Автоматизовані системи управління та прилади автоматики. 2025. № 4 (187)

UDC 004.415.2:004.415.53 **DOI:** https://doi.org/10.30837/0135-1710.2025.187.020

M. Hulevych

EVALUATION OF THE EFFECTIVENESS OF THE TEST CASE FORMING METHOD FOR C++ LIBRARIES BASED ON A Q-LEARNING AGENT

Effective optimization of test cases (TC) is a prerequisite for improving the effectiveness of regression testing of C++ libraries. **The subject of the study** is methods for forming (optimizing) TC for C++ libraries. **The purpose of the work** is to evaluate the effectiveness of the method of forming TC for C++ libraries based on a Q-learning agent. **Research tasks:** to improve the mathematical model of a Q-learning agent to increase the efficiency of forming TC for C++ libraries in conditions of high sparsity of the state space of the Q-learning agent; to investigate the influence of the parameters of the improved Q-learning agent model on its behavior under such conditions; to consider the possibility of minimizing the formed TC by the method of their formation using the delta-debugging algorithm for TC minimization; to evaluate the effectiveness of the proposed method and compare it with known TC optimization methods**. Research methods.** The work uses the Monte Carlo Tree Search method, the classical mathematical model of Q-learning, the delta debugging algorithm for TC minimization, and the greedy algorithm for TC optimization. The effectiveness of the proposed method was evaluated on two open-source C++ libraries using statistical analysis of 100 mathematical simulations of the configurations of the methods under study. **Results achieved:** the effectiveness evaluation indicates that the proposed method provides the following average values of TC optimization effectiveness coefficients for C++ libraries: the coverage retention ratio is up to 1.225, the test suite compression ratio is up to 0.86, and the test suite execution time reduction ratio is up to 0.74. It has been established that, when compared with the greedy optimization algorithm, the delta debugging minimization algorithm, and the optimization method based on Monte Carlo Tree Search, the proposed method significantly improves the efficiency of TC formation (optimization) for C++ libraries. **Conclusions.** The improved mathematical model provides generalization of the experience of the Q-learning agent between similar test cases and increases the efficiency of their formation in conditions of high sparsity of the state space of the Q-learning agent. Thus, the results of evaluating the method of forming TC for C++ libraries based on a Q-learning agent confirm its feasibility in solving the problem of forming (optimization) of TC for C++ libraries, which makes it possible to reduce the length of TC in the test suite without loss of the branch coverage of the C++ library code being tested and to reduce the execution time of TS. Further research will be devoted to the formation of TC for C++ libraries based on a deep reinforcement learning agent.

Keywords: test case optimization; software testing; Q-learning; Q-table; delta debugging; code coverage; minimization; C++; agent; reinforcement learning.

Introduction

Software testing is a key element of software quality assurance. The testing process uses test suite (TS) consisting of test cases (TC) for interaction with the corresponding software interfaces of C++ libraries and their components.

TC for C++ libraries and C++ libraries are developed in parallel, reflecting the practical features of using the API (Application Programming Interface, API) of the tested code. They accumulate throughout the software development life cycle and gradually cover a significant part of the functionality, as well as being the main source for verifying the correct

behavior of the output software product when changes are made to the code base. As the volume of such test cases increases, the cost of regression testing increases in both time and computational terms.

C++ software libraries are used in large projects for image processing, file system operations, data analysis, etc. In such conditions, changes to the C++ library code lead to the need to re-execute a significant amount of TC in redundant TS to detect software regression defects.

The execution time of C++ library TS can be reduced by optimizing the TC in the original TS. Thus, the task of TC optimization (Test Case Optimization, TCO) without loss of coverage is a relevant task that allows reducing testing costs, improving the informativeness of TC execution results, and, as a result, ensuring the effectiveness of regression testing.

Previous studies have proposed a method for forming TC for C++ libraries based on a Q-learning agent, which reduces the size of original TS without loss of branch coverage of the code [1]. To justify the effectiveness of the proposed method, it is advisable to conduct a comparative analysis with relevant TC optimization methods.

Analysis of literature

Over the past decade, a number of review papers have been published that provide a fundamental understanding of existing TC optimization methods and criteria for evaluating their effectiveness. Systematic reviews [2–4] provide a basis for understanding the task of TC optimization, covering a broad classification of TC optimization methods, as well as the features of software environments for researching TC optimization methods.

The importance of preserving the semantics of the original test suite, forming adequate criteria for evaluating effectiveness, and adhering to the methodological foundations of researching TC optimization methods is emphasized separately.

The task of TS optimization consists of three interrelated tasks: TCS (Test Case Selection, TCS), TSP (Test Suite Prioritization, TSP), and TSM (Test Suite Minimization, TSM) [5]. The purpose of TCS is to determine a subset of TC that need to be re-executed after changes in the program code of the library being tested. The purpose of TSP is the ordering of test cases in order to maximize the speed of software defect detection. TSM removes redundant TC from the TS while maintaining testing efficiency, thereby reducing the time required to execute the TS.

Machine learning is used to optimize TS [6]. In particular, clustering algorithms are effectively used to reduce redundancy in TS and prioritize the TC [7].

They are capable of operating based on the performance characteristics of C++ library tasks, which makes them useful for optimizing TC for C++ libraries without formal specifications (documented requirements for C++ library behavior). Reinforcement learning can also be effectively applied in these conditions [8].

An important factor in evaluating the effectiveness of TC optimization methods is the classification into adequate and inadequate methods, as formulated in [9]. According to this classification, adequate (coverage-preserving) TC optimization methods are those that ensure the preservation of the full level of testing effectiveness relative to the original TS.

After optimization, an adequate method guarantees that all structural (behavioral) constructs of the software that were tested by the original test cases remain covered by the new test cases.

Thus, the adequacy characteristic is critically important in terms of the suitability of the TC optimization method for regression testing, where the loss of the test case's ability to detect defects in the software under test is unacceptable.

In contrast, non-coverage-preserving methods allow for a partial loss of testing effectiveness in order to achieve higher test case compression rates. Inadequate methods can significantly reduce the amount of test coverage by reducing the ability to detect software defects, makingthem unsuitable for TC optimization for highly reliable software systems or libraries.

The need for adequate TC optimization methods is also confirmed by an analysis of the peculiarities of testing C++ libraries [10], which focuses on the following aspects:

– the absence of formal API specifications in open-source C++ libraries, which increases the role of methods capable of working in such conditions;

– the need to analyze the dependencies between test case instructions during execution in order to optimize the TC effectively;

– the complexity of the execution environment of C++ software, which necessitates flexible methods for forming TC for C++ libraries.

The author's previous work [1] provides a thorough overview of TC optimization methods – delta debugging, dynamic slicing, integer linear programming (ILP) TC optimization methods, methods based on classification and clustering, as well as hybrid methods.

To expand the analytical context of the previous work and clarify the role of individual methods in modern TC optimization research, Table 1 provides a comparative description of recent studies of TC optimization methods in different classes of problems.

Table 1. *Comparative characteristics of studies of TC optimization methods in different classes of problems*

Research	Method class	Research features	Research results
ATM method [11]	Heuristic method of TSP based on evolutionary search	Representation of TC in the form of abstract syntactic trees, search based on TC similarity.	Detection of up to 0.82 software defects when executing 50 % of TS.
GA-based Test Suite Minimization (TSM) [12]	Minimization of TS based on a genetic algorithm	An analysis of the configuration of vehicle parameters based on a genetic algorithm was conducted.	The possibility of finding a balance between maintaining coverage and reducing the execution time of the TS has been demonstrated.
Metaheuristic Fault Detection [13]	Metaheuristics for detecting software defects.	An analysis was conducted on particle swarm optimization, ant colony optimization, cuckoo search, firefly algorithm, etc.	Ability to work effectively without prior knowledge of software behavior. Low reproducibility of results.

Continuation of the Table 1

Research	**Method class**	**Research features**	**Research results**
EA-based prioritization [14]	Evolutionary method of prioritizing TC.	An analysis of the effectiveness of the method was performed with an increase in the volume of input data in large projects.	Effective allocation of time for testing software modules, which increases the efficiency of software testing.
RL+GA Hybrid [15]	A hybrid method based on reinforcement learning and a genetic algorithm.	The possibility of using genetic algorithms to configure the input parameters of a Q-learning agent was investigated.	Acceleration of policy convergence when training an agent on pre-configured input data based on a genetic algorithm.
General Monte Carlo Tree Search (MCTS) study [16]	Monte Carlo Tree Search	An analysis of the applicability of the Monte Carlo method for various classes of problems, including modifications of the method and hybrid configurations, was conducted.	The method is highly effective, but special improvements are needed for different classes of problems.
Monte Carlo Tree Search input parameters fuzzing [17]	Prioritization of input data for training deep neural networks based on Monte Carlo Tree Search method	Testing model as a decision-making process. Investigation of large input data search spaces for TC.	The results prove the effectiveness of the method and show an increase of up to + 30 % in code coverage compared to basic methods.
Monte Carlo Tree Search + UCT [18]	Monte Carlo Tree Search method with upper confidence bound	A classical algorithm based on the Upper Confidence Bound (UCB) is proposed. A classical combination of the MCTS method with the UCB algorithm is proposed.	A fundamental algorithm that laid the foundation for modern methods based on Monte Carlo Tree Search.
Greedy TSM [19]	Greedy method of TS minimization	Research on minimization of TS based on a greedy algorithm depending on the size of input TS and test requirements.	The results obtained prove the possibility of compressing TS to 50–75 % of the original size while maintaining coverage.
Greedy TSM [20]	Greedy method of TSM	Proposed two-criteria optimization of TS based on a greedy algorithm.	The effectiveness of the algorithm has been proven. The results obtained indicate a significant compression of TS while maintaining test requirements.
Greedy TSM [21]	Greedy method of TSM	Analysis of the effectiveness of the method based on mutation testing.	The possibility of compressing TS by an average of 70 % with high software defect detection capability has been proven.
Greedy TSP [22]	Greedy method of prioritizing TS	Analysis of the application of the greedy method for prioritizing TS in regression testing.	The results show a significant reduction in TS while maintaining the ability to detect software defects.

Continuation of the Table 1

Research	Method class	Research features	Research results
Delta Debugging TSM [23]	Delta debugging method for minimizing TS.	A classic delta debugging algorithm, ddmin, is proposed.	Significant reduction of input data without losing the ability to reproduce defects.
Delta Debugging for Fault Localization [24]	Delta debugging method for software defect localization.	Application of DD for localization of test cases leading to reproduction of specified software defects.	Significant reduction in input data without losing the ability to reproduce defects.
Hierarchical Delta Debugging [25]	Hierarchical modification of the delta debugging method for minimizing TS.	Hierarchical version of DD for structured scenarios.	Reduction of the time required to perform TS minimization while maintaining the effectiveness of reduced test cases.
Probabilistic Delta Debugging [26]	Stochastic modification of the delta debugging method for TS minimization.	A proposed probabilistic model based on the delta debugging algorithm and the representation of code as an abstract syntax tree.	The method considers the syntactic relationships between elements and the results of previous tests, which makes it possible to reduce the average processing time by almost 27 % and reduce the size of the TS by 3.4 times compared to existing methods.

To evaluate the effectiveness of the proposed method of TC forming based on a Q-learning agent, it is advisable to use those basic methods that:

– do not require large training data sets or pre-trained models describing the expected behavior of the software;

– can be applied to optimize TC for testing C++ libraries.

Greedy TC optimization algorithms remain the most effective and stable for minimizing TS. Studies [19–22] show that TS minimization methods based on greedy algorithms provide a significant reduction in TS size (up to 50–75 %) while maintaining the ability to detect software defects and preserving testing efficiency.

TC optimization methods based on the delta debugging algorithm (classical admin, hierarchical and probabilistic modifications) [23–26] have stable results in the task of minimizing TC with an increase in the volume of input data. They ensure effective localization of minimal test cases subset while maintaining the ability to reproduce software defects. However, these methods only optimize the structure of existing test suite without forming new test cases at the API call level.

Search methods on the Monte Carlo tree, including those based on the UCT algorithm [17–19], combine research and the use of acquired knowledge, making them relevant for the formation of TC in large action spaces.

At the same time, the application of the Monte Carlo method of TC optimization for C++ libraries requires adaptation: mechanisms are needed to verify the validity of the formed TC, determine the rational depth of the search tree, and limit the input data space.

Other groups – evolutionary algorithms, metaheuristics, and hybrid methods [13–16] – demonstrate high efficiency on large input data space but are not effective in the task of optimizing TC for C++ libraries in the absence of a predefined software behavior model (without formal specifications).

Therefore, for a comparative assessment of the effectiveness of the proposed method, the following were selected:

– Greedy TC optimization algorithm;
– Delta debugging algorithm for TC minimization;
– Monte Carlo Tree Search method.

Thus, the choice of these methods as basic ones is justified from both a theoretical and practical point of view.

Purpose and objectives

The purpose of the study is to evaluate the effectiveness of the method of forming TC for C++ libraries based on a Q-learning agent.

To achieve this goal, the following tasks are set in the study:

1. To improve the mathematical model of the Q-learning agent to increase the efficiency of TC formation for C++ libraries in conditions of high sparsity of the state space of the Q-learning agent.

2. To investigate the influence of the parameters of the improved Q-learning agent model on the agent's behavior in such conditions.

3. To consider the possibility of minimizing the obtained TC using the proposed TC formation method based on the delta-debugging algorithm for TS minimization.

4. To evaluate the effectiveness of the proposed method in a unified TC formation environment, based on the original test suites of two open-source C++ libraries. Use the classic greedy TC optimization algorithm, the delta debugging TC minimization algorithm, and the TC optimization method based on Monte Carlo Tree Search as the main methods for comparison.

Main part

1. Classic Q-learning agent model

The task of forming TC for C++ libraries without API specifications can be formalized as a Markov Decision Process (MDP) [27], where an agent builds a test case step by step, choosing the next action from the permissible API space:

$$\langle S, A, P(s'|s,a), R(s,a), \gamma \rangle,$$

where S is the set of states corresponding to the state s; A is the set of permissible actions in the state s; $P(s'|s,a)$ is the probability function of transition to a new state after performing

the action a in the state s; $\mathrm{R}(\mathrm{s},\mathrm{a})$ is the reward function provided for performing an action a in the state s; $\gamma \in (0,1)$ is the discount factor.

A test case of length $t \in \mathrm{N}$, which is formed as follows:

$$\mathrm{TC}^{\mathrm{t}} = \mathrm{a}_1, \mathrm{a}_2, \ldots, \mathrm{a}_{\mathrm{t}}, \ \mathrm{a}_{\mathrm{i}} \in \mathrm{A}, \tag{1}$$

consists of a sequence of a C++ library API calls.

The classic Q-learning algorithm belongs to the class of reinforcement learning methods, which allow the agent to sequentially improve its action selection policy based on experience gained during an interaction with the environment. The agent's goal is to maximize the expected total reward by approximating the optimal action utility function:

$$\mathrm{Q}^{*}(\mathrm{s},\mathrm{a}) = \max_{\pi} \mathrm{E}\left[\sum_{\mathrm{t}=0}^{\infty} \gamma^{\mathrm{t}} \times \mathrm{R}(\mathrm{s}_{\mathrm{t}}, \mathrm{a}_{\mathrm{t}})\right], \tag{2}$$

where π is the action selection policy.

The formula considers the randomness of transitions between agent states by maximizing the mathematical expectation of the discounted sum of rewards. During training, Q-values are updated according to Bellman's rule:

$$\mathrm{Q}(\mathrm{s}_{\mathrm{t}}, \mathrm{a}_{\mathrm{t}}) \leftarrow \mathrm{Q}(\mathrm{s}_{\mathrm{t}}, \mathrm{a}_{\mathrm{t}}) + \alpha \times \left[\mathrm{R}(\mathrm{s}_{\mathrm{t}}, \mathrm{a}_{\mathrm{t}}) + \gamma \times \max_{\mathrm{a}'} \mathrm{Q}(\mathrm{s}_{\mathrm{t+1}}, \mathrm{a}') - \mathrm{Q}(\mathrm{s}_{\mathrm{t}}, \mathrm{a}_{\mathrm{t}})\right], \tag{3}$$

where $\alpha \in (0,1]$ is the learning rate multiplied by $\delta^{t} = [\ldots]$ – the TDE (Temporal Difference Error, TDE). The agent sequentially updates Q-values in the Q-table, learning to distinguish useful actions (which increase the coverage or efficiency of the test case) from uninformative or redundant ones.

The process of training an agent with reinforcement can be presented in the form of a diagram shown in Fig. 1.

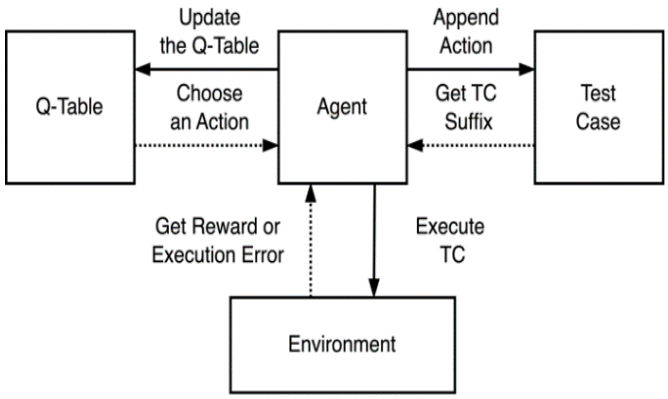


Fig. 1. Diagram of the reinforcement learning process for a Q-learning agent

The agent interacts with the environment sequentially, selecting actions based on the Q-table, updating it based on the results of the test case execution and the rewards received. Each interaction cycle includes selecting an action, forming a partial test case, executing the action in the environment, receiving a reward or execution error, and updating the corresponding Q-values in the Q-table.

The classic Q-learning agent model allows the agent to be gradually trained in effective TC construction strategies. However, the application of the classic Q-learning agent model to solve the TC forming (optimization) problem for C++ libraries has a number of limitations:

– *explosion of the state space dimension*. The space of all possible API call sequences grows exponentially depending on the complexity of the C++ library being tested. This approach leads to high sparsity of the agent's state space in the Q-table – a large number of states occur rarely or not at all, which complicates the generalization of the agent experience;

– *lack of generalization between similar states.* Classical Q-learning stores Q-values for each specific state and does not take into account the structural similarity between agent states that have common call suffixes. This approach can reduce the stability and speed of learning.

2. Improving the classic Q-learning agent model

The shortcomings of the classical Q-learning agent model, such as the explosion of the state space dimension and the lack of generalization between similar agent states, are eliminated by introducing an improved agent model based on a mixed Q-value estimation. To reduce computational complexity, the state s_t is restricted by introducing a test case suffix function.

Thus, let there be a current sequence of API calls of the tested library at step t that corresponds to a partially formed test case (1), and let $k \in \mathbb{N}$ be the length of the call history.

Then, in the previous model [1], the agent's state can be determined based on the last k actions using the test case suffix function as follows:

$$s_t^k \leftarrow \begin{cases} \varnothing, & if\ t = 0 \\ a_1, \ldots, a_t & if\ 0 < t \le k \\ a_{t-k+1}, \ldots, a_t & if\ t > k \end{cases} \quad (4)$$

Unlike the classic single-layer Q-table, in which each state is stored separately, the proposed model uses a multi-layer structure with hierarchical merging of test case suffixes. Each partially formed test case TC^t has a set of suffixes calculated according to (4), which can be represented as follows:

$$S_{suf} = \left\{ s_t^i \mid i = 1,..,k \right\}. \quad (5)$$

This approach allows us to interpret the Q-table as a multilayer table:

$$Q(s,a) = \left\{ Q(s,a)^{(0)}, Q(s,a)^{(1)}, \ldots, Q(s,a)^{(k)} \right\}, \quad (6)$$

where $Q(s,a)^{(0)}$ is the evaluation for the initial state of the agent, $Q(s,a)^{(k)}$ is the evaluation for the suffix of the test case of depth k .

The concept of the spatial influence of TC suffixes is based on the theory of sequence generalization in reinforcement learning [28].

When updating Q-values according to (3), the agent distributes the TDE not only to the current state, but also to all its suffixes. Updates for each suffix depth level $i = 1,..,k$ are performed with an exponential decrease in weight:

$$Q\left(s_t^i, a_t\right) \leftarrow Q\left(s_t^i, a_t\right) + \alpha \times \lambda^{k+1-i} \times \delta^t, \tag{7}$$

where $\lambda \in (0,1)$ is the decay coefficient of shorter TC suffixes influence.

The proposed rule for updating a multilayer Q-table is based on the classical principles of reinforcement learning described in [29], which assume exponential decay of the influence of past states on the parameter. The use of λ discounting in the structural dimension of shorter TC suffixes, as opposed to time discounting of rewards (eligibility traces), is based on the concept of state aggregation [30], which allows generalizing Q-values for similar states.

When selecting an action $a \in \mathsf{A}$, the Q-value estimate for available actions is calculated as the weighted average of Q-values calculated for all suffixes of the current state:

$$\tilde{Q}(s,a) = \sum_{i=1}^{k} w_i \times Q\left(s_t, a_t\right)^{(i)} \Big/ \sum_{i=1}^{k} w_i, \tag{8}$$

w_i is the weight coefficient of the suffix i, which is determined by the following formula:

$$w_i = \left(N_i + \xi\right)^{\beta} \times \lambda^{k+1-i}, \tag{9}$$

where N_i is the number of updates of the Q-table for the suffix i, ξ is a small stabilizing term, and $\beta \in (0,1]$ is the experience amplification coefficient.

Thus, formula (8) describes the mechanism of generalizing Q-values in the proposed multilayer Q-table. The weights w_k combine the frequency of Q-table visits for TC suffixes and exponential decay, which reduces the influence of shorter TC suffixes. This approach allows the agent to use information from previous states.

Procedure 1 for multi-layer generalization of Q-values based on current state suffixes is shown in Fig. 2.

Procedure 1. Multi-layer generalization of Q-values based on current state suffixes **(BlendQ-Table)**

Input data:
- Q – Q-table
- A – set of available actions
- s_t – current state of the agent
- λ – decay coefficient of shorter suffix influence
- β – experience amplification coefficient

Output data:
- $\tilde{Q}$ – table of weighted average Q-values

Procedure body:

if $\mathsf{A} = \varnothing$ ***then return None***
local $w_{sum} \leftarrow \varnothing, w_a \leftarrow \varnothing, \tilde{Q} \leftarrow \varnothing$
for each $a \in \mathsf{A}$ ***do***
for $i = 1$ ***to*** $Len\left(s_t\right)$ ***do***
local $s_t^i \leftarrow \text{Equation\_4}(TC, i)$
local $s_t^i \leftarrow \text{Equation\_4}(TC, i)$
if $Q\left(s_t^i, a\right) != 0$ ***then***
local $N_i \leftarrow CountUpdates\left(Q, s_t^i\right)$
local $w_i \leftarrow \left(N_i + 10^{-6}\right)^{\beta} \cdot \lambda^{k+1-i}$
$w_a(a) \leftarrow w_a(a) + w_i \cdot Q\left(s_t^i, a\right)$
$w_{sum}(a) \leftarrow w_{sum}(a) + w_i$
end if
end for
if $w_{sum}(a) > 0$ ***then***
$\tilde{Q}(a) \leftarrow w_a(a) / w_{sum}(a)$
else
$\tilde{Q}(a) \leftarrow 10^{-6}$
end if
end for
return $\tilde{Q}$

Fig. 2. Pseudocode of the procedure for multi-layer generalization of Q-values based on suffixes of the current state

Weighted averaging by Q-table visit frequency comes from [31, 32], which considers the role of visit frequency in stabilizing Q-values and preventing overestimation.

During training, the agent applies an ε-greedy policy in which actions are selected based on multi-layer averaging of Q-values calculated for all suffixes of the current state according to equation (8). The probability of selecting an action $a_t \in \mathrm{A}(s_t)$ in state s_t is defined as:

$$\mathrm{a}_t = \begin{cases} \arg\max\limits_{\mathrm{a}' \in \mathrm{A}(\mathrm{s}_t)} \tilde{\mathrm{Q}}(\mathrm{s}_t, \mathrm{a}'), & (1-\varepsilon), \\ \mathrm{rand}\ \mathrm{A}(\mathrm{s}_t) & \varepsilon, \end{cases} \tag{10}$$

where $\varepsilon \in (0,1)$ is the coefficient of stochasticity of the action space exploration.

Thus, formulas (8), (9), and (10) describe the modified ε-greedy policy of the agent, in which the choice of action is made based on the averaged Q-values for all state suffixes. This policy combines the simplicity of classical Q-learning with the generalization of a multi-layer Q-table, increasing the stability of learning and the accuracy of test case construction.

Algorithm 1 for agent training based on Q-learning is shown in Fig. 3, in which, unlike the classical version, the TDE is distributed among all suffixes of the current state.

Algorithm 1. Agent training algorithm with multi-layer Q-table update

Input data:
- Q – initial Q-table
- TC_0 – original test case
- $k \in \mathrm{N}, k > 0$ – call history length
- $\alpha_0 \in (0,1]$ – initial learning rate
- $\alpha_{final} \in (0,1]$ – final learning rate
- $\varepsilon_0 \in (0,1)$ – initial exploration probability
- $\varepsilon_{final} \in (0,1)$ – final probability of investigation
- $\gamma \in (0,1)$ – discount factor
- $N_{ep} \in \mathrm{N}$ – number of learning episodes
- $N_{retries} \in \mathrm{N}$ – maximum number of retries
- λ – decay coefficient of shorter suffixes influence
- β – experience amplification coefficient

Initial data:
- Q – trained Q-table.

Algorithm body:

1. ***for*** $ep = 1$ ***to*** N_{ep} ***do***
2. ***local*** $TC \leftarrow \varnothing$
3. ***local*** $\mathrm{A}_{ep} \leftarrow \mathrm{GetActionSpace}(TC_0)$
4. ***local*** $C_{orig} \leftarrow GetCoverage(TC_o)$
5. ***local*** $C_t \leftarrow \varnothing$
6. ***local*** $\alpha_{ep} \leftarrow \mathrm{Linear\ Decay}(N_{ep}, ep, \alpha_0, \alpha_{final})$
7. ***local*** $\varepsilon_{ep} \leftarrow \mathrm{LinearDecay}(N_{ep}, ep, \varepsilon_0, \varepsilon_{final})$
8. ***local*** $R_{total} \leftarrow 0,\ \mathrm{Loss} \leftarrow 0,\ s_t \leftarrow \varnothing, a_t \leftarrow \varnothing$
9. ***while*** $C_t < C_{orig}$ ***and not*** $\mathrm{Over}(\mathrm{A}_{ep})$ ***do***
10. ***local*** $N_{retry} \leftarrow 0$
11. ***local*** $\tilde{Q} \leftarrow BlendQTable(Q, s_t, \mathrm{A}_{ep}, \lambda, \beta)$
12. ***while*** $N_{retry} < N_{retries}$ ***do***
13. $a_t \leftarrow \mathrm{Equation\_10}(\tilde{Q}, s_t, \mathrm{A}_{ep}, \varepsilon_{ep})$
14. $TC \leftarrow TC \shuffle a_t$
15. ***if*** $isValid\ (TC)$ ***then***
16. $\mathrm{A}_{ep} \leftarrow \mathrm{A}_{ep} / a_t$
17. ***break***
18. ***else***
19. $TC \leftarrow TC / a_t$
20. $N_{retry} \leftarrow N_{retry} + 1$
21. ***end if***
22. ***end while***
23. ***if*** $N_{retry} > N_{retries}$ ***then***

Fig. 3. Pseudocode of the agent learning algorithm based on Q-learning with multi-layer Q-table update *(Beginning)*

Algorithm 1. Agent training algorithm with multi-layer Q-table update

24. ***break***
25. ***end if***
26. $s_{t+1} \leftarrow \text{Equation\_4}(TC,k)$
27. ***local*** $r_t \leftarrow \text{Execute}(TC)$
28. ***local*** $\delta^t \leftarrow r_t + \gamma \times \max Q(s_{t+1}, a') - Q(s_t, a_t)$
29. ***for*** $i = 1$ ***to*** k ***do***
30. $s_t^i \leftarrow \text{Equation\_4}(TC,i)$
31. $Q(s_t^i, a_t) \leftarrow Q(s_t^i, a_t) + \alpha \times \lambda^{k+1-i} \times \delta^t$
32. ***end for***
33. $R_{total} \leftarrow R_{total} + r_t$
34. $\text{Loss} \leftarrow \text{Loss} + \text{L}(Q, r_t, a_t, s_t, s_{t+1}, \gamma)$
35. $s_t \leftarrow s_{t+1}$
36. $C_t \leftarrow GetCoverage(TC)$
37. ***end while***
38. $\text{Log}(\text{Loss}, R_{total})$
39. ***end for***
40. ***return*** Q

Fig. 3. Pseudocode of the agent learning algorithm based on Q-learning with multi-layer Q-table update *(The end)*

Rule (7) is used to update the Q-table, and policy (10) is used to select actions. During learning, the learning rate and exploration probability change according to a linear decay law, which ensures a balance between exploration and exploitation of actions.

The proposed Procedure 1 implements a mechanism of multi-layer generalization of Q-values for a set of available actions. For each suffix of the current state s_t^i, a weight coefficient w_i is calculated, which takes into account the frequency of Q-table updates and the distance to the current state according to (9).

The averaged estimates for each action $\tilde{Q}(a)$ are formed as a normalized weighted average of Q-values across all suffixes. The resulting table is used by the agent to make decisions during training or TC forming according to ε-greedy policy.

Thus, the proposed model provides a hierarchical generalization of the Q-table, allowing the agent to use the knowledge acquired in previous states during forming new sequences of actions. Such a multilayered representation of states increases resistance to high sparsity of the agent's state space and improves the agent's adaptation to heterogeneous API environments.After the training stage, the agent uses the obtained Q-table to form new TC. The pseudocode of Algorithm 2 for forming TC is shown in Fig. 4.

Unlike the learning process, which uses an ε-greedy policy, at the TC formation stage, actions are selected using a softmax policy, which provides a smoother balance between exploration and exploitation of already accumulated knowledge.

The probability of choosing an action $a_t \in \text{A}(s_t)$ in state s_t can be determined as follows:

$$\text{P}(\text{a}_\text{t}|\text{s}_\text{t}) = \frac{\text{e}^{\tilde{\text{Q}}(\text{s}_\text{t},\text{a}_\text{t})/\tau}}{\sum_{\text{a}' \in \text{A}(\text{s}_\text{t})} \text{e}^{\tilde{\text{Q}}(\text{s}_\text{t},\text{a}')/\tau}}, \tag{11}$$

where τ is a temperature parameter that regulates the influence of stochasticity.

In Algorithm 2, the agent sequentially builds a new TC using the trained Q-table. At each iteration, it forms the current state based on the latest calls, performs multi-layer generalization of

Q-values according to Procedure 1, and then selects the next action according to policy (11). To ensure the correctness of the formed test case, a mechanism for rolling back actions and checking the admissibility of transitions is provided.

The algorithm ends when the reference coverage is reached or the action space is exhausted.

Algorithm 2. Algorithm for forming a test case

Input data:

– Q – Q-table

– TC_0 – original test case

– $N_{retries} \in \mathsf{N}$ – max. number of retries

– τ – temperature for softmax action selection

– λ – decay coeff of shorter suffix influence

– β – experience amplification coefficient

– δ – coverage stability tolerance.

Initial data:

– TC – formed test case

Algorithm body:

1. $TC \leftarrow \varnothing$
2. $k \leftarrow \text{Extract}(Q)$
3. $C_{orig} \leftarrow GetCoverage(TC_o)$
4. $\mathsf{A} \leftarrow \text{GetActionSpace}(TC_o)$
5. $C_t \leftarrow 0$
6. ***while*** $C_t < C_{orig}$ ***and not*** $\text{isOver}(\mathsf{A})$ ***do***
7. ***local*** $N_{retry} \leftarrow 0$
8. ***local*** $s_t \leftarrow \text{Equation\_4}(TC, k)$
9. ***local*** $a_t \leftarrow 0$
10. ***local*** $\tilde{Q} \leftarrow BlendQTable(Q, s_t, \mathsf{A}, \lambda, \beta)$
11. ***while*** $N_{retry} < N_{retries}$ ***do***
12. $a_t \leftarrow \text{Equation\_11}(\tilde{Q}, s_t, \mathsf{A}, \tau)$
13. $TC \leftarrow TC \mathbin{\#} a_t$
14. ***if*** $isValid(TC)$ ***then***
15. $\mathsf{A} \leftarrow \mathsf{A} / a_t$
16. ***break***
17. ***else***
18. $TC \leftarrow TC / a_t$
19. $N_{retry} \leftarrow N_{retry} + 1$
20. ***endif***
21. ***end while***
22. ***if*** $N_{retry} > N_{retries}$ ***then***
23. ***break***
24. ***endif***
25. $C_t \leftarrow GetCoverage(TC)$
26. ***if*** $C_t > C_{orig} + \delta$ ***then***
27. ***break***
28. ***endif***
29. ***end while***
30. ***return*** TC

Fig. 4. Pseudocode of the test scenario formation algorithm

Thus, within the scope of the study, model [1] has been extended to improve the stability of learning and generalization of the action selection policy.

The main changes include the following:

– *multi-layer updating of the Q-table:* in the proposed version, the agent distributes the temporal difference error among all suffixes of the current state, rather than just for one suffix of length k, which allows the context of previous partial test case to be taken into account and ensures better policy generalization;

– *averaging of Q-values:* for each state, a weighted generalization of estimates for all suffixes is performed using weight coefficients defined in (9);

– *action selection policy:* during training, an ε-greedy policy is used, taking into account the averaged Q-values, and during the TC formation the softmax policy is used (11).

3. Evaluation of the effectiveness of TC optimization algorithms and methods

The following three basic metrics are used to evaluate the effectiveness of TC optimization algorithms and methods [3]:

1. Retention of Coverage (RC) coefficient – evaluates the increase in original TS branch coverage, reflecting the loss or retention of testing efficiency:

$$RC = Cov(TS_{out}) \Big/ Cov(TS_{orig}), \tag{12}$$

where $Cov(TS_{out})$ is the TS coverage after optimization, $Cov(TS_{orig})$ is the coverage of the original TS;

2. Compression Ratio (CR) – evaluates the reduction in the number of test case instructions:

$$CR = \left(Len\left(TS_{orig}\right) - Len\left(TS_{out}\right)\right) \Big/ Len\left(TS_{out}\right), \tag{13}$$

where $Len\left(TS_{orig}\right)$ is the number of instructions in the original TS, $Len\left(TS_{out}\right)$ is the number of instructions after optimization.

3. Execution Cost Reduction (ECR) – estimates the reduction in time required to execute the test suite:

$$\mathrm{ECR} = \left(\mathrm{T}\left(\mathrm{TS}_{\mathrm{orig}}\right) - \mathrm{T}\left(\mathrm{TS}_{\mathrm{out}}\right)\right) \Big/ \mathrm{T}\left(\mathrm{TS}_{\mathrm{out}}\right), \tag{14}$$

where $T\left(TS_{orig}\right)$ is the time required to execute the original TS, and $T\left(TS_{out}\right)$ is the time required to execute the TS after optimization.

To evaluate the effectiveness of the proposed method, three basic classical methods of TC optimization are used: Greedy Reduction (GR**)**, Delta Debugging (DD), and Monte Carlo Tree Search (MCTS**)**.

Greedy Reduction. The GR method forms a TC by sequentially selecting actions that provide the greatest increase in branch coverage. At each iteration, an action is selected $a^* \in \mathrm{A}$, which maximizes the coverage difference:

$$\mathrm{a}^* = \arg\max_{\mathrm{a}' \in \mathrm{A}} \left(\mathrm{Cov}\left(\mathrm{TC} \oplus \mathrm{a}'\right) - \mathrm{Cov}\left(\mathrm{TC}\right)\right). \tag{15}$$

Greedy optimization methods are widely used in TS minimization problems due to their simplicity of implementation. However, they are prone to getting stuck in local optima and are sensitive to the initial order of actions [21].

Delta debugging. The DD method is based on iterative removal of actions from the TC to eliminate redundancy without losing testing efficiency. At each step, the equivalence of the specified criterion between the initial and reduced TC is checked. In particular, the execution error is preserved [33].

Search in the Monte Carlo tree. The MCTS method constructs a tree of possible test cases, where nodes correspond to states and edges correspond to API actions.

Actions are selected based on the Upper Confidence Bound (UCB**)** criterion [18, 34]:

$$\mathrm{UCT}(v_i) = \frac{w_i}{n_i} + c \times \sqrt{\frac{\ln N}{n_i}}, \tag{16}$$

where w_i is the total reward for node v_i, n_i is the number of visits to node v_i, N is the number of visits to the parent node, and $c > 0$ is a parameter that controls the balance between exploitation and exploration.

After selecting a branch, the TC simulation stage is performed, the reward for the achieved coverage is evaluated, and the reward is redistributed according to the following formula:

$$n_i \leftarrow n_i + 1, \quad w_i \leftarrow w_i + r_i, \tag{17}$$

where r_i is the simulation reward.

MCTS effectively explores a large space of possible test cases and gradually refines the policy of action selection, making it suitable for use as the main method of TC optimization.

Research results

1. Environment setup

To evaluate the effectiveness of the methods, two open-source C++ libraries were selected, which differ in scale and structural complexity. The main characteristics are shown in Table 2.

The *lizard* utility was used to analyze the number of lines of code and cyclomatic complexity of the target libraries. The following abbreviations are used in Table 2:

– API – number of public functions of the target C++ library;
– LoC – number of lines in the source code;
– TCN – number of test cases in the test suite;
– IC – total number of instructions in the test suite;
– BC – branch coverage of original test suite;
– Avg.CCN – average cyclomatic complexity of the library functions.

Table 2. *Characteristics of C++ libraries selected for research*

Library	API	LoC	TCN	IC	BC, %	Avg. CCN
Bitmap PlusPlus	33	760	7	71	27	2.5
Hjson	122	3936	57	1259	36.1	5.4

The evaluation was performed on a workstation with an Intel Core i7 processor (6 cores, 2.6 GHz) and 16 GB of RAM.

The libraries and their test suites were compiled using AppleClang 15.0.0 (LLVM 15) with coverage profiling options (*-fprofile-instr-generate fcoverage-mapping*) enabled. The *llvm-profdata* and *llvm-cov* utilities were used to collect and analyze branch coverage.

The parameters for training the Q-learning agent are consistent with the previous implementation described in [1] and ensure stable convergence of the utility function in conditions of high dimensionality of the action space. Table 3 lists the configuration parameters used for all experiments, unless otherwise specified.

Table 3. *Configuration of the Q-learning agent training according to Algorithm 1*

Parameter name	Designation	Value
Number of episodes	N_{ep}	1000
Maximum number of retries	$N_{retries}$	30
Initial probability of investigation	ε_0	1
Final probability of investigation	ε_{final}	0
Initial learning rate	α_0	0.3
Final learning rate	α_{final}	0.1
Discount factor	γ	0.85
Experience amplification factor	β	1

2. Analysis of the impact of parameters of the improved Q-learning agent model

The parameter k determines the length of the call history used in forming the agent's state. To study its impact, modeling was performed on the Hjson and BitmapPlusPlus libraries, where the relative frequency of state-action pairs occurrences in the Q-table during training in dependence to test case suffix length was analyzed.

Fig. 5, *a* shows the dependence of the relative frequency of finding records (suffixes) in the Q-table on the parameter k for the BitmapPlusPlus library.

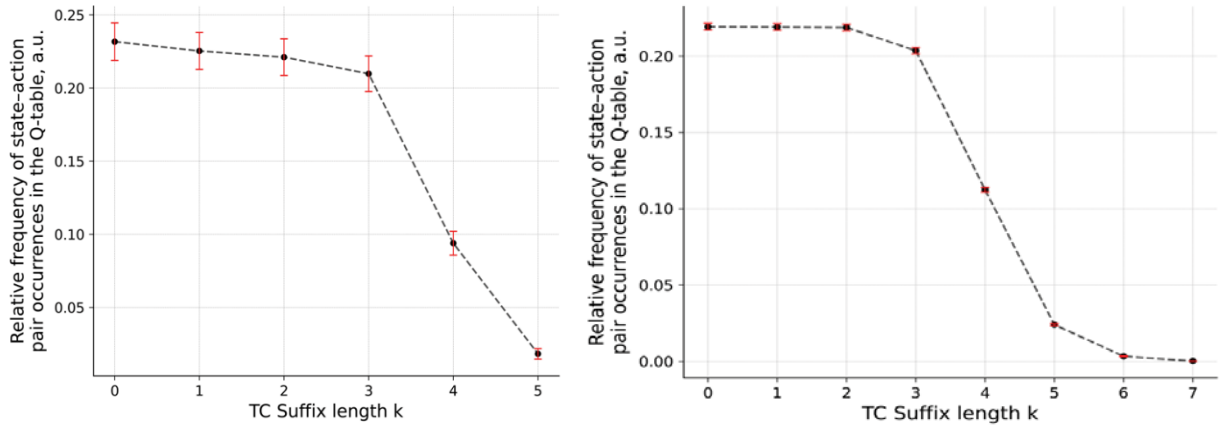
It can be observed that as k increases, the number of records found in the Q-table decreases, which indicates an increase in the sparsity of the agent's state space. For $k \geq 4$, most suffixes occur rarely, which leads to a deterioration in policy generalization.

Fig. 5, *b* shows a similar dependence for the Hjson library, which is characterized by a more branched API call structure. Up to $k \leq 3$, the proportion of states found remains virtually constant. Starting from $k = 4$, there is a sharp drop in relative frequency, and for $k \geq 6$, most suffixes occur very rarely, indicating a loss of the agent's generalization ability, which begins to accumulate unique but uninformative states.

Thus, the optimal choice is the value of the suffix $k = 5$, which provides a compromise between context depth and stability of Q-value updates.

To assess the impact of the parameter λ on learning efficiency, we chose the indicator of the proportion of suffixes found, i.e., the fraction of the discovered state-action pairs in the Q-table, e.g. the states that the agent has already encountered in previous learning episodes to the total number of transitions between the agent's states during training.

This indicator allows us to quantitatively assess the agent's ability to reuse experience when making decisions.

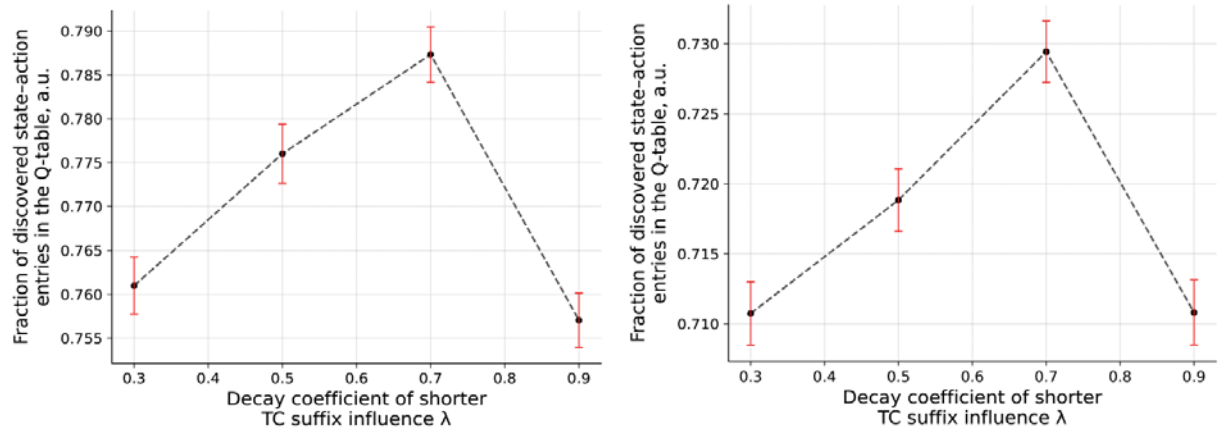


a) b)

Fig. 5. Relative frequency of state-action pairs occurrences in the Q-table during training of the Q-learning agent on the Hjson (b) and BitmapPlusPlus (a) libraries

Figure 6, *a* shows the results for the BitmapPlusPlus library. As λ increases, the proportion of suffixes found initially increases, reaching a maximum at $\lambda = 0,7$, indicating the most efficient reuse of experience. A further increase in λ causes a decrease in the indicator, as the agent begins to rely too much on the experience of shorter suffixes, reducing the number of transitions learned.

A similar trend is observed for the Hjson library (Fig. 6, *b*). An increase in the proportion of suffixes found to $\lambda \le 0,7$ indicates a gradual increase in policy stability and improved consistency of Q-table updates. At $\lambda = 0,9$, the indicator decreases slightly.



a) b)

Fig. 6. Dependence of the average fraction of discovered state-action entries int the Q-table on the coefficient λ during training on the Hjson (b) and BitmapPlusPlus (a) libraries

Thus, a comparative analysis of the two C++ libraries indicates that excessively low values of λ lead to a loss of generalization, while excessively high values lead to a decrease in the use of knowledge from the Q-table. The optimal choice is a decay coefficient for shorter suffixes of TC equal to $\lambda = 0,7$. This value provides the best compromise between exploiting previous experience and exploring new states, which confirms the stability of the policy for different types of C++ libraries.

3. Evaluation of the effectiveness of the TC formation method for C++ libraries based on a Q-learning agent

For comparative analysis, three basic methods and two groups of configurations of the proposed method were used:

– configuration of the classic greedy algorithm for TC formation (GR);

– configuration of the classic delta debugging algorithm for TS minimization (DSL);

– configurations for forming TC using the Monte Carlo tree search method (MCTS1, MCTS2, MCTS3);

– configurations of the TC formation method based on a Q-learning agent (QLB-M1, QLB-M2, QLB-M3);

– configurations of the TC formation method based on a Q-learning agent and a post-processing filter based on the delta debugging algorithm for TS minimization (QLB-MD1, QLB-MD2, QLB-MD3).

For each algorithm configuration, 100 independent mathematical simulations were performed on each target library. The configuration parameters are shown in Tables 4 and 5.

Table 4. *Configurations for TC formation by a Q-learning agent according to Algorithm 2*

Configuration	Temperature, τ	Coefficient, λ	Coefficient, β
QLB-M1	1.5	0.7	1
QLB-M2	3	0.7	1
QLB-M3	5	0.7	1

In the current implementation, the experience amplification factor β is fixed at 1, which corresponds to linear consideration of the update frequency without additional experience amplification.

Thus, the weights of suffixes are determined only by their visit frequency and exponential decay based on the λ parameter.

Table 5. *Configurations for forming TC using the Monte Carlo Tree Search method*

Configuration	Coeff., c	Number of iterations
MCTS1	0.7	200
MCTS2	1.4	200
MCTS3	2.0	200

The distribution of branch coverage metrics for the BitmapPlusPlus library (Fig. 7, *a*) shows that the classical DSL and GR methods consistently maintain the initial coverage level corresponding to the original TS.

The MCTS method demonstrates a moderate increase in average coverage.

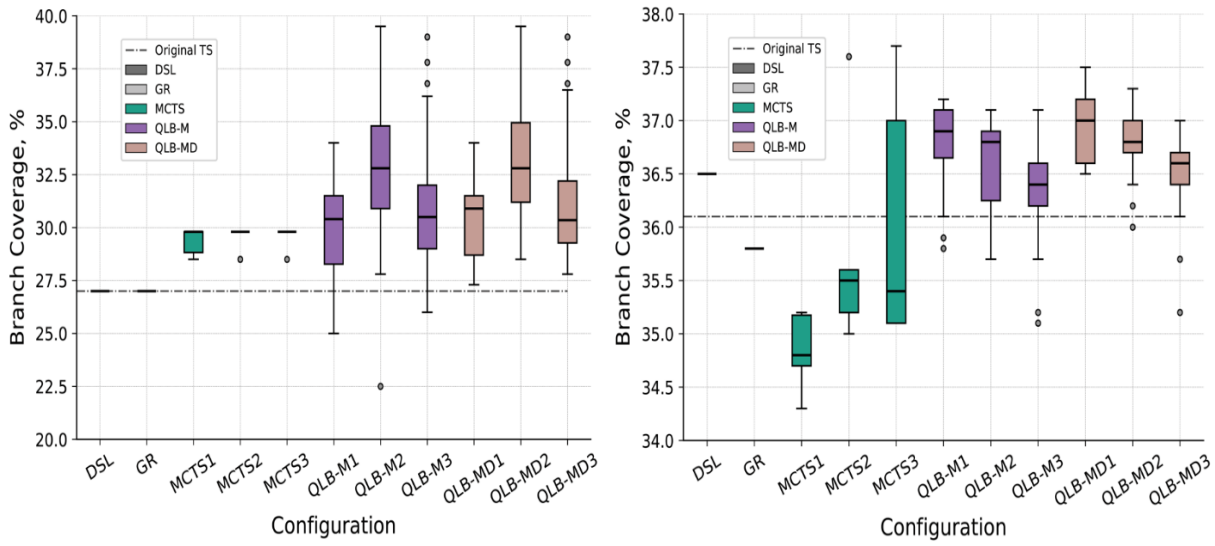
However, it is characterized by limited variability of results due to the stochastic, but not learning, nature of the search.

The proposed QLB-M and QLB-MD configurations provide a significant improvement in performance, with median coverage values exceeding 32 %. This improvement is explained by the ability of the Q-learning agent to generalize the experience of previous episodes, and the subsequent application of delta debugging allows to maintain and increase coverage after reducing the TC.

A similar trend is observed for the Hjson library (Fig. 7, *b*). The classical DSL and GR methods hardly change the initial coverage level (36.1 %), while MCTS provides a non-stable increase to 37 %.

In contrast, the proposed QLB-M and QLB-MD configurations demonstrate a systematic increase in both the median and upper quartiles of the distribution, reaching a peak value of 37.5 %.

Thus, the agent is capable of effectively reproducing complex combinations of API calls in libraries with deep structural dependencies.



a) b)

Fig. 7. Distribution of branch coverage metrics depending on the configuration of the TC optimization method for the BitmapPlusPlus (a) and Hjson (b) libraries**,** where: the dotted line indicates the coverage level of the original TS

Fig. 8 shows the results of comparing the dynamics of branch coverage growth during the execution of TC formed using different optimization methods.

The curves obtained reflect the relationship between the number of instructions executed and the achieved level of branch coverage of the code.

The dotted horizontal line indicates the coverage level of the original TS, and its intersection with the curve of a specific method corresponds to the number of instructions for which adequate compression ratio is maintained without loss of code coverage.

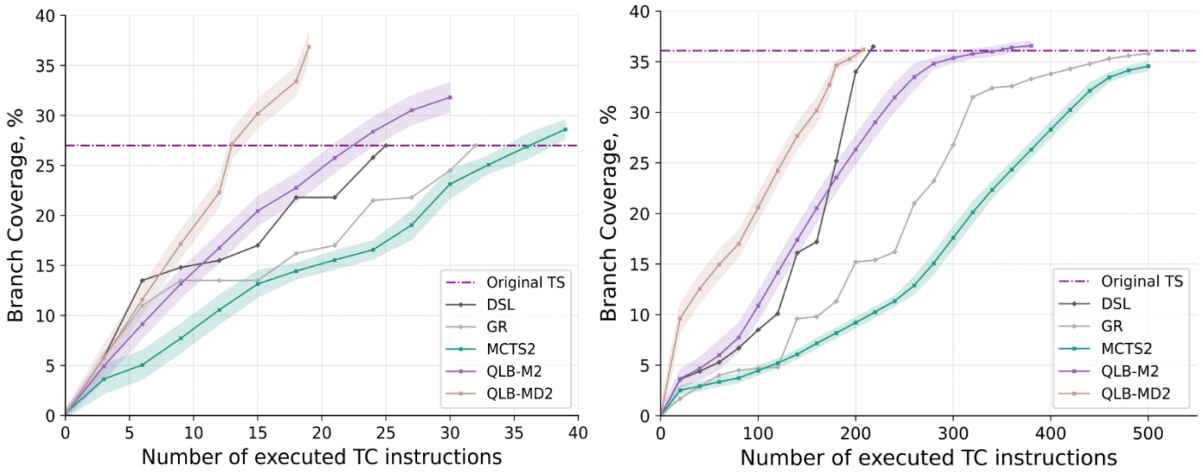


Fig. 8. Dependence of branch coverage on the number of executed TC instructions for different TC optimization methods for BitmapPlusPlus (a) and Hjson (b) libraries

Fig. 9 shows the dependence of the compression ratio on the configuration of TC optimization methods for the BitmapPlusPlus and Hjson libraries.

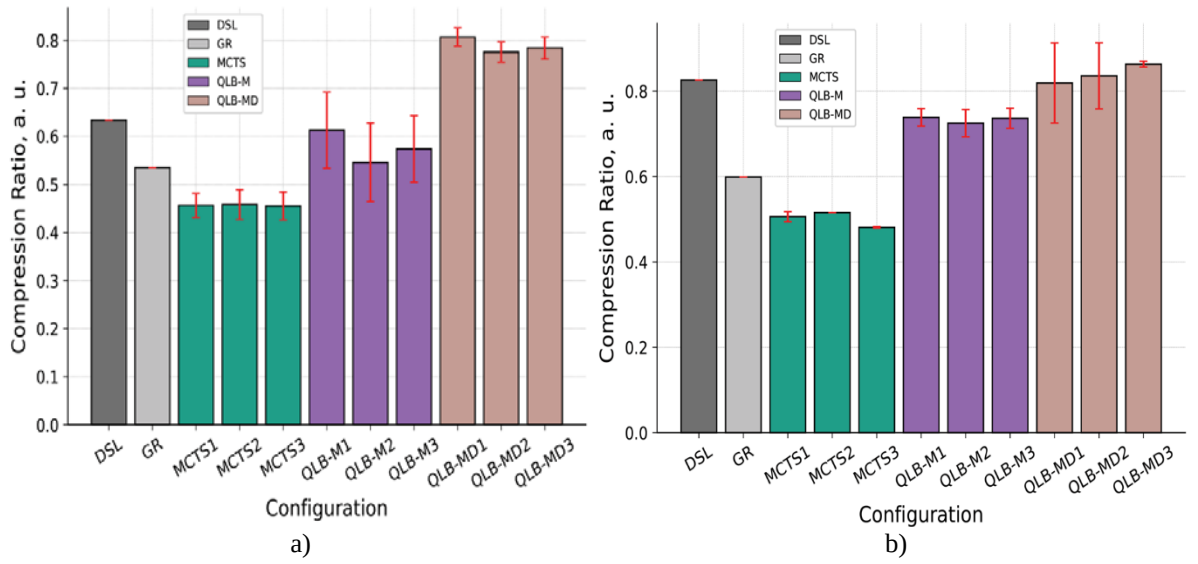


Fig. 9. Compression ratio depending on the configuration of optimization methods for the BitmapPlusPlus (a) and Hjson (b) libraries, where: the red segments on the columns indicate the standard error

For the BitmapPlusPlus library (Fig. 9, *a*), there is a gradual increase in the compression ratio from classical methods (DSL, GR) to the proposed configurations (QLB-M, QLB-MD).

The DSL and GR methods show average values of 0.5–0.6, which corresponds to the basic level of reduction without taking into account the dependencies between API calls.

The MCTS method shows similar results (≈ 0.45), since stochastic TC formation does not guarantee avoidance of redundant actions.

The proposed QLB-M method improves this indicator to 0.55–0.6.

The highest results are achieved in QLB-MD configurations, where the delta debugging algorithm is applied as a post-processing filter after the Q-learning agent formation stage. This approach enables the median compression ratio values exceed 0.8, which indicates a significant reduction of TC without loss of coverage.

For the Hjson library (Fig. 9, *b*), the indicators turned out to be more variable due to the more complex structure of API calls and the greater depth of the call tree.

The basic DSL method shows the highest compression ratio (~ 0.83), while GR is significantly lower (~ 0.6) due to the lack of consideration of dependencies between actions.

Stochastic MCTS configurations remain within the range of 0.48–0.5, reflecting low reduction efficiency without explicit coverage analysis.

QLB-M configurations show a steady increase in compression ratio (≈ 0.7–0.75), while QLB-MD combinations show the highest results (0.82–0.86) with low variance, indicating the stability of the effect after training.

Fig. 10 shows the results of evaluating the average execution time of TS for the BitmapPlusPlus and Hjson libraries.

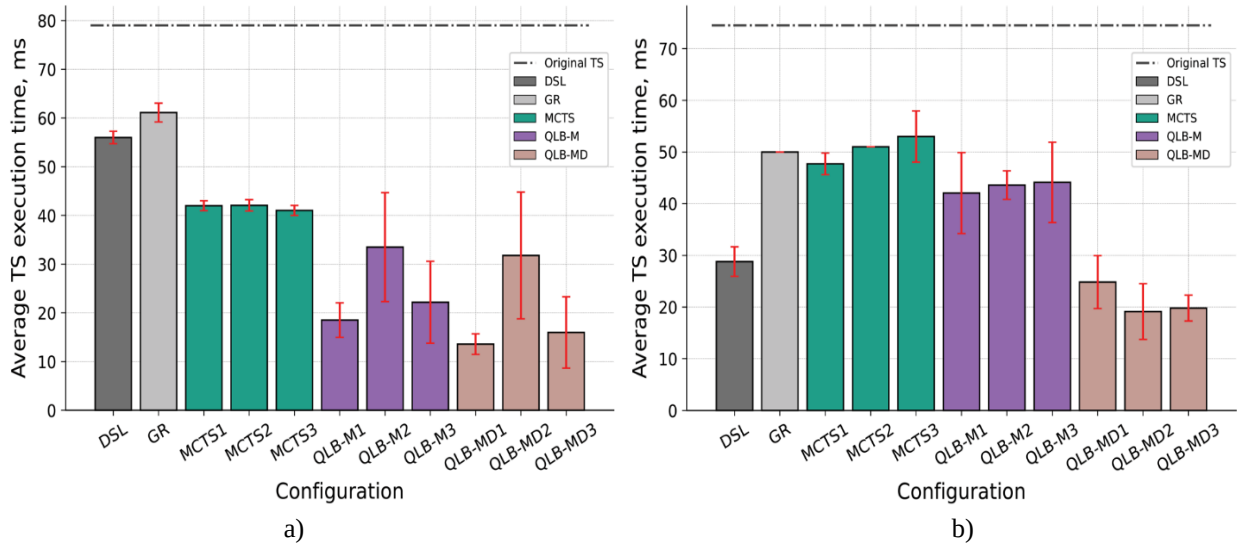


Fig. 10. Average TS execution time depending on the configuration of TC optimization methods for the BitmapPlusPlus (a) and Hjson (b) libraries, where: the red segments on the columns represent the standard error, and the dotted line corresponds to the execution time of the original TS

For the BitmapPlusPlus library (Fig. 10, *a*), there is a clear decrease in execution time from classical methods (DSL and GR) to the proposed ones (QLB-M and QLB-MD).

The basic configurations show the biggest average time (55–60 ms).

The MCTS configurations show a reduction in time to 40–42 ms. However, they do not achieve significant improvements due to the lack of generalized experience.

The QLB-M method provides a further reduction in average time to 20–35 ms due to the ability of the Q-learning agent to build shorter and more focused sequences of actions.

QLB-MD configurations show the highest efficiency, where the combination of learning and

delta debugging reduces the average execution time to 12–30 ms. That is, almost three times less than the basic methods.

For the Hjson library (Fig. 10, *b*), the trend generally remains the same. However, the difference between configurations is less pronounced due to the greater depth of calls and structural complexity of the library.

The DSL method provides the lowest time among the classic basics (~ 30 ms), while GR shows higher values (~ 50 ms), which indicates its low stability when working with more complex C++ libraries.

The MCTS method maintains a time of 45–53 ms with little variation, while QLB-M gradually reduces it to 40–45 ms.

The QLB-MD configuration group provides the lowest results (18–25 ms) while maintaining a high level of coverage and compression ratio, indicating effective coordination of TC optimization processes and reduction of redundant calls in a complex environment.

To evaluate the effectiveness of the methods under study, a comparative analysis of the average TS formation time was performed (Fig. 11).

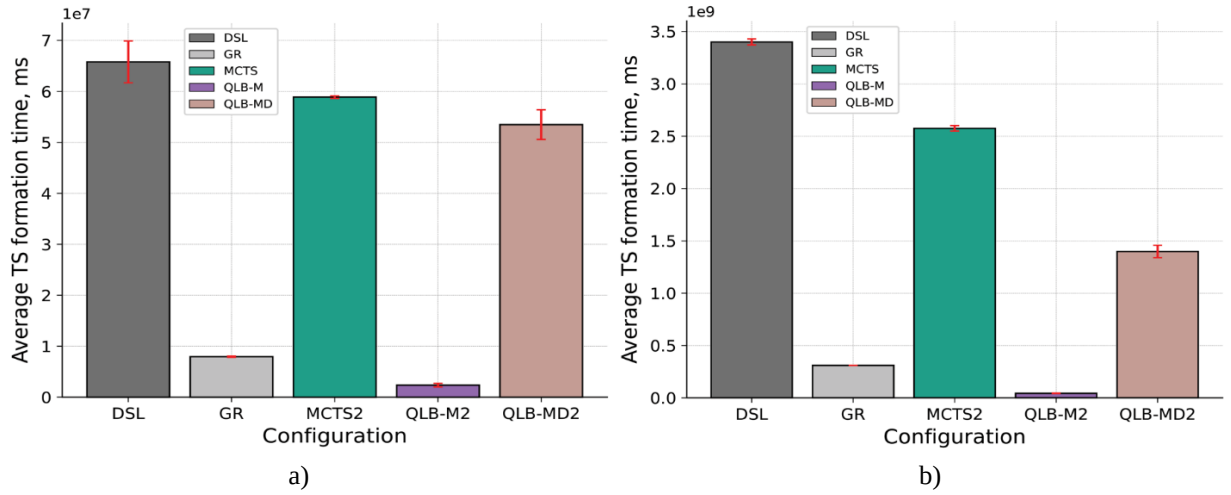


Fig. 11. Average TS formation time for TC optimization methods for BitmapPlusPlus (a) and Hjson (b) libraries

The graphs show that the basic DSL and MCTS2 methods have the highest time costs, while GR shows the fastest formation due to simple heuristics.

The improved QLB-M2 and QLB-MD2 methods show a significant reduction in formation time compared to the basic methods, while maintaining stability and moderate variability of results.

Tables 6 and 7 show the summary results of the evaluation of the effectiveness of the configurations of the studied methods for the C++ libraries BitmapPlusPlus and Hjson.

For the BitmapPlusPlus library (Table 6), the basic DSL and GR methods retain the original coverage (RC = 1), but are characterized by a moderate level of compression (CR = 0.54–0.63) and a low reduction in execution time (ECR ≈ 0.15–0.20).

MCTS search variants increase coverage by ≈ 9 % (to RC ≈ 1.09) and reduce execution time by ≈ 20 %. However, they do not provide a high value of compression ratio.

The proposed QLB-M configurations provide a coverage gain of up to ≈ 1.22, a compression ratio of 0.55–0.61, and a reduction in ECR execution time of ≈ 0.7–0.8.

The highest results were achieved for QLB-MD, where the combination of Q-learning with a delta debugging algorithm filter enables for a ≈ 22 % increase in branch coverage, compression ratio of up to 0.8, and a 70–85 % reduction in ECR execution time.

Table 6. *Evaluation of the effectiveness of the configurations of the studied methods for the BitmapPlusPlus library*

Configuration	RC	CR	ECR
DSL	1	0.63	0.19 ± 0.02
GR	1	0.54	0.15 ± 0.03
MCTS1	1.091 ± 0.02	0.46 ± 0.02	0.23 ± 0.015
MCTS2	1.092 ± 0.02	0.46 ± 0.03	0.23 ± 0.016
MCTS3	1.093 ± 0.02	0.46 ± 0.03	0.21 ± 0.015
QLB-M1	1.116 ± 0.08	0.61 ± 0.08	0.79 ± 0.07
QLB-MD1	1.127 ± 0.08	0.81 ± 0.02	0.85 ± 0.04
QLB-M2	1.215 ± 0.10	0.55 ± 0.08	0.62 ± 0.15
QLB-MD2	1.225 ± 0.10	0.78 ± 0.02	0.71 ± 0.18
QLB-M3	1.153 ± 0.09	0.57 ± 0.07	0.72 ± 0.13
QLB-MD3	1.163 ± 0.09	0.78 ± 0.02	0.78 ± 0.12

For the Hjson library (Table 7), the results are similar, but with less pronounced differences between methods due to the greater complexity of the internal structure of API calls.

Table 7. *Evaluation of the effectiveness of the configurations of the studied methods for the Hjson library*

Configuration	RC	CR	ECR
DSL	1.000	0.83	0.61 ± 0.05
GR	0.982	0.6	0.23 ± 0.00
MCTS1	0.969 ± 0.03	0.51 ± 0.01	0.39 ± 0.03
MCTS2	0.981 ± 0.03	0.52 ± 0.00	0.40 ± 0.005
MCTS3	0.988 ± 0.03	0.48 ± 0.00	0.34 ± 0.07
QLB-M1	1.008 ± 0.01	0.74 ± 0.02	0.41 ± 0.11
QLB-MD1	1.012 ± 0.01	0.82 ± 0.09	0.70 ± 0.07
QLB-M2	1.002 ± 0.01	0.73 ± 0.03	0.40 ± 0.04
QLB-MD2	1.008 ± 0.01	0.84 ± 0.08	0.71 ± 0.074
QLB-M3	0.996 ± 0.01	0.74 ± 0.02	0.42 ± 0.11
QLB-MD3	1.000 ± 0.01	0.86 ± 0.01	0.74 ± 0.039

The main DSL method maintains full coverage (RC = 1) and provides a high compression ratio (CR = 0.83) at ECR ≈ 0.61.

The GR method shows a decrease in coverage (RC = 0.98) and lower efficiency in reducing execution time (ECR ≈ 0.23).

MCTS search configurations achieve a moderate reduction in time (ECR ≈ 0.34–0.40) with stable but slightly lower coverage than the original TS.

QLB-M methods provide a balance between coverage (RC ≈ 1.00–1.01), compression ratio (0.73–0.74), and execution time reduction (ECR ≈ 0.4).

QLB-MD configurations show the highest results: RC ≈ 1.00–1.01, CR ≈ 0.84–0.86, ECR ≈ 0.70–0.74, which confirms the stability and effectiveness of the proposed method in conditions of deeper data structures in complex C++ libraries.

Thus, for both libraries, the QLB-MD configurations are the most effective, providing a balanced increase in coverage (up to + 22.5 %), compression ratio (up to 0.86), and a significant reduction in TS execution time.

To summarize the results of the comparison of methods, radial diagrams were built (Fig. 12), which reflect the relationship between the three main coefficients of effectiveness for the methods under study.

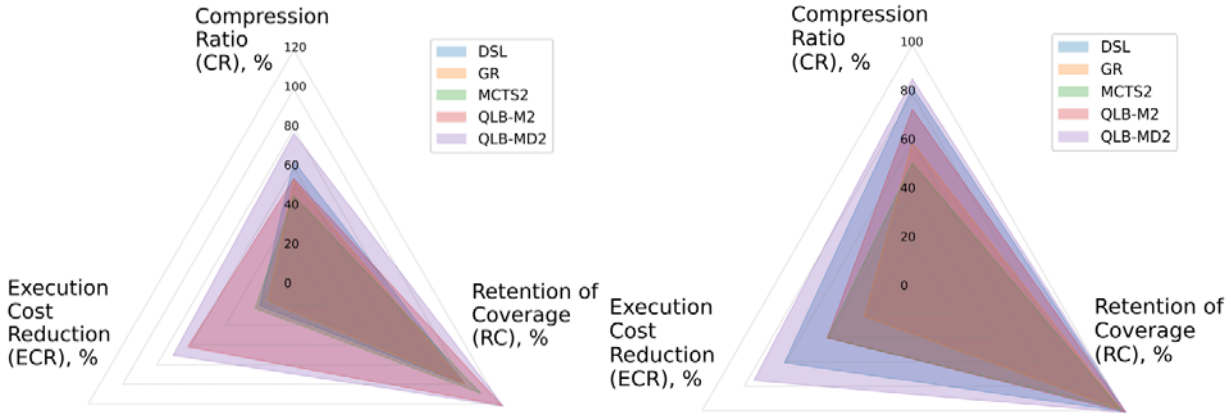


Fig. 12. Radial diagrams of the effectiveness of methods for the BitmapPlusPlus (a) and Hjson (b) libraries

Thus, the combination of the Q-learning mechanism with delta debugging action filtering provides comprehensive optimization of the TC and increases the efficiency of the testing process without losing coverage quality.

Conclusions

Thus, this article evaluates the effectiveness of the method of forming TC for C++ libraries based on a Q-learning agent. The proposed method combines step-by-step formation of function call sequences based on Q-learning with a post-processing filter based on the delta debugging algorithm.

Unlike classical greedy and search-based optimization methods, the proposed agent adaptively takes into account previous experience, balancing between exploring new states and using already known trajectories, which increases the ability to generalize experience and the stability of learning.

Analytical research of the parameters of the mathematical model indicates that the optimal length of the call history k is 5, since with a deeper history there is a sharp decrease in the frequency of finding states in the Q-table.

Research into the parameter of the mathematical model of the decay of shorter suffixes λ shows maximum efficiency at a value of 0.7, when the agent demonstrates the highest proportion of use of accumulated experience, reflecting the system's ability to maintain a balance between stability and flexibility of learning.

A comparative assessment of effectiveness with the main methods showed the advantage of the developed method in all key testing effectiveness criteria, which have the following average values: coverage retention coefficient up to 1.225, compression ratio up to 0.86, and TS execution time reduction ratio up to 0.74.

The improved mathematical model provides generalization of the Q-learning agent's experience between similar TC and increases the efficiency of their formation in conditions of high sparsity of the state space of the Q-learning agent. Unlike the classical mathematical model of Q-learning, in which Q-values are updated only for the current state of the agent (test case suffix), in the proposed model, the temporal difference error is distributed among all test case suffixes with the decay of influence of the shorter suffixes. Thus, according to the results of evaluating the effectiveness of the method of forming TC for C++ libraries based on a Q-learning agent, its effectiveness in solving the problem of forming (optimization) of TC for C++ libraries, which allows reducing the length of original TS without losing the branch coverage of the C++ library code being tested and reducing its execution time.

Further research directions may include the formation of test cases for C++ libraries based on a deep learning agent with reinforcement and the application of the proposed method of forming (optimizing) test cases for software developed using other programming languages.

References

1. Semenov, S., Kolomiitsev, O., Hulevych, M., Mazurek, P., Chernyk, O. (2025), "An Intelligent Method for C++ Test Case Synthesis Based on a Q-Learning Agent", *Applied Sciences*, Vol. 15(15), Art. No. 8596. DOI: https://doi.org/10.3390/app15158596
2. Alian, M., Suleiman, D., Shaout, A. (2016), "Test Case Reduction Techniques – Survey", *International Journal of Advanced Computer Science and Applications*, Vol. 7(5), P. 264–275. DOI: https://doi.org/10.14569/IJACSA.2016.070537
3. Khan, S. U. R., Lee, S., Javaid, N., Abdul, W. (2018), "A Systematic Review on Test Suite Reduction: Approaches, Experiment's Quality Evaluation, and Guidelines", *IEEE Access*, Vol. 6, P. 11816-11841. DOI: https://doi.org/10.1109/ACCESS.2018.2809600
4. Rahman, M., Zamli, K., Kader, M., Sidek, R., Din, F. (2024), "Comprehensive Review on the State-of-the-arts and Solutions to the Test Redundancy Reduction Problem with Taxonomy", *Journal of Advanced Research in Applied Sciences and Engineering Technology*, Vol. 35(1), P. 62–87. DOI: https://doi.org/10.37934/araset.34.3.6287
5. Marappan, R., Raja, S. (2025), "Recent Trends in Regression Testing: Modeling and Analyzing the Critiques in Selection, Optimization, and Prioritization", *National Academy Science Letters*. DOI: https://doi.org/10.1007/s40009-025-01613-6

6. Fontes, A., Gay, G. (2023), "The integration of machine learning into automated test generation: A systematic mapping study", *Software Testing, Verification and Reliability,* Vol. 33(4), e1845. DOI: https://doi.org/10.1002/stvr.1845
7. Sebastian, A., Naseem, H., Catal, C. (2024), "Unsupervised Machine Learning Approaches for Test Suite Reduction", *Applied Artificial Intelligence*, Vol. 38(1), Art. No. 2322336. DOI: https://doi.org/10.1080/08839514.2024.2322336
8. Nayab, S., Wotawa, F. (2024), "Testing and Reinforcement Learning: A Structured Literature Review", *2024 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Cambridge, United Kingdom, 2024, P. 326-335. DOI: https://doi.org/10.1109/QRS-C63300.2024.00049
9. Coviello, C., Romano, S., Scanniello, G., Marchetto, A., Corazza, A., Antoniol, G. (2019), "Adequate vs. inadequate test suite reduction approaches", *Information and Software Technology,* Vol. 119, Art. No. 106224. DOI: https://doi.org/10.1016/j.infsof.2019.106224
10. Hulevych, M., Kolomiitsev, O. (2025), "Automated Test Generation Techniques for C++ Software", *Control, Navigation and Communication Systems*, Vol. 2(80), P. 102-107. DOI: https://doi.org/10.26906/SUNZ.2025.2.102
11. Pan, R., Ghaleb, T. A., Briand, L. (2023), "ATM: Black-box Test Case Minimization Based on Test Code Similarity and Evolutionary Search", *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*, Melbourne, Australia, 2023, pp. 1700-1711. DOI: https://doi.org/10.1109/ICSE48619.2023.00146
12. Koitz-Hristov, R., Sterner, T., Stracke, L., Wotawa, F. (2024), "On the suitability of checked coverage and genetic parameter tuning in test suite reduction", *Journal of Software: Evolution and Process,* Vol. 36(8), e2656. DOI: https://doi.org/10.1002/smr.2656
13. Dang, Z., Wang, H. (2024), "Leveraging meta-heuristic algorithms for effective software fault prediction: A comprehensive study", *Journal of Engineering and Applied Science,* Vol. 71, Art. No. 189. DOI: https://doi.org/10.1186/s44147-024-00529-0
14. Rheaf, T. S. (2025), "Prioritization of Modules to Reduce Software Testing Time and Costs Using Evolutionary Algorithms and the KLOC Method", *Journal of Education for Pure Science,* Vol. 15(1). P. 116-131. DOI: https://doi.org/10.32792/jeps.v15i1.658
15. Bai, R., Chen, R., Lei, X., Wu, K. (2024), "A Test Report Optimization Method Fusing Reinforcement Learning and Genetic Algorithms", *Electronics*, Vol. 13(21), Art. No. 4281. DOI: https://doi.org/10.3390/electronics13214281
16. Świechowski, M., Godlewski, K., Sawicki, B., Mańdziuk, J. (2023), "Monte Carlo Tree Search: A review of recent modifications and applications", *Artificial Intelligence Review,* Vol. 56, P. 2497–2562. DOI: https://doi.org/10.1007/s10462-022-10228-y
17. Ye, A., Wang, L., Zhao, L., Ke, J. (2022), "Ex²: Monte Carlo Tree Search-based test inputs prioritization for fuzzing deep neural networks", *International Journal of Intelligent Systems,* Vol. 37(12), P. 11966–11984. DOI: https://doi.org/10.1002/int.23072
18. Kocsis, L.; Szepesvári, C. (2006), "Bandit Based Monte-Carlo Planning", *Lecture Notes in Computer Science (ECML 2006)*, Vol. 4212, P. 282-293. DOI: https://doi.org/10.1007/11871842_29
19. Lin, C.-T., Tang, K.-W., Wang, J.-S., Kapfhammer, G. M. (2017), "Empirically evaluating Greedy-based test suite reduction methods at different levels of test suite complexity", *Science of Computer Programming*, Vol. 150, P. 1-25. DOI: https://doi.org/10.1016/j.scico.2017.05.004
20. Parsa, S.; Khalilian, A. (2009), "A Bi-objective Model Inspired Greedy Algorithm for Test Suite Minimization", *Lecture Notes in Computer Science (FGIT 2009)* Vol. 5899, P. 208–215. DOI: https://doi.org/10.1007/978-3-642-10509-8_24
21. Jehan, S., Wotawa, F. (2023), "An Empirical Study of Greedy Test Suite Minimization Techniques Using Mutation Coverage", *IEEE Access*, Vol. 11, P. 65427–65442. DOI: https://doi.org/10.1109/ACCESS.2023.3289073

22. Putra, A. W., Legowo, N. (2025), "Greedy Algorithm Implementation for Test Case Prioritization in the Regression Testing Phase", *Journal of Computer Science*, Vol. 21(2), P. 290–303. DOI: https://doi.org/10.3844/jcssp.2025.290.303
23. Zeller, A., Hildebrandt, R. (2002), "Simplifying and isolating failure-inducing input", *IEEE Transactions on Software Engineering*, Vol. 28(2), P. 183–200. DOI: https://doi.org/10.1109/32.988498
24. Cleve, H., Zeller, A. (2005), "Locating causes of program failures", *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, Association for Computing Machinery, New York, USA, 2005, P. 342–351. DOI: https://doi.org/10.1145/1062455.1062522
25. Misherghi, G., Su, Z. (2006), "HDD: Hierarchical delta debugging", *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, Association for Computing Machinery, New York, USA, 2006, P. 142–151. DOI: https://doi.org/10.1145/1134285.1134307
26. Wang, G., Wu, Y., Zhu, Q., Xiong, Y., Zhang, X., Zhang, L. (2023), "A Probabilistic Delta Debugging Approach for Abstract Syntax Trees", *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, Florence, Italy, 2023, pp. 763–773. DOI: https://doi.org/10.1109/ISSRE59848.2023.00060
27. Puterman, M. L. (1994), "Markov Decision Processes: Discrete Stochastic Dynamic Programming", John Wiley & Sons. DOI: https://doi.org/10.1002/9780470316887
28. Singh, S. P. (1992), "Transfer of learning by composing solutions of elemental sequential tasks", *Machine Learning*, Vol. 8, P. 323–339. DOI: https://doi.org/10.1007/BF00992700
29. Sutton, R. S., Barto, A. G. (2018), "Reinforcement Learning: An Introduction", (2nd ed.). MIT Press.
30. Li, L., Walsh, T. J., Littman, M. L. (2006), "Towards a Unified Theory of State Abstraction for *MDPs*", *Proceedings of the 9th International Symposium on Artificial Intelligence and Mathematics, 2006*.
31. Watkins, C.J.C.H., Dayan, P. (1992), "Q-learning", *Machine Learning*, Vol. 8, P. 279–292. DOI: https://doi.org/10.1007/BF00992698
32. Hasselt, H. (2010), "Double Q-learning", *Advances in Neural Information Processing Systems (NIPS 2010)*, Vol. 23.
33. Groce, A., Alipour, M. A., Zhang, C., Chen, Y., Regehr, J. (2016), "Cause reduction: Delta debugging, even without bugs", *Software Testing, Verification and Reliability*, Vol. 26, P. 40–68. DOI: https://doi.org/10.1002/stvr.1574
34. Auer, P., Cesa-Bianchi, N., Fischer, P. (2002), "Finite-time Analysis of the Multiarmed Bandit Problem", *Machine Learning*, Vol. 47, P. 235–256. DOI: https://doi.org/10.1023/A:1013689704352

Received (Надійшла) 13.11.2025
Accepted for publication (Прийнята до друку) 08.12.2025
Publication date (Дата публікації) 28.12.2025

About the Authors / Відомості про авторів

Hulevych Mykhailo – National Technical University "Kharkiv Polytechnic Institute", PhD Student, Computer Engineering and Programming Department, Kharkiv, Ukraine;
e-mail: gulevich30misha@gmail.com; ORCID ID: https://orcid.org/0009-0003-8622-3271

Гулевич Михайло Володимирович – Національний технічний університет "Харківський політехнічний інститут", аспірант кафедри комп'ютерної інженерії та програмування, Харків, Україна.

ОЦІНЮВАННЯ ЕФЕКТИВНОСТІ МЕТОДУ ФОРМУВАННЯ ТЕСТОВИХ СЦЕНАРІЇВ ДЛЯ C++ БІБЛІОТЕК НА ОСНОВІ АГЕНТА З Q-НАВЧАННЯМ

Оптимізація тестових сценаріїв (ТС) є необхідною умовою підвищення ефективності регресійного тестування C++ бібліотек. **Предметом дослідження** є методи формування (оптимізації) ТС для C++ бібліотек. **Мета роботи** – оцінити ефективність методу формування ТС для C++ бібліотек на основі агента з Q-навчанням. **Завдання дослідження:** удосконалити математичну модель агента з Q-навчанням для підвищення ефективності формування ТС для C++ бібліотек в умовах високої розрідженості простору станів агента з Q-навчанням; дослідити вплив параметрів удосконаленої моделі агента з Q-навчанням на його поведінку за такими умовами; розглянути можливість мінімізації сформованих ТС методом їх формування завдяки алгоритму дельта-дебаггінг мінімізації ТС; оцінити ефективність запропонованого методу й порівняти з відомими методами оптимізації ТС. **Методи дослідження.** У роботі застосовано метод пошуку на дереві Монте-Карло, класичну математичну модель Q-навчання, алгоритм дельта-дебаггінг мінімізації ТС і жадібний алгоритм оптимізації ТС. Ефективність запропонованого методу оцінено на двох C++ бібліотеках з відкритим вихідним кодом за допомогою статистичного аналізу 100 математичних моделювань конфігурацій методів, які досліджуються. **Досягнуті результати:** оцінка ефективності вказує на те, що запропонований метод забезпечує такі середні значення показників ефективності оптимізації ТС для C++ бібліотек: коефіцієнт збереження покриття становить до 1.225, коефіцієнт стиснення тестового набору (ТН) – до 0.86 й коефіцієнт скорочення часу на виконання ТН – до 0.74. Установлено, якщо порівнювати з жадібним алгоритмом оптимізації ТС, дельта-дебаггінг алгоритмом мінімізації ТС та методом оптимізації ТС на основі пошуку на дереві Монте-Карло, то запропонований метод має суттєве підвищення ефективності формування (оптимізації) ТС для C++ бібліотек. **Висновки.** Удосконалена математична модель забезпечує узагальнення досвіду агента з Q-навчанням між подібними ТС і підвищує ефективність їх формування в умовах високої розрідженості простору станів агента з Q-навчанням. Отже, за результатами оцінювання методу формування ТС для C++ бібліотек на основі агента з Q-навчанням підтверджено його доцільність у розв’язанні задачі формування (оптимізації) ТС для C++ бібліотек, що дає змогу скоротити довжину ТС без втрати гілкового покриття коду C++ бібліотеки, яка тестується, і зменшити час на виконання ТН. Подальші дослідження будуть присвячені формуванню ТС для C++ бібліотек на основі агента глибинного навчання з підкріпленням.

Ключові слова: оптимізація тестових сценаріїв; тестування програмного забезпечення; Q-навчання; Q-таблиця; дельта-дебаггінг, покриття коду; мінімізація; C++; агент; навчання з підкріпленням.

Bibliographic descriptions / Бібліографічні описи

Hulevych, M. (2025), "Evaluation of the effectiveness of the test scenarios forming method for C++ libraries based on a Q-learning agent", *Management Information Systems and Devises*, No. 4 (187), P. 20–46. DOI: https://doi.org/10.30837/0135-1710.2025.187.020

Гулевич М. В. Оцінювання ефективності методу формування тестових сценаріїв для C++ бібліотек на основі агента з Q-навчанням. *Автоматизовані системи управління та прилади автоматики*. 2025. № 4 (187). С. 20–46. DOI: https://doi.org/10.30837/0135-1710.2025.187.020