper understanding of the potential external dependencies and interactions between a class and other components (Zhou and Leung, 2006; Chidamber and Kemerer, 1991). In this paper, we extend the scope and combine the above two into the number of features per class (NOF) to simplify the computation. CBO represents the number of classes that have dependent relationships with the target class (Subramanyam and Krishnan, 2003). The calculation process of C is shown in Eq. (12):

$$C_i = W_{nof}\frac{NOF_i}{\sum_{k=1}^{m} NOF_k} + W_{cbo}\frac{CBO_i}{\sum_{k=1}^{m} CBO_k} \tag{12}$$

W_{nof} and W_{cbo} represent the weights of NOF and CBO respectively, and m is the number of classes. The sum of W_{nof} and W_{cbo} is 1. We use the entropy weight method again to calculate the weights of the two indicators. At this point, the two normalized indicators in **Step1** of the entropy weight method become NOF and CBO, and in **Step4**, W_j becomes W_{nof} when $j = 1$ and W_{cbo} when $j = 2$.

After obtaining C, Wang et al. (2016a) set it as the initial value of IC and CC. Since an edge in a digraph between classes i and j does not mean that 100% of class j's dependencies are linked to class i, here we use q_{ij} to represent the percentage of relevancy. $q_{ij} = call(i,j)/NOF_j$ $(1 \le j, j \le m)$, where $call(i,j)$ denotes the number of features that class i depends on in class j, and NOF_j denotes the number of features of class j. Next, the initial value of IC and CC are set as the value of C, where $IC(0) = (C_1, C_2, \ldots, C_m)^T$ and $CC(0) = (C_1, C_2, \ldots, C_m)^T$. Then, IC and CC are corrected according to Eqs. (13) and (14) (Wang et al., 2016a), where a_{ij} $(i = 1, 2, \ldots, m;\ j = 1, 2, \ldots, m)$ is used to determine whether there is an association edge between classes i and j, and $a_{ij} = 1$ means that there exists a directed edge. Subsequently, IC and CC are normalized using Eqs. (15) and (16).

$$IC_i'(k) = \sum_{j=1}^{m} q_{ij} \times a_{ij} \times CC_j'(k-1)(i = 1, 2, \ldots, m) \tag{13}$$

$$CC_i'(k) = \sum_{j=1}^{m} q_{ij} \times a_{ij} \times IC_j'(k)(i = 1, 2, \ldots, m) \tag{14}$$

$$IC_i(k) = \frac{IC_i'(k)}{\|IC'(k)\|} \tag{15}$$

$$CC_i(k) = \frac{CC_i'(k)}{\|CC'(k)\|} \tag{16}$$

In this calculation, *IC* is used to calculate *CC*, it is because a class's complexity is not only related to its own structural complexity, but also to the complexity and propagation impact of other classes it depends on. *CC* is then used to calculate *IC* because a class's propagation impact is not only related to the number of other classes that directly or indirectly call this class, but also to the complexity of these classes. The more complex and error-prone the classes that depend on a class are, the wider the range of affected classes and functional paths when an error occurs, and the more important the system functions that may be affected.

In this paper, we are inspired by the above method to initialize the values of *IC* and *CC* using *C* and to iterate these values. During the iteration process, we obtain the values of *IC* and *CC* for each class in each iteration and compare them with the results of the previous iteration. If the difference between two iterations is less than the preset threshold, denoted as *H*, we consider that these values have converged. The update process of the values of *IC* and *CC* is shown in Eqs. (17) and (18), where $j \rightarrow i$ represents there exists a dependency relationship between class *j* and class *i*. "$\sum$" in Eq. (17) means that if there is a dependency from class *i* to class *j*, we add the current CC value of class *i* to the position of class *j* in the *IC* array. The same applies in Eq. (18). The values of *IC* and *CC* will be normalized and after that, the differences in values from these two arrays will be added to calculate an error value. This error value will be accumulated in iterations. When the error value is less than the preset threshold *H*, we consider two sets of vectors in *IC* and *CC* tend to be stable. The calculation process of the preset threshold *H* is shown in Eq. (19), where *m* is the number of classes in a program.

$$IC_i = \sum_{j \rightarrow i} CC_j \tag{17}$$

$$CC_i = \sum_{i \rightarrow j} IC_j \tag{18}$$

$$H = 0.1 \times m \tag{19}$$

Finally, the importance of class T_i is obtained as Eq. (20).

$$T_i = 0.5 \times IC_i + 0.5 \times CC_i \tag{20}$$

4.4. The design of reward functions

In this section, to better integrate the class importance into the generation of CITOs, we conceive two reward functions that combine the class importance: one is to combine the class importance with the stubbing complexity, and the other is to combine the class importance with the test profit (Zhang et al., 2017).

4.4.1. Combining the class importance with the stubbing complexity

In the learning period of the agent, the state is represented by *s*, the action is represented by *a*, and the state path is represented by σ. The environment will reward the agent according to the reward function for choosing a certain action. In this paper, the state path of the agent from the initial state s_1 to the final state s_f is represented by $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_f)$, $\sigma_0 = s_1$, and $\sigma_{k+1} \in \delta(\sigma_k)\ \forall \{k \in 1, \ldots, n-1\}$. The order obtained by the agent performing *n* actions is denoted as a_σ, which is the action history corresponding to the state path. When the path does not contain any repetitive actions, a_σ can be regarded as an optional integration test order.

If i ($i < n$) actions are selected, the action history is represented by $a_{\sigma_0}, a_{\sigma_1}, \ldots, a_{\sigma_{i-1}}$. We use $Cplx_i$ to denote the stubbing complexity required for integrating class *i* into the current order. In other words, for each existing class in this order that has a dependency relationship with class *i*, the stubbing complexity of simulating this dependency is added to $Cplx_i$.

Before combining the class importance with the stubbing complexity to guide the agent in learning to minimize stubbing cost, we normalize $Cplx_i$ and the class importance T_i primarily to unify the values within the same range. The normalization processes are given in Eqs. (21) and (22). It should be noted that, in Eq. (21), $Cplx_{max}$ and $Cplx_{min}$ refer to the maximum and minimum values among the selected actions, and they do not represent the overall maximum and minimum values of CITOs. Subsequently, we calculate the reward function using Eq. (23).

$$\overline{Cplx_i} = \frac{Cplx_i - Cplx_{min}}{Cplx_{max} - Cplx_{min}} \tag{21}$$

$$\overline{T_i} = \frac{T_i - T_{i_{min}}}{T_{i_{max}} - T_{i_{min}}} \tag{22}$$

$$r(\sigma_i) = \begin{cases} -\infty \\ c \times (1 + \overline{T_i} - \overline{Cplx_i}) \\ Max \end{cases} \tag{23}$$

As shown in Eq. (23), there are three conditions for choosing different branches of the reward:

(1) If the path, which has *i* classes, explores an action that has already appeared in the path, such as selecting a class that already exists in the path, the environment will assign a very small value to the agent. This is the first condition of the reward, represented by negative infinity.

(2) If the path, which has *i* classes, is an alternative path, the environment will give the agent a positive value based on the importance of its classes and the stubbing complexity. This is the second condition of the reward.

(3) When the iteration and traversal times set in the system or the program results converge, the agent will give the current path a Max value if the overall stubbing complexity of the current path is the smallest among all alternative paths and all classes have been selected. This is the third condition of the reward, represented by Max.

4.4.2. Combining the class importance with the test profit

The multi-level feedback approach of Zhang et al. (2017) integrates classes based on the feedback information received by the program, rather than breaking cycles or searching for optimum. Besides minimizing the stubbing complexity efficiently, the mechanism of this method is kind of like the reward function of RL. Therefore, we designed a second reward function that combines the test profit of classes (Zhang et al., 2017) with the class importance, to explore whether it can further reduce the stubbing cost required for generating CITOs.

First, to get the test profit *NR*, we subtract the stubbing cost *TC* from the testing revenue *TR*. Assuming that the currently untested class *u* belongs to the set *Suntest*, the stubbing cost *TC* of class *i* selected by the agent is the complexity between classes *i* and *u*, and the calculation of *TC* is shown in Eq. (24).

Then, we use the complexity between classes *u* and *i* to represent the testing revenue for selecting class *i*. The calculation process is shown in Eq. (25).

The *NR* of choosing class *i* indicates that the stubbing cost that is avoided by choosing class *i* instead of other untested classes. It is obtained by subtracting *TC* from *TR*, as shown in

Table 1
The calculation process of the complexity of classes.

No.	Class name	R		P		C
0	Building	0.24	0.32	0.07	0.10	5.60
1	DoorClosedException	0.00	0.00	0.00	0.00	0.39
2	Elevator	1.00	0.62	0.31	0.19	21.39
3	ElevatorController	0.29	1.00	0.09	0.32	7.81
4	ElevatorFullException	0.00	0.00	0.00	0.00	0.39
5	ElevatorMovingException	0.00	0.00	0.00	0.00	0.39
6	ElevatorState	0.16	0.00	0.05	0.00	3.54
7	Floor	0.37	0.62	0.11	0.19	8.80
8	Logger	0.14	0.00	0.04	0.00	3.15
9	Person	0.76	0.57	0.23	0.18	16.59
10	PersonState	0.12	0.00	0.04	0.00	2.75
11	Simulator	0.18	0.05	0.05	0.02	4.01

Eq. (26) (Zhang et al., 2017).

$$TC_i = \sum_{u \epsilon Suntest} SCplx(i,u)(i \neq u) \tag{24}$$

$$TR_i = \sum_{u \epsilon Suntest} SCplx(u,i)(i \neq u) \tag{25}$$

$$NR_i = TR_i - TC_i \tag{26}$$

Next, we normalize the two indicators as shown in Eqs. (27) and (28).

$$\overline{T_i} = \frac{T_i - T_{min}}{T_{max} - T_{min}} \tag{27}$$

$$\overline{NR_i} = \frac{NR_i - NR_{min}}{NR_{max} - NR_{min}} \tag{28}$$

Finally, we change the reward function, which is shown in Eq. (29). The selection of reward branches in this section is the same as in Section 4.4.1. However, the factor for the second condition to obtain reward is allocated based on the test profit and class importance, as shown in Eq. (29). When the second branch condition is satisfied, the reward value received by the agent is higher if the class has higher importance or the test profit is greater.

$$r(\sigma_i) = \begin{cases} -\infty \\ c \times (\overline{T_i} + \overline{NR_i}) \\ Max \end{cases} \tag{29}$$

4.5. A sample program

We take the program *elevator* from SIR (Software-artifact Infrastructure Repository, http://sir.unl.edu/portal/index.html) as an example to show the measurement process of class importance and the generation process of CITOs by our proposed approach. It is an elevator scheduling algorithm implemented in Java, including 12 classes, 97 methods, 114 variables, and 934 lines of code.

4.5.1. Measurement of class importance

The calculation process of the complexity of classes is shown in Table 1. First, by static analysis, we get the *NOF* and *CBO*. After normalizing them, we use Eq. (6) to get a matrix R, where r_{ij} represents the values of *NOF* and *CBO*. Then, we use Eq. (7) to calculate the evaluation value and obtain the matrix P. Next, we get that the information entropy of *NOF* and *CBO* are 0.77 and 0.65 respectively based on Eqs. (8) and (9). According to Eq. (10), the weight of W_{nof} is 0.39, and the weight of W_{cbo} is 0.61. After that, the initial complexity C of each class can be obtained according to Eq. (12). Details of this calculation have been uploaded on GitHub (https://github.com/Dia-cat/CITO_CRL).

Table 2
The generation process of the importance of classes.

No.	Class name	IC_{last}	CC_{last}	T
0	Building	0.05	0.16	0.10
1	DoorClosedException	0.04	0.00	0.02
2	Elevator	0.15	0.21	0.18
3	ElevatorController	0.10	0.22	0.16
4	ElevatorFullException	0.04	0.00	0.02
5	ElevatorMovingException	0.04	0.00	0.02
6	ElevatorState	0.11	0.00	0.06
7	Floor	0.11	0.18	0.15
8	Logger	0.17	0.00	0.08
9	Person	0.12	0.18	0.15
10	PersonState	0.05	0.00	0.02
11	Simulator	0.00	0.06	0.03

After obtaining the initial complexity C for classes, the following steps are performed. First, the C of each class is used as the initial value for the two metrics, IC and CC. Then, the class dependencies are analyzed based on Eqs. (17) and (18) to update IC and CC. Specifically, the following steps are taken:

Updating the IC array: (1) For each class i, we iterate it through the dependency relationship matrix; (2) if there is a dependency from class i to class j, we add the current CC value of class i to the position of class j in the IC array; (3) by performing above steps, we calculate the new IC array.

Updating the CC array: (1) For each class i, we iterate it through the dependency relationship matrix; (2) if there is a dependency from class i to class j, we add the new IC value of class j to the position of class i in the CC array; (3) by performing above steps, we calculate the new CC array.

After obtaining the values of the two arrays, we obtain the sum of the new CC array and the new IC array, respectively, and apply normalization of each value in arrays. To define whether these values have converged, we compare normalized values from this iteration with the previous iteration and get the sum of the differences. This sum is accumulated in the error variable.

The error value is then checked against the threshold H, which is defined in Eq. (19). If the error value is greater than the threshold, a new iteration is performed. If the error value is smaller than or equal to the threshold, the values of the IC and CC arrays are outputted. In this case, we name the final obtained IC and CC arrays as IC_{last} and CC_{last}, respectively.

Finally, according to Eq. (20), a new array $T_i (i = 1, 2, \ldots, n)$ is created by taking the average of the corresponding values from the IC_{last} and CC_{last} arrays and summing them up. As shown in Table 2, this array T_i contains n values, representing the importance of the classes.

The classes in the *elevator* sorted in descending order of class importance are as follows: [2, 3, 9, 7, 0, 8, 6, 11, 10, 1, 4, 5]. In this order, class 2, *Elevator*, ranks first for three reasons: (1) Class 2 contains 52 features, which is the largest sum of the number of attributes and methods in the program; (2) class 2 depends on many classes. Statistically, it accessed attributes of other classes 6 times and called methods of other classes 33 times; (3) many classes depend on class 2. Class 2's attributes are accessed by other classes 8 times, and class 2's methods are invoked by other classes 46 times. In summary, class 2's structural complexity and propensity for error are relatively high because of the high number of in-degrees and out-degrees. Hence, it makes sense that class 2 has the greatest class importance.

4.5.2. Generation of the class integration test order

After static analysis, the class names and class numbers are obtained, as is shown in Tables 1 and 2. Accordingly, the inter-class

Cla.	0	1	2	3	4	5	6	7	8	9	10	11
0			(0, 2)	(1, 7)			(1, 0)	(2, 2)				
1												
2		(0, 2)		(2, 1)	(0, 1)	(0, 2)	(1, 2)		(1, 24)	(2, 1)		
3			(4, 25)				(1, 0)	(1, 15)	(1, 19)	(2, 2)		
4												
5												
6												
7			(2, 2)	(2, 2)					(1, 9)	(3, 9)		
8												
9	(2, 3)		(2, 6)					(1, 10)	(1, 4)		(1, 2)	
10												
11	(0, 2)									(0, 4)		

Fig. 3. Inter-class dependencies.

Table 3
Execution process of *elevator* (success).

Num.	ASP	ASB	AP	RV	NA	ES	CITO
1	0.12	(2)	8	1000	6	FALSE	[8]
2	0.21	(2)	6	1000	10	FALSE	[8, 6]
3	0.54	(2)	10	1000	9	FALSE	[8, 6, 10]
4	0.86	(1)	9	990	2	FALSE	[8, 6, 10, 9]
5	0.30	(2)	2	996	3	FALSE	[8, 6, 10, 9, 2]
6	0.21	(2)	3	996	1	FALSE	[8, 6, 10, 9, 2, 3]
7	0.46	(2)	1	1000	4	FALSE	[8, 6, 10, 9, 2, 3, 1]
8	0.22	(2)	4	1000	5	FALSE	[8, 6, 10, 9, 2, 3, 1, 4]
9	0.75	(3)	5	1000	7	FALSE	[8, 6, 10, 9, 2, 3, 1, 4, 5]
10	0.16	(2)	7	1000	11	FALSE	[8, 6, 10, 9, 2, 3, 1, 4, 5, 7]
11	0.02	(2)	11	1000	0	FALSE	[8, 6, 10, 9, 2, 3, 1, 4, 5, 7, 11]
12	0.39	(2)	0	10000		TRUE	[8, 6, 10, 9, 2, 3, 1, 4, 5, 7, 11,0]

Table 4
Execution process of *elevator* (failure).

Num.	ASP	ASB	AP	RV	NA	ES	CITO
1	0.77	(3)	6	1000	8	FALSE	[6]
2	0.30	(2)	8	1000	94	FALSE	[6, 8]
3	0.65	(3)	1	1000	1119	FALSE	[6, 8, 1]
4	0.69	(3)	4	1000	13422	FALSE	[6, 8, 1, 4]
5	0.97	(1)	6	(2147483648)	161060	TRUE	[6, 8, 1, 4, 6]

dependencies between classes are shown in Fig. 3, representing the attribute and method dependencies separately.

In Fig. 3, rows represent source classes, and columns represent target classes. The intersection value represents the dependencies between a source class and a target class. For example, $[0, 2] = (0, 2)$ in Fig. 3 means that the attributes number of the class 0 (*Building*) to class 2 (*ElevatorController*) is 0, and the methods number of the class 0 (*Building*) to the class 2 (*Elevator*) is 2, and so on. The blank space indicates no attribute or method dependency between the source and target classes.

When executing the improved Q-learning algorithm aforementioned, we should input parameter values first. Specifically, before each selection round, s must be initialized, and then actions are selected based on randomly generated action selection probabilities. After the agent gets the reward, the next state will be entered, and then the Q-values will be updated. A selection process that obtains a complete order by our proposed method is shown in Table 3, where *ASP* represents action selection probability, *ASB* represents action selection branch, which maps to the ϵ-Greedy mechanism in Section 4.1, *AP* represents action being performed, *RV* represents reward value, *NA* represents next action, and *ES* represents end sign. When the termination flag is *TRUE* and every class is selected, the environment will give the agent a reward, and this selection round is completed.

When a repeated selection of the same class occurs during the selection process, the environment will give the agent a lower reward. The termination state is set to *TRUE* to end the current round of selection, as shown in Table 4, where a negative value is indicated by the number enclosed in parentheses. Assuming that the number of execution rounds we set is two, the system selects the action order with the maximum reward value from the above two results, which is [8, 6, 10, 9, 2, 3, 1, 4, 5, 7, 11, 0]. This action order is also output as the optimal CITO obtained in this training.

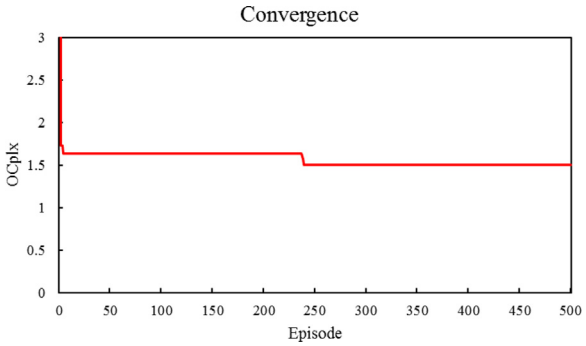


Fig. 4. The convergence of OCplx.

Table 5
Parameter settings.

Q-learning parameters	Values
Initial learning rate α	0.9
Discount factor γ	0.9
Number of episodes	$2{\cdot}10^5$
Exploration parameter ϵ	0.8 & ϵ'
c	100
Max	1000

Regarding *elevator*, it is estimated that about 10^{13} states are required to explore all possible combinations of classes. This estimation is based on the fact that the maximum number of states for a program with n classes can be calculated as $(n^{n+1} - 1)/(n-1)$. Thus, for *elevator* with 12 classes, the number of states is approximately $(12^{12+1} - 1)/(12 - 1) \approx 9.73 \times 10^{12}$, which is close to 10^{13} states. The convergence of the overall stubbing complexity (OCplx) learned by *elevator* is presented in Fig. 4. It is evident that the optimal CITO of *elevator* can be achieved after 239 training episodes, demonstrating the effectiveness of the proposed exploration mechanism in significantly reducing the convergence time.

5. Evaluation

To verify the effect of our methods, we conduct experiments on nine representative programs selected from SIR. We perform all experiments on the Eclipse platform, and our methods are implemented in Java.

5.1. Parameters design and subjects description

5.1.1. Parameters design

It is necessary to set parameters for the Q-learning algorithm before the experiment, as shown in Table 5.

We set the initial learning rate α to 0.9, and this α will linearly decrease during the training process until it becomes 0.01. The discount rate γ represents the influence of future rewards on the current state, and we set it to 0.9 to help the agent pay more attention to future payments. Since the maximum number of classes in the selected programs does not exceed 70, we train the agent 200,000 times each episode. Therefore, we may have the greatest extent to obtain the optimal result and prevent the experimental equipment from loading too high.

Table 6
Description of subject programs.

Program	Description	Classes	Deps	Cycles	LOC
elevator	Classic elevator scheduling algorithm	12	27	23	934
SPM	Security patrol monitoring system	19	72	1178	1198
ATM	ATM simulation system	21	67	30	1390
daisy	Network file system under Unix	23	36	4	1148
ANT	Integrated package management system based on Java platform	25	83	654	4093
email_spl	Email tool	39	63	38	2276
BCEL	Instrumentation tool for creating, analyzing and operating Java class files	45	294	416091	3033
DNS	Network domain name service system	61	276	16	6710
notepad_spl	Source code editor	65	142	227	2419

There are two ways to set the ϵ: (1) set it to 0.8; (2) obtain it dynamically according to Eq. (30).

$$\epsilon' = 1 - \epsilon + (2 \times \epsilon - 1) \times e^{\frac{-1}{times/5000}} \tag{30}$$

To dynamically adjust the action selection process, we use Eq. (30) to modify the parameter ϵ. In this equation, "*times*" represents the number of train times that we set to 200,000, and the original value of ϵ is 0.8. As the "*times*" value increases, the ϵ value gradually decreases. Research shows that after around 5000 iterations, the agent's learning process tends to converge (Hu, 2008), but the process is not smooth due to less exploration. To solve this, we use the ϵ-greedy algorithm with a dynamic parameter adjustment counter for a smoother learning process and better convergence performance. We experimented with eight different ϵ values between 0 and 1 and found that $\epsilon = 0.8$ gave the best results.

It has also been verified that for our selected programs, *elevator*, *SPM*, *ATM*, *daisy*, *ANT*, and *email_spl* can get better results when ϵ is 0.8. The other three programs: *BCEL*, *DNS*, and *notepad_spl* can obtain better results by using ϵ' (Ding et al., 2022). Setting the constant c to 100 is a more balanced way than other choices according to the value range of the stubbing complexity, and it is good to distinguish the total reward of obtained action orders.

Suppose that the agent obtains a CITO that has the least cost of all alternative paths, the agent will be assigned a *Max* reward, which is 1000. When all the training times are finished, we choose the action order with the largest overall reward as the optimal CITO.

5.1.2. Subjects description

We select nine open-source programs to evaluate the performance of our methods, and the information on all subjects is shown in Table 6. The five programs of *SPM*, *ATM*, *ANT*, *BCEL*, and *DNS* are from Briand et al. (2002). They are usually used as benchmark programs. The other four programs are obtained from SIR.

The columns in Table 6 respectively represent program name, description, number of classes, number of dependencies, number of cycles, and lines of code (*LOC*). For these programs, the number of classes is less than 70, the number of inter-class dependencies varies from 27 to 294, the number of cycles varies from 4 to 416091, and the number of code lines varies from 934 to 6710. To avoid errors caused by different program versions, we derive the information on programs from Soot, a Java program analysis framework. Therefore, some systems differ slightly from SIR or other documents.

Before running the programs, we calculate the class importance in each program and sort them to prepare for the later analysis. Table 7 contains the order in which the classes in each program are arranged in descending order of importance.

5.2. Research questions

Our experiments are designed to address the following four research questions:

RQ1: To which degree can class importance affect the generated CITOs?

Inter-class dependencies of programs are intricate. At present, there is almost no evaluation of class importance in various methods of generating CITOs. Therefore, we first compute the class importance and sort classes according to it. Then, for each program, we compare the differences between the CITOs generated with or without considering class importance to assess the impact of class importance on those orders.

RQ2: To which degree can class importance affect the stubbing cost of the generated CITOs?

If we confirm that class importance does have a certain impact on CITOs, further exploration is needed to figure out whether or to which degree class importance can affect stubbing cost. To answer this question, we compare the experimental results under three indicators: overall stubbing complexity, number of stubs, and runtime.

RQ3: What is the effect of the two reward functions designed in combination with class importance?

To compare the impact of the two methods: (1) the reward function that combines class importance with stubbing complexity (CITO_CRL); (2) the reward function that combines class importance with test profit (CITO_CRL') on the generated CITOs, we conduct experiments on the two methods separately and analyze the experimental results to assess the effects.

RQ4: What is the performance of CITO_CRL and CITO_CRL' compared with other methods?

To measure the pros and cons of the two proposed methods, we compare them with graph-based, search-based, and RL-based methods under three evaluation indicators: stubbing complexity, number of stubs, and runtime. These methods can also be classified into two types: those that consider class importance and those that do not. For the methods that consider class importance, we directly compared them with our method. For the methods that do not consider class importance, we carefully and precisely implemented those methods with their corresponding parameters as described in their original papers to make a fair comparison.

In this paper, we design our methods based on RL, and the data we obtain may not be the same every time. Considering the randomness, we execute each method 30 times and use the average of these results for analysis.

5.3. Experimental results

In this section, we conduct experiments and analyze the results to answer the four research questions.

5.3.1. Result analysis for RQ1: To which degree can class importance affect the generated CITOs?

After getting the ranks of classes in each program according to class importance, we conduct experiments and analyze the CITOs generated by 30 runs.

Table 8 shows the average positions of the top ten important classes in CITOs generated by running three methods, CITO_RL, CITO_CRL, and CITO_CRL', 30 times. CITO_RL is the method without considering class importance but only stubbing complexity. CITO_CRL is the method of considering class importance and stubbing complexity. CITO_CRL' is the method of combining class importance with test profit. To better control variables when

Table 7
Order of classes according to their importance from largest to the smallest for each program.

Program	Classes	Order of classes according to their importance from largest to the smallest
elevator	12	2, 3, 9, 7, 0, 8, 6, 11, 10, 1, 4, 5
SPM	19	8, 2, 15, 13, 4, 6, 7, 1, 12, 14, 0, 3, 5, 17, 16, 18, 11, 9, 10
ATM	21	9, 7, 8, 20, 13, 11, 12, 14, 19, 10, 0, 4, 2, 17, 18, 15, 16, 1, 3, 5, 6
daisy	23	2, 1, 14, 13, 16, 4, 17, 19, 3, 0, 8, 5, 11, 7, 22, 15, 6, 9, 12, 18, 20, 10, 21
ANT	25	15, 17, 2, 19, 20, 23, 18, 9, 21, 24, 1, 22, 12, 11, 5, 0, 16, 3, 6, 7, 4, 14, 8, 10, 13
email_spl	39	8, 26, 9, 5, 36, 10, 30, 32, 31, 24, 3, 17, 14, 19, 6, 28, 7, 33, 2, 23, 34, 29, 35, 27, 21, 13, 18, 25, 4, 38, 0, 15, 16, 1, 20, 22, 12, 37, 11
BCEL	45	22, 44, 23, 1, 17, 20, 34, 5, 29, 3, 18, 33, 32, 31, 13, 21, 19, 4, 6, 25, 28, 39, 27, 11, 12, 15, 10, 8, 24, 38, 30, 42, 43, 40, 14, 9, 36, 37, 7, 16, 26, 35, 0, 41, 2
DNS	61	20, 54, 45, 31, 11, 24, 49, 37, 50, 10, 7, 34, 6, 27, 19, 9, 59, 30, 38, 35, 36, 8, 46, 41, 17, 51, 28, 42, 13, 2, 53, 15, 55, 29, 3, 40, 1, 44, 47, 18, 22, 14, 60, 57, 23, 0, 56, 5, 32, 25, 4, 52, 48, 43, 12, 58, 33, 39, 21, 26, 16
notepad_spl	65	57, 21, 64, 17, 53, 49, 41, 45, 60, 30, 34, 54, 42, 46, 31, 35, 38, 51, 55, 39, 43, 47, 50, 28, 32, 36, 52, 56, 40, 44, 48, 27, 59, 29, 33, 37, 22, 62, 26, 63, 7, 10, 12, 19, 18, 61, 24, 23, 20, 11, 16, 25, 58, 14, 6, 0, 8, 3, 1, 5, 4, 15, 2, 13, 9

Table 8
Positions of important classes in the generated CITOs.

Program	Method	Im1	Im2	Im3	Im4	Im5	Im6	Im7	Im8	Im9	Im10	Count
elevator	CITO_RL	7	10	8	9	11	1	2	12	6	3	0
	CITO_CRL	7	10	8	9	11	1	2	12	6	3	0
	CITO_CRL'	7	10	8	9	11	1	2	12	6	3	0
SPM	CITO_RL	**4**	5	9	10	14	15	16	3	**7**	**8**	3
	CITO_CRL	**4**	5	9	10	14	15	16	**2**	8	9	2
	CITO_CRL'	5	**2**	9	10	14	15	16	3	**7**	**8**	3
ATM	CITO_RL	7	**3**	5	2	18	**15**	17	19	1	**8**	3
	CITO_CRL	7	4	5	2	18	16	17	**18**	1	9	1
	CITO_CRL'	7	4	5	2	18	**15**	17	**18**	1	**8**	3
daisy	CITO_RL	20	19	21	22	13	11	18	23	17	2	0
	CITO_CRL	20	19	21	22	13	**10**	18	23	17	2	1
	CITO_CRL'	20	19	21	22	13	**10**	18	23	17	2	1
ANT	CITO_RL	8	18	1	10	11	17	9	14	13	25	0
	CITO_CRL	8	18	1	10	11	17	9	14	13	25	0
	CITO_CRL'	**5**	18	1	**8**	**10**	17	**7**	**10**	13	**23**	6
email_spl	CITO_RL	13	9	10	**23**	**26**	2	**21**	**22**	19	26	4
	CITO_CRL	13	**7**	**9**	24	**26**	2	**21**	**22**	19	**25**	6
	CITO_CRL'	13	**7**	**9**	**23**	28	2	22	24	19	28	3
BCEL	CITO_RL	41	37	42	34	6	32	35	38	36	26	0
	CITO_CRL	41	**36**	42	37	6	32	**34**	38	**35**	26	3
	CITO_CRL'	41	37	42	**32**	**5**	32	35	38	36	**23**	3
DNS	CITO_RL	7	32	1	14	40	14	58	41	39	24	0
	CITO_CRL	7	31	1	**12**	**39**	**13**	57	40	38	**23**	4
	CITO_CRL'	7	**30**	1	**12**	**39**	**13**	**56**	**39**	**36**	**23**	8
notepad_spl	CITO_RL	13	5	**51**	**14**	59	54	48	51	**14**	37	3
	CITO_CRL	12	7	57	16	59	55	**47**	51	15	36	1
	CITO_CRL'	**7**	**4**	55	17	**55**	**51**	48	**48**	17	**35**	6

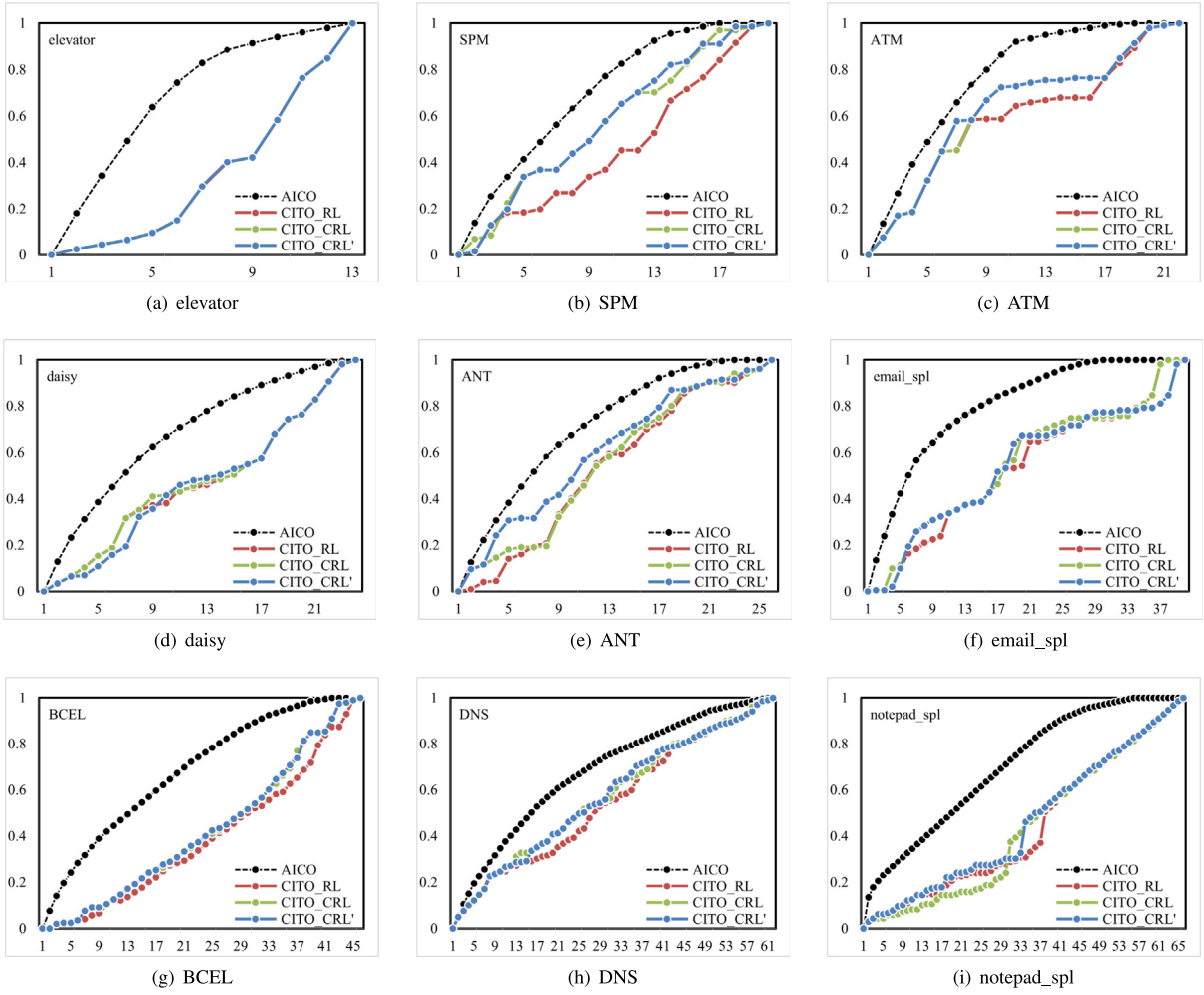
assessing the impact of class importance on CITOs in RQ1, we focused on comparing CITO_RL and CITO_CRL in this section. In this table, *Im* indicates the importance of a class, and *Im*1 represents the average ranking position of the class with the highest importance in the sequence. The importance decreases as the number *Im* increases. *Count* is the sum of positions that rank first in the comparison of three methods.

We take *SPM* as an example. The *Im*1 is the position of the most important class in *SPM*. Based on Table 7, the most important class in *SPM* is class 8. The 30 positions of class 8 in CITOs obtained through these three methods are averaged, and class 8 is located in the 4th, 4th, and 5th positions, respectively. The following positions can be deduced by analogy. For 9 programs that were chosen, a total of 90 important classes are counted in the experimental orders.

Among the 90 classes, CITO_RL, CITO_CRL, and CITO_CRL' have 13, 18, and 33 classes rank first in the comparison, respectively. It should be noted that if two comparison methods yield the same results, we double-count selections, and if all methods produce the same results, we ignore this comparison.

Accordingly, important classes are more likely to be put in prior positions by CITO_CRL and CITO_CRL' rather than CITO_RL. Meanwhile, considering the importance of the top ten classes is not enough to explain how the class importance can affect the integration test order, we calculate the cumulative result of the class importance in the CITOs as shown in Fig. 5. It should be noted that the "AICO" in Fig. 5 represents the accumulation of the importance of the class order obtained by sorting the classes in descending order of importance.

We select one of the 30 run results based on the cumulative class importance values at the middle position of the sequence to generate Fig. 5. To reduce the impact of outliers, the semi-sequence cumulative value of CITO we selected is the second largest in 30 results, rather than the largest. RQ1 focuses on the impact of class importance on the generated CITOs, while the difference between CITO_RL and CITO_CRL is only that the latter considers class importance. Therefore, prioritizing the comparison of these two methods is more in line with the principle of variable control.



(a) elevator (b) SPM (c) ATM

(d) daisy (e) ANT (f) email_spl

(g) BCEL (h) DNS (i) notepad_spl

Fig. 5. Cumulative results of the class importance in the generated CITOs, where AICO represents the accumulation of the importance of the class order obtained by sorting the classes in descending order of importance.

Compare CITO_RL (red line) and CITO_CRL (green line): There is no difference between the two in *elevator*; for *SPM*, the green line is significantly higher than the red line; for *ATM*, the green line is higher than the red line form point 8 to point 17; for *BCEL* and *DNS*, due to a large number of classes, the difference between the two lines is not significant, but it can still be seen that the green line has always been above the red line; and *notepad_spl* has a red line higher than the green line before the 30th point. The green line for the other three programs is slightly higher. Overall, when compared with CITO_RL, CITO_CRL demonstrates a superior ability to prioritize important classes in generating CITOs for *SPM*, *ATM*, *BCEL*, and *DNS*, while also holding a slight advantage in generating CITOs for *daisy*, *ANT*, and *email_spl*.

In addition, CITO_CRL also considers the importance of classes for training, and unlike the first two, it combines the test profit as another guiding information. CITO_CRL' is represented by a blue line, and it can be seen that except for *elevator*, the results obtained by the three programs are consistent, with little difference compared to *daisy*. All other programs indicate that the blue line accumulates significantly higher results in class importance than the red line. This indicates that considering the class importance, CITO_CRL' has significant advantages in prioritizing the integration of important classes.

Accordingly, considering class importance can affect the CITOs, and both our proposed methods are likely to give priority to the class with a higher importance in integration testing.

5.3.2. Result analysis for RQ2: To which degree can class importance affect the stubbing cost of the generated CITOs?

Prioritizing the integration of classes with high importance does not guarantee that the additional stubbing cost will not be added, which means that although considering class importance is important to avoid delayed discovery of program errors, this additional consideration may increase the stubbing cost of CITOs. Therefore, we compare the experimental results in terms of the overall stubbing complexity and the number of stubs.

For each program, we run the method 30 times and take the average to make the experimental results less random. Table 9 shows these average values obtained by CITO_RL and CITO_CRL. Both these two methods use stubbing complexity as guiding information for the agent exploration, while the difference lies in the latter considering the class importance. Comprising these two methods meets the requirements for variable control in experiments.

The average results obtained by different programs using CITO_RL and CITO_CRL are shown in Table 9, where we denote *OCplx*, *StubsG*, *StubsS*, and *RT*(*ms*) as the overall stubbing complexity, the number of generic stubs, the number of specific stubs,

Table 9
Comparison of results between CITO_RL and CITO_CRL.

Program	CITO_RL				CITO_CRL			
	OCplx	StubsG	StubsS	RT(ms)	OCplx	StubsG	StubsS	RT(ms)
elevator	1.50	4	5	**2540**	1.50	4	5	3095
SPM	3.62	13	21	**5259**	**3.48**	13	21	6984
ATM	**2.31**	15	15	**5257**	2.32	**14**	**14**	7553
daisy	0.09	3	3	**6305**	0.09	3	3	8276
ANT	**1.45**	8	13	**7499**	1.48	8	13	11435
email_spl	**0.32**	6	7	**13182**	0.33	6	7	18658
BCEL	**5.85**	16	**73**	**15485**	6.19	16	74	99143
DNS	**1.90**	10	12	**23437**	2.14	10	**11**	197282
notepad_spl	4.76	44	47	**38905**	**4.31**	44	47	211084

and the runtime (in milliseconds) of each method, respectively. Excluding equal values, the lower values are marked as bold.

In terms of *OCplx*, five systems can be implemented through CITO_RL obtains the lowest value, and two systems with the most classes can use CITO_CRL to obtain the lowest value. Therefore, we cannot rule out that when conducting more experiments with larger class numbers, CITO_CRL can generate more advantages. The other two systems have the same results in both methods.

In terms of the number of stubs, CITO_RL has one program (specific stubs of *BCEL*) to obtain the minimum value, while CITO_CRL has three programs (specific stubs of *ATM* and *DNS*, and generic stubs of *ATM*) to obtain the minimum value. In terms of runtime, CITO_RL requires less time than CITO_CRL when not considering the factor of class importance. However, the way to obtain dynamic ϵ of *BCEL*, *DNS*, and *notepad_spl* also increases the time consumption for both methods.

The standard deviation of the results obtained by running the three methods 30 times is shown in Table 10. The standard deviation data for the different programs and algorithms show that the standard deviation values for *OCplx* are generally low, indicating relatively consistent complexity measurements across the programs and algorithms. The standard deviation values for *StubsG* and *StubsS* tend to be relatively higher compared to *OCplx*, suggesting greater variation in the measurement of stubs. Additionally, the standard deviation values for *RT*(*ms*) vary significantly across different programs and algorithms.

Based on the analysis of the above data, we can conclude that after considering the importance of classes, CITO_CRL has no advantage in overall stubbing complexity; there are slight differences in the number of stubs between the two methods, but in most cases, the number of stubs required for both methods is still equivalent. It is obvious that considering the class importance not only affects CITO, but also may increase the overall stubbing complexity. Therefore, while ensuring the priority of important classes during the testing process, it is crucial to explore alternative methods to minimize the stubbing costs.

5.3.3. Result analysis for RQ3: What is the effect of the two reward functions designed in combination with class importance?

To reduce the stubbing cost of methods that consider class importance, we further studied another RL method considering the class importance, CITO_CRL'. Its difference with CITO_CRL is that it uses the test profit instead of the stubbing complexity to guide intelligent agent exploration.

The accumulations of class importance in the CITOs generated by these two methods are shown in Fig. 5. For *elevator*, the results have no difference. For other programs, the marked line of CITO_CRL' is higher or slightly higher than CITO_CRL. Therefore, we can conclude that CITO_CRL' performs better than CITO_CRL in integrating important classes.

We also compare CITO_CRL and CITO_CRL' from *OCplx*, *StubsG*, *StubsS*, and *RT*(*ms*). The results obtained by the two methods are shown in Table 11. And excluding equal values, the lower values are marked as bold.

In terms of *OCplx*, CITO_CRL has one program with the lowest value, while CITO_CRL' has six. CITO_CRL' has an advantage. In terms of the number of stubs, CITO_CRL has the five lowest values, and CITO_CRL' only has the two lowest values of specific stubs. But the differences between these two methods are all less than or equal to three. Regarding *RT*(*ms*), the results of the first seven programs are close. But for the latter three larger programs, the *RT*s got by CITO_CRL' are better, especially for *notepad_spl* which cost 0.56 times shorter than CITO_CRL.

Sometimes, the two methods require the same number of stubs but differ in overall stubbing complexity, as shown in Table 12. In this table, "$Stubs_{Info}$" indicates the information of specific stubs constructed, and "(*i*)->*j*" represents a specific stub simulating class *i* for class *j*. For *ATM* and *daisy*, the two methods have required the same number of specific stubs but exhibit different stubbing complexity. This discrepancy arises from the fact that the same number of stubs may be constructed for simulating different dependencies.

For *ANT*, *email_spl*, *DNS*, and *notepad_spl*, although CITO_CRL' has more stubs constructed than CITO_CRL, its overall stubbing complexity is equal or lower. To further explain this situation, Table 13 shows the differences in the stubs constructed by CITO_CRL and CITO_CRL' for *ANT*. The first and second columns show the information of stubs constructed by the two methods, where "19->1-(7, 1)" represents that a specific stub simulating class 19 for class 1 is constructed, and the attribute and method complexity of this stub are seven and one, respectively. The third column describes the changes caused by constructing different stubs. Although CITO_CRL' constructs two more specific stubs than CITO_CRL, the attribute complexity is reduced by nine, and the method complexity is increased by one. These measures reduce the overall stubbing complexity by 0.13.

In summary, from the perspective of generated CITOs, the impact of CITO_CRL and CITO_CRL' is relatively equal, and CITO_CRL' performs better in programs with more dependencies. From the perspective of stubbing cost, CITO_CRL' has an advantage in the overall stubbing complexity for having the minimum value in more than half of the programs. Therefore, CITO_CRL' can be more conducive than CITO_CRL to reduce the stubbing cost as much as possible while giving priority to important classes in integration testing.

5.3.4. Result analysis for RQ4: What is the performance of CITO_CRL and CITO_CRL' compared with other methods?

In RQ3, we only compared two methods based on RL considering class importance, which is unconvincing. There are few methods considering this factor, except methods of Zhao et al. (2015) and Wang et al. (2016a). And both these methods evaluated the stubbing cost by the stubbing complexity.

For *DNS*, the overall stubbing complexity is 1.47 by the method of Zhao et al. (2015), and for *ANT*, *BCEL*, and *DNS*, the overall stubbing complexity is 3.26, 5.87, and 1.47 respectively by the method of Wang et al. (2016a). Compared with the two methods mentioned above, both CITO_CRL and CITO_CRL' have advantages on *ANT*, CITO_CRL' also has better performance on *BCEL*, while the method of Wang et al. (2016a) performs better on *DNS*.

To evaluate the performance of the proposed methods more comprehensively, we select several typical graph-based methods and search-based methods for comparison. The results of overall stubbing complexity are shown in Table 14. We select three graph-based methods that are classic and widely used as comparison objects: the methods of Le Traon et al. (2000), Tai and Daniels (1997), and Briand et al. (2003). We also select three search-based algorithms that are easy to implement and can achieve

Table 10
Standard deviation of CITO_RL, CITO_CRL, and CITO_CRL'.

Program	CITO_RL				CITO_CRL				CITO_CRL'			
	OCplx	StubsG	StubsS	RT(ms)	OCplx	StubsG	StubsS	RT(ms)	OCplx	StubsG	StubsS	RT(ms)
elevator	0.00	0.00	0.00	137.33	0.00	0.00	0.00	26.74	0.00	0.00	0.00	49.01
SPM	0.16	0.68	1.30	398.44	0.23	0.73	1.01	301.64	0.12	0.71	0.84	982.01
ATM	0.04	0.56	0.80	438.92	0.05	0.70	0.88	166.58	0.00	0.18	0.40	702.07
daisy	0.01	0.00	0.00	490.23	0.01	0.00	0.00	501.53	0.01	0.00	0.00	851.54
ANT	0.04	0.60	0.68	357.30	0.00	0.18	0.18	343.19	0.02	1.04	1.52	518.43
email_spl	0.00	0.25	0.18	843.22	0.01	0.43	0.45	764.17	0.01	0.46	0.67	1665.85
BCEL	0.15	0.77	0.84	937.76	0.03	0.26	0.26	286.22	0.10	0.92	0.60	3521.39
DNS	0.18	0.73	2.23	2864.26	0.05	0.30	0.00	7775.87	0.22	1.01	1.29	47835.31
notepad_spl	0.31	0.85	0.70	1378.41	0.55	1.49	1.09	1234.32	0.41	0.81	12.54	17240.35

Table 11
Comparison of results between CITO_CRL and CITO_CRL'.

Program	CITO_CRL				CITO_CRL'			
	OCplx	StubsG	StubsS	RT(ms)	OCplx	StubsG	StubsS	RT(ms)
elevator	1.50	4	5	**3095**	1.50	4	5	3599
SPM	3.48	13	21	**6984**	**3.00**	13	**20**	9170
ATM	2.32	14	14	**7553**	**2.28**	14	14	9394
daisy	**0.09**	3	3	**8276**	0.10	3	3	9950
ANT	1.48	**8**	**13**	**11435**	**1.45**	9	15	12816
email_spl	0.33	6	**7**	**18658**	0.33	6	8	21556
BCEL	6.19	16	74	99143	**5.44**	16	**72**	**94659**
DNS	2.14	10	**11**	197282	**2.12**	10	12	**168858**
notepad_spl	4.31	44	**47**	211084	**4.20**	44	50	**92303**

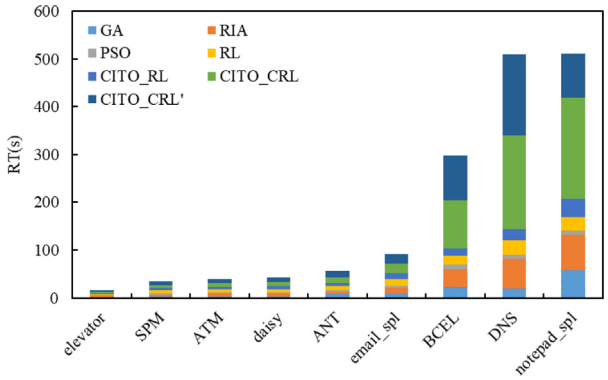


Fig. 6. Comparison of RT(s) with other methods.

satisfactory performance: the GA method proposed by Briand et al. (2002), the RIA method proposed by Wang et al. (2011), and the PSO method proposed by Zhang et al. (2018a). In addition, we reproduce the RL method of Czibula et al. (2018), which is denoted as "RL" in Table 14. Our previous method "CITO_RL" is added to the comparison, too. It should be noted that none of the eight methods considered class importance. We highlighted the best values in bold, even the equal values.

In terms of OCplx, among the three graph-based methods, only the method of Briand et al. (2003) has an advantage in *DNS*. Among the three search-based methods, RIA has a better effect on *notepad_spl*. The optimal results for other programs are produced in our RL-based approaches. CITO_RL that does not consider class importance has four optimal results, while CITO_CRL and CITO_CRL' that consider class importance have two and five optimal results, respectively. Although additional consideration has been given to class importance, the advantages of our methods are still obvious compared with the graph-based, search-based, and Czibula et al.'s RL-based (Czibula et al., 2018) methods.

For *DNS*, if not considering class importance, the graph-based method of Briand et al. (2003) performs best. When considering it, the method of Wang et al. (2016a) performs best. The reason for this phenomenon is that *DNS* has a lot of weak dependencies. Among its 276 dependencies, there are 234 weak dependencies. The two graph-based methods are integrated with the method of Tai and Daniels (1997), which aims to break cycles with the most correlations. Meanwhile, our methods pursue the lower current stubbing complexity and are likely to break strong dependencies such as aggregation and inheritance during the training process. Therefore, for *DNS*, our methods are worse than the two graph-based methods.

For *notepad_spl*, RIA performs the best. *Notepad_spl* has a lot of relationships to focus on one or two classes, such as the 21st class *Smashed.Actions* and the 57th class *Smashed.Notepad*. These classes with more in-degrees rank higher in CITOs generated by RIA, and they can be tested first, which decreases the overall stubbing complexity.

Table 15 shows the number of specific stubs required by each method, and we highlight the optimal results in Table 15 in bold, like Table 14. In terms of *Stubs*, the RL method of Czibula et al. (2018) dominates on four programs while the method of Briand et al. (2003), CITO_RL, and CITO_CRL all dominate on three programs. The optimization goal of the RL method (Czibula et al., 2018) is to minimize the number of stubs (both generic and specific), so this result is not surprising. The learning ability of RL is very strong, so our methods can have a significant advantage in stubbing complexity.

In terms of runtime, the consumption of the graph-based methods is mainly in the operation of breaking cycles, and only a one-time run could obtain the CITO, so the time consumed by graph-based methods is mostly less than 100 ms. The key to the RL-based methods lies in the agent's exploration based on the reward function, and there is no behavior of breaking cycles during the training process. Therefore, for runtime, the RL-based and graph-based methods do not belong to the same order of magnitude, and they should not be compared together. As shown in Fig. 6, where the *y*-axis is seconds (s), when compared with the three search-based methods, RL-based methods cost more time, which is because they require 200,000 training times for each result, while the initial population of the search-based methods is 100, and they could get CITOs after hundreds of iterations. Besides, both CITO_CRL and CITO_CRL' additionally considered the class importance, so this result is reasonable, and next, we will make further optimizations to shorten the runtime in future work.

Because of the limited space, we only present here the total distribution of OCplx of the nine programs in Fig. 7. We have made more statistical analyses for the 30 executions, which have been uploaded onto https://github.com/Dia-cat/CITO_CRL.

We apply the Mann–Whitney U test (Mann and Whitney, 1947) to evaluate whether significant differences exist in the overall stubbing complexity between our methods and other methods. The Mann–Whitney U test is a non-parametric test relative to two independent samples. Table 16 shows the p-values obtained by comparing different methods, respectively. When

Table 12
Two methods require the same number of stubs but different stubbing complexity.

Program	Method	Order	OCplx	Stubs	$Stubs_{Info}$
ATM	CITO_CRL	[19, 20, 15, 7, 8, 16, 9, 10, 1, 0, 2, 3, 13, 4, 5, 18, 6, 11, 12, 14, 17]	2.26	15	(0, 1, 2, 3, 4, 5, 6, 8, 9, 15)->7, (10, 11, 12, 13, 14)->9
	CITO_CRL'	[19, 20, 15, 7, 8, 16, 9, 0, 17, 10, 18, 1, 2, 3, 4, 5, 6, 11, 12, 13, 14]	2.16	15	(0, 1, 2, 3, 4, 5, 6, 8, 9)->7, (8)->17, (10, 11, 12, 13, 14)->9
daisy	CITO_CRL	[7, 0, 8, 15, 6, 10, 9, 4, 12, 18, 20, 21, 16, 5, 11, 22, 3, 17, 1, 2, 14, 13, 19]	0.13	3	(17)->16, (1)->11, (13)->14
	CITO_CRL'	[7, 0, 8, 15, 6, 10, 12, 18, 20, 21, 16, 9, 4, 5, 11, 22, 3, 17, 1, 2, 13, 14, 19]	0.16	3	(17)->16, (1)->11, (14)->13

Table 13
Differences of stubs for *ANT*.

Stubs created by CITO_CRL	Stubs created by CITO_CRL'	Difference
19->1–(7, 1)	4->5–(0, 0)	ACplx: −9
19->20–(7, 0)	18->19–(5, 2)	MCplx: +1
	20->19–(0, 0)	Stubs: +2
	22->19–(0, 0)	TCplx: −0.13

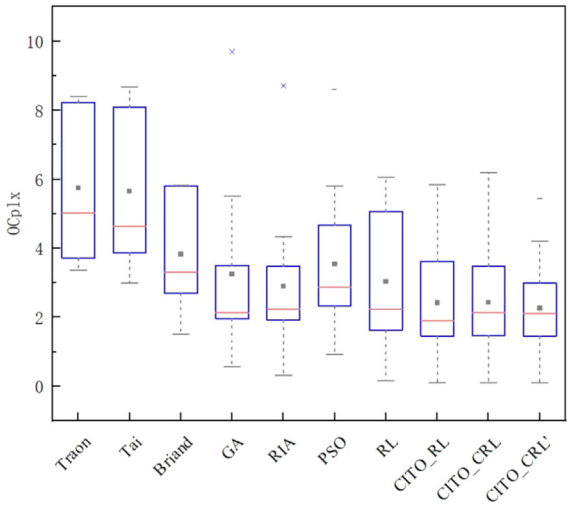


Fig. 7. Distribution of OCplx obtained by all comparison methods.

the p-value is less than 0.05, we can consider that there is a significant difference between Method1 and Method2. The results with a p-value not less than 0.05 in the table are all marked red. We have highlighted a portion of the data in bold, implying that Method1 (our method) has a significant advantage on these programs.

Effect size is an important consideration in interpreting the results of statistical tests. In the context of the Mann–Whitney U test, the effect size indicator, often estimated by Cohen's d (Fritz et al., 2012), is calculated by the following equation: $(2 \times Z)/\sqrt{n_1 + n_2}$, where "$n_1$" and "$n_2$" represent the sample sizes of the two groups. In our case, since we ran the program 30 times separately, both "n_1" and "n_2" are 30. The "Z" value is the standardized test statistic, obtained from the non-parametric test.

We present the effect size for the Mann–Whitney U test in Table 17. According to Tables 16 and 17, for CITO_CRL and CITO_CRL', there are six and four results that are greater than 0.05 when compared with other methods. In this case, we reject the null hypothesis and admit that Method1 and Method2 are not significantly different. 75% and 83% of the results in the upper and lower parts of the table are marked in bold, respectively, which means CITO_CRL and CITO_CRL' have significant advantages in most cases.

Based on the experimental results, we can get a preliminary summary of the applicability of different methods: Briand's graph-based method (Briand et al., 2003) is effective for situations with a large number of weak dependencies forming a cycle, such as DNS. RIA (Wang et al., 2011) performs better on programs with a large number of relationships concentrated on one or two classes, like notepad_spl. And our proposed RL methods perform better in most cases. However, our conclusions are based on a preliminary analysis, and we cannot ensure that they have universality for programs of different categories. In the future, we plan to conduct more experiments and expand the sample size to further investigate this issue.

From these results, we can conclude that both CITO_CRL and CITO_CRL' have more significant advantages than other methods. And for most programs, CITO_CRL' can generate CITOs with lower overall stubbing complexity while prioritizing important classes.

5.4. Threats to validity

In this section, we summarize the potential factors that affect the experimental results of our methods and analyze the validity from both external and internal aspects.

External threats. External threats relate to the generalizability of our results. We analyzed nine programs of varying sizes and functions, including small and medium programs with different numbers of classes from 12 to 65. These programs cover a diverse range of domains, including programming tools, social media, and security systems. Each of them has been widely recognized by researchers and is commonly used as a benchmark in software engineering research. The statistical results show that our methods have significant advantages compared with other methods in these programs, indicating the effectiveness of our approach for measuring class importance in software systems of varying sizes and complexity.

However, we acknowledge that the characteristics of the investigated systems may affect the generalizability of our findings. While our results are credible within the context of the analyzed systems, they may not apply to other programming languages or systems with different characteristics. Therefore, we emphasize the importance of considering the specific characteristics of the investigated system when interpreting the results.

Internal threats. To mitigate potential internal threats to our study's validity, we made several design choices in our analysis and research. For instance, we referred to the indicators summarized by Zhou and Leung (2006) that have a significantly stronger impact on class importance and combined them. Additionally, we used the entropy weight method to optimize the weights of the indicators, including attribute and method complexity to calculate the stubbing complexity, *CBO* and the number of features per class *NOF* to calculate the class importance. However, these choices may also affect the generalizability of our findings.

As for the comparative methods, we chose the most commonly used graph-based and search-based methods, which have been

Table 14
Comparison of OCplx with other methods.

Program	Graph-based method			Search-based method			RL-based method			
	Traon	Tai	Briand	GA	RIA	PSO	RL	CITO_RL	CITO_CRL	CITO_CRL'
elevator	–	–	–	2.03	2.04	2.87	1.63	**1.50**	**1.50**	**1.50**
SPM	8.40	8.08	5.82	3.50	3.48	3.02	6.06	3.62	3.48	**3.00**
ATM	3.37	2.99	2.70	3.09	2.43	2.59	3.69	2.31	2.32	**2.28**
daisy	–	–	–	0.58	0.32	0.92	0.16	**0.09**	**0.09**	0.10
ANT	3.72	3.87	3.31	2.13	2.23	2.32	2.22	**1.45**	1.48	**1.45**
email_spl	–	–	–	0.74	0.66	1.02	0.40	**0.32**	0.33	0.33
BCEL	8.23	8.68	5.81	9.70	8.71	8.61	5.91	5.85	6.19	**5.44**
DNS	5.02	4.63	**1.51**	5.51	4.33	5.81	2.11	1.90	2.14	2.12
notepad_spl	–	–	–	1.96	**1.92**	4.68	5.07	4.76	4.31	4.20

Table 15
Comparison of stubs with other methods.

Program	Graph-based method			Search-based method			RL-based method			
	Traon	Tai	Briand	GA	RIA	PSO	RL	CITO_RL	CITO_CRL	CITO_CRL'
elevator	–	–	–	8	8	6	**5**	**5**	**5**	**5**
SPM	25	20	17	13	14	**12**	16	21	21	20
ATM	9	8	**7**	11	9	8	8	15	14	14
daisy	–	–	–	7	5	8	**3**	**3**	**3**	**3**
ANT	18	28	**11**	14	13	14	12	13	13	15
email_spl	–	–	–	15	14	19	**6**	7	7	8
BCEL	68	128	70	63	**51**	55	69	73	74	72
DNS	11	27	**6**	33	27	17	11	12	11	12
notepad_spl	–	–	–	55	49	57	**47**	**47**	**47**	50

Table 16
P-values of OCplx.

Method1	Method2	elevator	SPM	ATM	daisy	ANT	email_spl	BCEL	DNS	notepad_spl
CITO_CRL	Traon	–	**1.06E−12**	**5.18E−13**	–	**1.58E−14**	–	**3.90E−14**	**5.74E−14**	–
	Tai	–	**1.06E−12**	**5.18E−13**	–	**1.58E−14**	–	**3.90E−14**	**5.74E−14**	–
	Briand	–	**1.06E−12**	**5.18E−13**	–	**1.58E−14**	–	3.90E−14	5.74E−14	–
	GA	**7.11E−10**	0.63	**1.30E−05**	**3.40E−09**	**2.12E−06**	**1.19E−11**	**2.24E−12**	**2.98E−12**	4.13E−11
	RIA	**1.14E−12**	**3.02E−04**	0.31	**4.87E−12**	**1.14E−12**	**1.19E−11**	**2.02E−12**	**2.97E−12**	2.77E−11
	PSO	**1.14E−12**	6.23E−09	**2.22E−04**	**4.87E−12**	**1.14E−12**	**1.19E−11**	**1.61E−12**	**4.35E−09**	**5.66E−03**
	RL	**2.54E−14**	**2.48E−11**	**9.57E−12**	**1.68E−13**	**7.66E−13**	**1.49E−12**	6.55E−08	0.18	**8.26E−05**
	CITO_RL	1.00	**3.84E−03**	0.66	0.23	**2.64E−03**	4.22E−03	3.52E−11	1.15E−11	**8.40E−03**
CITO_CRL'	Traon	–	**7.09E−13**	**1.58E−14**	–	**3.91E−13**	–	**7.11E−13**	**6.17E−13**	–
	Tai	–	**7.09E−13**	**1.58E−14**	–	**3.91E−13**	–	**7.11E−13**	**6.17E−13**	–
	Briand	–	**7.09E−13**	**1.58E−14**	–	**3.91E−13**	–	**7.11E−13**	6.17E−13	–
	GA	**7.11E−10**	**7.98E−03**	**2.12E−06**	**1.11E−08**	**1.36E−06**	**1.84E−11**	**1.99E−11**	**1.94E−11**	2.59E−11
	RIA	**1.14E−12**	**1.64E−05**	**1.77E−02**	**1.18E−11**	**1.26E−11**	**1.84E−11**	**1.83E−11**	**7.91E−11**	2.59E−11
	PSO	**1.14E−12**	0.14	**1.17E−07**	**1.18E−11**	**1.26E−11**	**1.84E−11**	**1.52E−11**	**2.57E−04**	**1.85E−05**
	RL	**5.18E−12**	**2.31E−09**	**3.89E−10**	**2.15E−11**	**1.37E−09**	**1.21E−10**	**8.08E−06**	0.42	**1.82E−07**
	CITO_RL	1.00	**2.01E−09**	**7.04E−04**	4.27E−03	0.08	6.62E−05	**1.85E−09**	3.24E−05	**6.13E−05**

widely used in multiple literatures (Jiang et al., 2021). We categorized these methods based on whether they consider class importance, and directly compared the methods that do consider class importance. For methods that do not consider class importance, we fully followed the original method's philosophy and carefully implemented the details of the method to compare them with our method. Although we did not optimize these methods, we do encourage future research to explore these approaches to measure class importance and investigate their impact on the generalizability of our findings.

6. Conclusions

The determination of CITO is an important issue in software testing. Important classes are error-prone in programs, and they can influence the stability of the whole testing. Most of the previous methods omit the consideration of class importance when generating CITOs. In this paper, we proposed two integration test order strategies considering the class importance based on RL, where two different reward functions combining class importance with stubbing complexity and test profit were applied, respectively. To verify the performance of our methods, we designed four research questions and performed experiments on nine programs to answer them. The experimental results showed that the different positions of important classes in the order could affect the CITOs and the stubbing cost. While allowing for early testing of important classes, our methods can decrease the stubbing cost. Compared with other methods, our method CITO_CRL' had more advantages in terms of the overall stubbing complexity.

Although we have verified the effectiveness of the proposed method on benchmarks and open-source programs, we have not evaluated it in experiments on real large-scale programs in different languages. Meanwhile, time effectiveness and fault detection effectiveness (Sharif et al., 2021) are also very important indicators to measure software testing methods, and the performance of our proposed methods on these indicators needs to be improved. Therefore, how to integrate our methods with the actual program under test, to address these issues requires further research.

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.

Table 17
Effect size *d* of OCplx.

Method1	Method2	elevator	SPM	ATM	daisy	ANT	email_spl	BCEL	DNS	notepad_spl
CITO_CRL	Traon	–	−1.84	−1.86	–	−1.98	–	−1.95	−1.94	–
	Tai	–	−1.84	−1.86	–	−1.98	–	−1.95	−1.94	–
	Briand	–	−1.84	−1.86	–	−1.98	–	−1.95	−1.94	–
	GA	−1.59	−0.13	−1.13	−1.53	−1.22	−1.75	−1.81	−1.80	−1.70
	RIA	−1.84	−0.93	−0.26	−1.78	−1.84	−1.75	−1.82	−1.80	−1.72
	PSO	−1.84	−1.50	−0.95	−1.78	−1.84	−1.75	−1.82	−1.52	−0.71
	RL	−1.97	−1.72	−1.76	−1.90	−1.85	−1.83	−1.40	−0.34	−1.02
	CITO_RL	0.00	−0.75	−0.11	−0.31	−0.78	−0.74	−1.71	−1.75	−0.68
CITO_CRL'	TRAON	–	−1.85	−1.98	–	−1.87	–	−1.85	−1.86	–
	TAI	–	−1.85	−1.98	–	−1.87	–	−1.85	−1.86	–
	BRIAND	–	−1.85	−1.98	–	−1.87	–	−1.85	−1.86	–
	GA	−1.59	−0.68	−1.22	−1.47	−1.25	−1.73	−1.73	−1.73	−1.72
	RIA	−1.84	−1.11	−0.61	−1.75	−1.75	−1.73	−1.73	−1.68	−1.72
	PSO	−1.84	−0.38	−1.37	−1.75	−1.75	−1.73	−1.74	−0.94	−1.11
	RL	−1.78	−1.54	−1.62	−1.73	−1.56	−1.66	−1.15	−0.21	−1.35
	CITO_RL	0.00	−1.55	−0.87	−0.74	−0.46	−1.03	−1.55	−1.07	−1.03

Acknowledgment

This work is supported by the Fundamental Research Funds for the Central Universities, China under grant No. 2022XSCX18.

References

Beizer, B., 2003. Software Testing Techniques. Dreamtech Press.
Binder, R.V., 1999. Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Longman Publishing.
Borner, L., Paech, B., 2009. Integration test order strategies to consider test focus and simulation effort. In: Proc. of the 1st Int'L Conf. on Advances in System Testing and Validation Lifecycle. IEEE, pp. 80–85. http://dx.doi.org/10.1109/VALID.2009.30.
Briand, L.C., Feng, J., Labiche, Y., 2002. Using genetic algorithms and coupling measures to devise optimal integration test orders. In: Proc. of the 14th Int'L Conf. on Software Engineering and Knowledge Engineering. pp. 43–50. http://dx.doi.org/10.1145/568760.568769.
Briand, L.C., Labiche, Y., Wang, Y., 2001. Revisiting strategies for ordering class integration testing in the presence of dependency cycles. In: Proc. of the 12th Int'L Symposium on Software Reliability Engineering. IEEE, pp. 287–296. http://dx.doi.org/10.1109/ISSRE.2001.989482.
Briand, L.C., Labiche, Y., Wang, Y., 2003. An investigation of graph-based class integration test order strategies. IEEE Trans. Softw. Eng. 29 (7), 594–607. http://dx.doi.org/10.1109/TSE.2003.1214324.
Cheng, Q., 2010. Structure entropy weight method to confirm the weight of evaluating index. Syst. Eng. Theory Pract. 30 (7), 1225–1228.
Chidamber, S.R., Kemerer, C.F., 1991. Towards a metrics suite for object oriented design. In: Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications. pp. 197–211.
Cui, Y., Zhang, Q., Feng, Z., Wei, Z., Shi, C., Yang, H., 2022. Topology-aware resilient routing protocol for FANETs: An adaptive Q-learning approach. IEEE Internet Things J. 1–18. http://dx.doi.org/10.1109/JIOT.2022.3162849.
Czibula, G., Czibula, I.G., Marian, Z., 2018. An effective approach for determining the class integration test order using reinforcement learning. Appl. Soft Comput. 65, 517–530. http://dx.doi.org/10.1016/j.asoc.2018.01.042.
Ding, Y., Zhang, Y., Jiang, S., Yuan, G., Qian, J., 2022. Research on generation method of class integration test order based on reinforcement learning. J. Softw. 33 (5), 1–22. http://dx.doi.org/10.13328/j.cnki.jos.006555.
Fritz, C.O., Morris, P.E., Richler, J.J., 2012. Effect size estimates: current use, calculations, and interpretation. J. Exp. Psychol. Gen. 141 (1), 2.
Gass, S.I., Fu, M.C. (Eds.), 2013. Bellman optimality equation. In: Encyclopedia of Operations Research and Management Science. Springer, p. 115. http://dx.doi.org/10.1007/978-1-4419-1153-7_200007.
Guizzo, G., Fritsche, G.M., Vergilio, S.R., Pozo, A.T.R., 2015. A hyper-heuristic for the multi-objective integration and test order problem. In: Proc. of the 17th Annual Conf. on Genetic and Evolutionary Computation. pp. 1343–1350. http://dx.doi.org/10.1145/2739480.2754725.
He, D., Xu, J., Chen, X., 2016. Information-theoretic-entropy based weight aggregation method in multiple-attribute group decision-making. Entropy 18 (6), http://dx.doi.org/10.3390/e18060171.
Hu, X., 2008. Action choice mechanism of reinforcement learning based on adjusted dynamic parameters. Comput. Eng. Appl. 44 (28), 29–31. http://dx.doi.org/10.3778/j.issn.1002-8331.2008.28.009.
Huang, C., Zhang, R., Ouyang, M., Wei, P., Lin, J., Su, J., Lin, L., 2021. Deductive reinforcement learning for visual autonomous urban driving navigation. IEEE Trans. Neural Netw. Learn. Syst. 32 (12), 5379–5391. http://dx.doi.org/10.1109/TNNLS.2021.3109284.
Huo, X., Thung, F., Li, M., Lo, D., Shi, S., 2021. Deep transfer bug localization. IEEE Trans. Softw. Eng. 47 (7), 1368–1380. http://dx.doi.org/10.1109/TSE.2019.2920771.
Jiang, S., Zhang, M., Zhang, Y., Wang, R., Yu, Q., Keung, J.W., 2021. An integration test order strategy to consider control coupling. IEEE Trans. Softw. Eng. 47 (7), 1350–1367. http://dx.doi.org/10.1109/TSE.2019.2921965.
Kleinberg, J., 1999. Authoritative sources in a hyperlinked environment. J. ACM 46 (5), 604–632.
Kung, D.C., Gao, J., Hsia, P., Lin, J., Toyoshima, Y., 1995a. Class firewall, test order, and regression testing of object-oriented programs. J. Object-Oriented Program. 8 (2), 51–65.
Kung, D., Gao, J., Hsia, P., Toyoshima, Y., Chen, C., 1995b. A test strategy for object-oriented programs. In: Proc. of the 19th Int'L Computer Software and Applications Conf. pp. 239–244. http://dx.doi.org/10.1109/CMPSAC.1995.524786.
Le Hanh, V., Akif, K., Le Traon, Y., Jézéque, J.-M., 2001. Selecting an efficient OO integration testing strategy: an experimental comparison of actual strategies. In: Proc. of the 15th European Conf. on Object-Oriented Programming. Springer, pp. 381–401. http://dx.doi.org/10.1007/3-540-45337-7_20.
Le Traon, Y., Jéron, T., Jézéquel, J.-M., Morel, P., 2000. Efficient object-oriented integration and regression testing. IEEE Trans. Reliab. 49 (1), 12–25. http://dx.doi.org/10.1109/24.855533.
Li, H., Gao, G., Ge, X., Guo, S., Hao, L.-Y., 2018. The influence ranking for software classes. In: Proc. of the 14th Int'L Conf. on Natural Computation, Fuzzy Systems and Knowledge Discovery. pp. 1127–1133. http://dx.doi.org/10.1109/FSKD.2018.8687115.
Mann, H.B., Whitney, D.R., 1947. On a test of whether one of two random variables is stochasitcally larger than the other. Ann. Math. Stat. 18 (1), 60–70. http://dx.doi.org/10.1214/AOMS/1177730491.
Mariani, T., Guizzo, G., Vergilio, S.R., Pozo, A.T., 2016. Grammatical evolution for the multi-objective integration and test order problem. In: Proc. of the 18th Annual Conf. on Genetic and Evolutionary Computation. pp. 1069–1076. http://dx.doi.org/10.1145/2908812.2908816.
Meyer, P., Siy, H., Bhowmick, S., 2015. Identifying important classes of large software systems through K-core decomposition. Adv. Complex Syst. 17, 1–49. http://dx.doi.org/10.1142/S0219525915500046.
Pan, W., Song, B., Li, K., Zhang, K., 2018. Identifying key classes in object-oriented software using generalized k-core decomposition. Future Gener. Comput. Syst. 81, 188–202. http://dx.doi.org/10.1016/j.future.2017.10.006.
Rana, R., Staron, M., Berger, C., Hansson, J., Nilsson, M., 2014. Analysing defect inflow distribution of automotive software projects. In: Proc. of the 10th Int'L Conf. on Predictive Models in Software Engineering. pp. 22–31. http://dx.doi.org/10.1145/2639490.2639507.
Rana, R., Staron, M., Berger, C., Hansson, J., Nilsson, M., Meding, W., 2016. Analyzing defect inflow distribution and applying Bayesian inference method for software defect prediction in large software projects. J. Syst. Softw. 117, 229–244. http://dx.doi.org/10.1016/j.jss.2016.02.015.
Sharif, A., Marijan, D., Liaaen, M., 2021. DeepOrder: Deep learning for test case prioritization in continuous integration testing. In: IEEE International Conference on Software Maintenance and Evolution. pp. 525–534. http://dx.doi.org/10.1109/ICSME52107.2021.00053.
Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D., 2016. Mastering the game of Go with deep neural networks and tree search. Nature 529, 484–489. http://dx.doi.org/10.1038/nature16961.

Subramanyam, R., Krishnan, M., 2003. Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects. IEEE Trans. Softw. Eng. 29 (4), 297–310. http://dx.doi.org/10.1109/TSE.2003.1191795.

Sutton, R.S., Barto, A.G., 2018. Reinforcement Learning: An Introduction, second ed. MIT Press.

Tahvili, S., Afzal, W., Saadatmand, M., Bohlin, M., Sundmark, D., Larsson, S., 2016. Towards earlier fault detection by value-driven prioritization of test cases using fuzzy TOPSIS. In: Information Technology: New Generations. Springer, pp. 745–759. http://dx.doi.org/10.1007/978-3-319-32467-8_65.

Tahvili, S., Ahlberg, M., Fornander, E., Afzal, W., Saadatmand, M., Bohlin, M., Sarabi, M., 2018. Functional dependency detection for integration test cases. In: Proc. of the 18th IEEE Int'L Conf. on Software Quality, Reliability and Security Companion. pp. 207–214. http://dx.doi.org/10.1109/QRS-C.2018.00047.

Tai, K.C., Daniels, F.J., 1997. Test order for inter-class integration testing of object-oriented software. In: Proc. of the 21st Annual Int'L Computer Software and Applications Conf. IEEE, pp. 602–607. http://dx.doi.org/10.1109/CMPSAC.1997.625079.

Tarjan, R., 1972. Depth-first search and linear graph algorithms. SIAM J. Comput. 1 (2), 146–160.

Wang, Z., Li, B., Wang, L., Li, Q., 2011. An effective approach for automatic generation of class integration test order. In: Proc. of the 35th Annual Computer Software and Applications Conf. IEEE, pp. 680–681. http://dx.doi.org/10.1109/COMPSAC.2011.122.

Wang, Y., Yu, H., Zhu, Z., 2016a. A class integration test order method based on the node importance of software. J. Comput. Res. Dev. 53 (3), 517. http://dx.doi.org/10.7544/issn1000-1239.2016.20148318.

Wang, T., Zhang, Z., Jing, X., Zhang, L., 2016b. Multiple kernel ensemble learning for software defect prediction. Autom. Softw. Eng. 23 (4), 569–590. http://dx.doi.org/10.1007/s10515-015-0179-1.

Watkins, C.J.C.H., Dayan, P., 1992. Q-learning. Mach. Learn. 8 (3–4), 279–292. http://dx.doi.org/10.1007/BF00992698.

Xie, Z., Li, M., 2018. Cutting the software building efforts in continuous integration by semi-supervised online AUC optimization. In: Proc. of the 27th Int'L Joint Conference on Artificial Intelligence. IJCAI-18, International Joint Conf. on Artificial Intelligence Organization, pp. 2875–2881. http://dx.doi.org/10.24963/ijcai.2018/399.

Yang, Y., Pan, C., Li, Z., Zhao, R., 2021. Adaptive reward computation in reinforcement learning-based continuous integration testing. IEEE Access 9, 36674–36688. http://dx.doi.org/10.1109/ACCESS.2021.3063232.

Zaidman, A., Demeyer, S., 2008. Automatic identification of key classes in a software system using webmining techniques. J. Softw. Maint. Evol.-R. 20 (6), 387–417. http://dx.doi.org/10.1002/smr.370.

Zhang, Y., Jiang, S., Chen, R., Wang, X., Zhangmiao, 2018a. Class integration testing order determination method based on particle swarm optimization algorithm. Chinese J. Comput. 41 (4), 931–945.

Zhang, Y., Jiang, S., Zhang, M., Ju, X., 2018b. Survay of class test order generation techniques for integration test. Chinese J. Comput. 41 (3), 670–694. http://dx.doi.org/10.11897/SP.J.1016.2018.00670.

Zhang, M., Jiang, S., Zhang, Y., Wang, X., Yu, Q., 2017. A multi-level feedback approach for the class integration and test order problem. J. Syst. Softw. 133, 54–67. http://dx.doi.org/10.1016/j.jss.2017.08.026.

Zhang, M., Keung, J.W., Xiao, Y., Kabir, M.A., 2021. Evaluating the effects of similar-class combination on class integration test order generation. Inf. Softw. Technol. 129, 1–16. http://dx.doi.org/10.1016/j.infsof.2020.106438.

Zhang, M., Keung, J., Xiao, Y., Kabir, M.A., Feng, S., 2019. A heuristic approach to break cycles for the class integration test order generation. In: Proc. of the 43rd Annual Computer Software and Applications Conf. 1, IEEE, pp. 47–52. http://dx.doi.org/10.1109/COMPSAC.2019.00016.

Zhao, Y., Wang, Y., Yu, H., Zhu, Z., 2015. An inter-class integration test order generation method based on complex networks. J. Northeast. Univ. Nat. Sci. 36 (12), 1696–1700. http://dx.doi.org/10.3969/j.issn.1005-3026.2015.12.006.

Zhou, Y., Leung, H., 2006. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. IEEE Trans. Softw. Eng. 32 (10), 771–789. http://dx.doi.org/10.1109/TSE.2006.102.

Yanru Ding is a Ph.D. student in the Computer Science Department at China University of Mining and Technology, Xuzhou, Jiangsu. Her research interests include software engineering, program analysis, and testing.

Yanmei Zhang received the Ph.D. degree in Computer Science from China University of Mining and Technology, Xuzhou, Jiangsu, in 2012. From 2012 to 2017, she has been a Lecturer; since 2018, she has been an Associate Professor at the Computer Science Department, China University of Mining and Technology, Xuzhou, Jiangsu. She is the author of one book, more than 10 articles, and more than 2 inventions. Her research interests include software engineering, program analysis and testing, software quality, reinforcement learning, intelligent evolutionary algorithm, etc.

Guan Yuan received the B.E., M.E., and Ph.D. degrees in Computer Science and Technology from China University of Mining and Technology in 2004, 2009, and 2012, respectively. From 2012 to today, he has been a Teaching Assistant, Lecturer, Associate Professor, and Professor in the Computer Science Department, China University of Mining and Technology, Xuzhou, Jiangsu. His research interests include software engineering and data mining, especially moving object data mining.

Shujuan Jiang received her Ph.D. degree in Computer Science from Southeast University, Nanjing, Jiangsu, in 2006. From 1995 to today, she has been a Teaching Assistant, Lecturer, Associate Professor, and Professor in the Computer Science Department, China University of Mining and Technology, Xuzhou, Jiangsu. She is the author of more than 80 articles. Her research interests include software engineering, program analysis and testing, software maintenance, etc. She is a Member of the IEEE.

Wei Dai received the M.S. and Ph.D. degrees in control theory and control engineering from Northeastern University, Shenyang, China, in 2009 and 2015, respectively. From 2013 to 2015, he was as a Teaching Assistant with the State Key Laboratory of Synthetical Automation for Process Industries, Northeastern University. He is currently an Associate Professor with the China University of Mining and Technology, Xuzhou, China. His current research interests include modeling, optimization, and control of the complex industrial process, data mining, and machine learning.

Yinghui Zhang is currently an M.S. student in the School of Computer Science and Technology, China University of Mining and Technology, Xuzhou, Jiangsu. His research interests include reinforcement learning and software integration testing.