applied sciences

MDPI

Article

An Intelligent Method for C++ Test Case Synthesis Based on a Q-Learning Agent

Serhii Semenov [1,*], Oleksii Kolomiitsev [2], Mykhailo Hulevych [2,*], Patryk Mazurek [1] and Olena Chernyk [2]

[1] Institute of Security and Informatics, University of the National Education Commission, 30-084 Krakow, Poland; patryk.mazurek@uken.krakow.pl
[2] Department of "Computer Engineering and Programming", National Technical University "Kharkiv Polytechnic Institute", 61000 Kharkiv, Ukraine; oleksii.kolomiitsev@khpi.edu.ua (O.K.); lenachernikh@gmail.com (O.C.)
* Correspondence: serhii.semenov@uken.krakow.pl (S.S.); mykhailo.hulevych@cs.khpi.edu.ua (M.H.)

Abstract

Ensuring software quality during development requires effective regression testing. However, test suites in open-source libraries often grow large, redundant, and difficult to maintain. Most traditional test suite optimization methods treat test cases as atomic units, without analyzing the utility of individual instructions. This paper presents an intelligent method for test case synthesis using a Q-learning agent. The agent learns to construct compact test cases by interacting with an execution environment and receives rewards based on branch coverage improvements and simultaneous reductions in test case length. The training process includes a pretraining phase that transfers knowledge from the original test suite, followed by adaptive learning episodes on individual test cases. As a result, the method requires no formal documentation or API specifications and uses only execution traces of the original test cases. An explicit synthesis algorithm constructs new test cases by selecting API calls from a learned policy encoded in a Q-table. Experiments were conducted on two open-source C++ libraries of differing API complexity and original test suite size. The results show that the proposed method can reach up to 67% test suite reduction while preserving branch coverage, confirming its effectiveness for regression test suite minimization in resource-constrained or specification-limited environments.

Keywords: C++; test suite minimization; test case synthesis; reinforcement learning; Q-learning; Q-table; dynamic environment; code coverage

Academic Editors: Hamido Fujita, Héctor Manuel Pérez-Meana and Andres Hernandez-Matamoros

Received: 29 June 2025
Revised: 23 July 2025
Accepted: 31 July 2025
Published: 2 August 2025

Citation: Semenov, S.; Kolomiitsev, O.; Hulevych, M.; Mazurek, P.; Chernyk, O. An Intelligent Method for C++ Test Case Synthesis Based on a Q-Learning Agent. *Appl. Sci.* **2025**, *15*, 8596. https://doi.org/10.3390/app15158596

1. Introduction

1.1. Motivation

During software development, particularly within the C++ ecosystem, ensuring software quality is a critical task that heavily relies on effective testing. This is especially relevant for libraries that evolve intensively and are applied across diverse domains, from embedded systems to high-performance computing. As the size of the codebase and the complexity of such libraries' APIs grow, the associated test suites also expand significantly. In the context of continuous integration (CI) systems, the need to execute bulky and redundant test cases (TCs) after each codebase modification leads to the inefficient use of resources and hampers the timely detection of defects. An additional challenge arises from the fact that open-source libraries frequently lack well-documented test requirements and specifications for valid API usage. However, classical test suite optimization methods, such

as greedy minimization, static test slicing, or delta debugging, typically operate under restrictive assumptions. These include the atomicity of TCs, the availability of failure traces, or the presence of specifications which are rarely available in black-box testing or libraries with undocumented APIs. Moreover, as test suites scale with the growth of codebases and the combinatorial explosion of API usage patterns, such methods struggle to remain efficient, both in terms of computational cost and reduction quality. Consequently, there is a clear need for an adaptive, specification-agnostic method capable of fine-grained optimization at the API level, without compromising coverage or increasing computational burden.

1.2. State of the Art

1.2.1. Overview

Approaches to solving the test suite optimization (TSO) problem encompass a broad range of methods aimed at reducing the size of a test suite (TS) without compromising testing effectiveness. Among the main TSO directions, two key optimization subproblems are typically distinguished: test suite minimization (TSM) and test suite prioritization (TSP). In the TSP context, TCs within a suite are ordered according to criteria that maximize expected utility, such as code coverage or fault detection probability. In contrast, the goal of TSM is to eliminate redundant TCs that do not affect predefined characteristics of testing effectiveness. In addition, TSO techniques can be broadly categorized by the granularity at which they operate. Coarse-grained techniques treat TC as atomic units, performing selection, prioritization, or reduction on whole TCs without analyzing their internal structure. In contrast, fine-grained techniques operate at a lower abstraction level, such as individual instructions, statements, or API calls within test cases. These approaches enable more precise optimization by evaluating the utility of specific actions or call sequences, rather than entire TCs. This distinction is especially important when test cases are long or composed of semantically diverse segments. Furthermore, optimization techniques are often assessed in terms of adequacy [1], which refers to their ability to preserve the desired characteristics of testing effectiveness (e.g., code coverage, fault detection potential, behavioral diversity). An adequate technique ensures that the reduced suite remains effective in its original purpose, while inadequate methods may inadvertently degrade coverage or lose critical behaviors.

A systematic review [2] analyzed 29 studies published between 2006 and 2020 concerning the use of machine learning for both TSP and TSM. The review indicates that although most studies rely on supervised learning, there is a growing interest in the application of reinforcement learning. However, the authors highlight the lack of a unified evaluation methodology in this domain, which complicates the comparison of different techniques. Furthermore, a broader review [3] examined more than 43 methods employing machine learning for TSO, including classification, clustering, reinforcement learning, and hybrid approaches. A detailed survey of minimization, selection, and prioritization strategies for TCs is presented in [4], which systematizes regression testing strategies and classifies them according to their impact on coverage, execution cost, and defect detection ability. This survey emphasizes the importance of multi-criteria analysis in test suite optimization and outlines the strengths and weaknesses of various methods such as selection-based testing, prioritization, and minimization. A comprehensive systematic review [5] provides a broad taxonomy of test suite reduction (TSR) techniques, highlighting methodological diversity across greedy, clustering, slicing, and search-based approaches, and emphasizing quality evaluation criteria and guidelines for reproducibility. This work serves as a foundational reference for understanding the landscape of TSR methods and benchmarking new techniques. As noted in [6], source-level fine-grained test suite reduction techniques often achieve high precision by analyzing code at the level of statements or control dependencies. However,

such methods tend to suffer from limited scalability when applied to large codebases, due to the high cost of program analysis and dependency tracking. This creates a fundamental trade-off between reduction precision and computational efficiency, especially in contexts lacking modular decomposition or parallel analysis support. A recent structured literature review by [7] provides a comprehensive overview of the intersection between software testing and reinforcement learning (RL). The authors categorize existing approaches based on testing objectives, RL formulations, and application domains, highlighting emerging trends such as policy learning for test prioritization, environment modeling for input generation, and reward engineering for behavioral coverage. Notably, the review points out that most RL-based testing methods focus on fuzzing, prioritization, or simple decision strategies, while fine-grained TSM as test synthesis especially for compiled languages like C++ remains underexplored. This underscores the need for research into RL-driven test construction strategies that do not rely on formal specifications.

1.2.2. Classic and Fine-Grained TSM Approaches

Greedy Reduction. One of the most well-known approaches to reducing test suites is greedy minimization, which iteratively selects those TCs that maximize coverage improvement. Greedy reduction strategies remain among the most used approaches for test suite minimization due to their simplicity and practical efficiency. Recent empirical studies confirm their effectiveness in preserving fault detection capability while reducing suite size. In [8], the authors evaluated several greedy TSM algorithms using mutation coverage as the adequacy criterion and demonstrated that these techniques can reduce the size of the original test suites to approximately 70% on average without compromising their ability to detect faults. This result underscores the utility of greedy methods in contexts where scalability and preservation of fault-revealing behavior are essential, though the authors also note variability in effectiveness across different program structures. Despite their practicality, greedy test suite minimization methods suffer from several well-known limitations. The outcome of greedy minimization is highly sensitive to the order of test cases. Minor changes in input ordering can lead to significantly different minimized sets, undermining reproducibility. Also, these methods do not account for inter-test dependencies, which can result in redundant tests being retained or essential behavior being lost if dependencies are broken. As noted in [8], greedy techniques also show variable performance across different program structures, performing worse when fault detection requires intricate interactions between test inputs or when fault-revealing tests are not highly ranked by simple coverage heuristics. Lastly, greedy methods are limited to coarse-grained selection, typically treating entire test cases as atomic units, which reduces precision in eliminating internal redundancy.

Delta Debugging. This method was introduced in [9] and it achieves an automated minimization of failure-inducing TCs by iteratively removing fragments and checking whether the failure persists. While useful for fault localization, the method treats TCs as atomic units and does not account for the semantics of API call sequences.

Test Slicing. The HSFal method [10] implements hybrid spectrum-based slicing for fault localization by preserving only those TC components that causally contribute to a failure. Despite its effectiveness for fault localization, the method does not support the synthesis of new API call sequences and is not applicable in contexts without source code access. While traditional test suite optimization methods typically operate at the level of entire test cases, recent work has explored fine-grained reduction techniques that analyze the contribution of individual instructions or execution slices. For example, Ref. [11] proposed a dynamic slicing-based method for fine-grained test minimization, which removes non-essential parts of test cases while preserving their ability to reveal faults. Their evaluation

on five real-world Java projects showed that up to 52% of partially redundant test cases could be eliminated without reducing fault detection capability. This demonstrates the potential of instruction-level optimization for improving test suite maintainability without compromising effectiveness.

Search-Based Minimization. An important direction within classical test suite minimization involves search-based techniques, where the problem is framed as an optimization task. Among them, genetic algorithms (GAs) are widely applied due to their ability to explore large solution spaces efficiently. A notable early contribution [12] introduced a GA-based approach for reducing test suites while preserving coverage, formulating the problem using a fitness function that balances coverage preservation with minimization. The evaluation demonstrated that GAs could significantly reduce test suite size with minimal loss in fault detection capability. This method laid the ground for subsequent evolutionary and metaheuristic approaches in test optimization. However, search-based techniques often require careful parameter tuning and may incur higher computational costs compared to greedy heuristics.

ILP-based Minimization. Integer Linear Programming (ILP) has also been actively explored in the context of test suite minimization, particularly when multiple criteria such as coverage, cost, and fault detection must be jointly optimized. In [13], the authors introduced multi-objective ILP formulations for the test suite minimization problem, demonstrating how Pareto-optimal solutions can be obtained with formal guarantees. Their approach models the TSM task as a constrained optimization problem, solving it using exact methods instead of heuristics. Building upon this line, research [14] proposed an improved ILP encoding for multi-criteria TSM, offering more scalable and compact formulations. These methods, while precise, may suffer from increased computational cost, especially when applied to large-scale systems with complex constraints. Nonetheless, ILP provides a solid theoretical foundation for minimizing test suites under multiple conflicting objectives.

Clustering. These techniques enable the identification of functionally duplicate TCs, which can be replaced with a smaller number of representative ones. In [15], the authors applied a Doc2Vec model to convert textual requirement descriptions into numerical vectors, followed by using the HDBSCAN clustering algorithm to group semantically similar TCs. This approach made it possible to detect functionally similar TCs and reduce the total number in the test suite without sacrificing coverage. Such methods are effective in projects that contain many functionally similar TCs.

1.2.3. Reinforcement Learning TSO

In contrast to classification and clustering approaches, RL involves an agent that learns through interaction with the environment. RL-based methods are gaining popularity in the context of complex or dynamic testing environments. Recent studies highlight the promise of RL techniques for addressing TSO problems. In [16], a Q-learning agent that learns from user interaction logs to determine the execution order of tests was proposed. In [17], Q-learning was applied for test TC prioritization in CI environments. The agent learns to select the execution order of TCs to maximize effectiveness metrics such as the Average Percentage of Faults Detected (APFD). Additionally, Q-learning agents have already been used for automated testing of mobile and web applications [18,19]. In [20], the authors proposed a method based on an XCSF agent with experience replay, enabling operation in high-dimensional state spaces. Among the studies conceptually closest to ours is RLFUZZ [18], in which a Q-learning agent, based on the Deep Deterministic Policy Gradient (DDPG) algorithm, optimizes input parameter fuzzing by rewarding the discovery of new execution traces. However, RLFUZZ operates at the byte mutation level and does not address the problem of structured TCs synthesis composed of API calls. In addition, [21] proposed

a reinforcement learning-based Detected Goal Catalyst (DGC) method for minimizing crashing paths in software debugging, demonstrating that RL can be effectively used for fault localization and reduction of failure-inducing sequences.

1.2.4. Deep Learning TSO

In addition to reinforcement learning, numerous studies explore the application of deep learning (DL) in the context of CI. Typically, these models are trained to predict TC fault-proneness or effectiveness based on large volumes of historical execution data. Deep neural networks (DNNs) have been used [22] to predict TCs capable of detecting faults, with the aim of incorporating them into new TSs to improve testing efficiency. In [23], a DNN-based model is enriched with meta-information such as execution duration and status to optimize resource allocation in constrained environments. Furthermore, papers [24,25] employ long short-term memory (LSTM) recurrent neural networks to model the temporal dynamics of TC execution. This approach enables the capture of sequential dependencies between atomic TCs and the identification of behavioral patterns over time. However, DL-based methods typically require large volumes of training data, which are often generated in advance. This limits their applicability in contexts where formal specifications required for synthetic data generation are unavailable.

1.2.5. Hybrid Architectures and Metaheuristics

Hybrid methods combine the advantages of multiple approaches, typically integrating heuristic algorithms with machine learning (ML) models to achieve higher performance. For example, in [26], an Adaptive Neuro-Fuzzy Inference System (ANFIS) is combined with genetic algorithms for multi-criteria optimization of regression test suites in the context of CI. In [27], the SBLA-AdaBoost CNN method was introduced, combining Side-blotched Lizard Algorithm-based optimization with an ensemble deep neural network to address TC minimization and prioritization in regression testing. Although this method demonstrates high accuracy in classifying faulty TC, it does not support the synthesis of new TC. The Law of Minimum (LoM) has recently been applied to test case prioritization to emphasize the influence of limiting factors such as method-level change impact on early fault detection. The authors [28] proposed a prioritization strategy based on LoM, which ranks test cases by the most constrained method coverage. However, they observed that such prioritization can lead to repeated execution of structurally similar test cases, potentially delaying the exposure of faults in under-tested regions. To address this, they introduced a dissimilarity-based enhancement using Jaccard similarity to penalize redundancy.

1.2.6. Comparison and Research Gap

Table 1 summarizes a comparative analysis of representative TSO methods across several evaluation dimensions. The precision refers to the selectivity with which a method eliminates redundant test fragments without degrading effectiveness. Computational cost characterizes the expected resource consumption during optimization, including instrumentation overhead and runtime complexity. Scalability denotes the ability of the method to handle large and complex test suites. Finally, input requirement indicates the necessary information or artifacts required for the method to operate effectively.

Empirical results illustrate the practical trade-offs between these dimensions. For example, greedy reduction methods achieve up to a 70% test suite size reduction on average without compromising fault detection capability, as shown in [8], yet remain sensitive to TC ordering and coverage redundancy. ILP-based approaches provide formally optimal solutions across multiple criteria (e.g., coverage, fault detection, execution cost), but suffer from scalability constraints when applied to large, complex test suites [13,14].

Table 1. Summary of the capabilities of classical, machine learning, and hybrid methods for TSO across dimensions such as precision, computational cost, scalability, and required input artifacts.

Method/Approach	Type	Precision	Comput. Cost	Scalability	Input Requirement
Greedy Reduction [8]	Heuristic	Medium	Low	High	Source
Delta Debugging [9]	Diff Analysis + Fault localization	High	Medium	Low	Source + Outcome
Test Slicing HSFal [10]	Static Slicing + Fault localization	Medium	High	Low	Source + Spectrum
Dynamic Slicing [11]	Dynamic Slicing	High	High	Medium	Execution Trace
Genetic Algorithms [12]	Search-Based	Medium	Medium-High	Medium	Coverage
ILP-Based TSM [13,14]	Optimization (ILP/MOIP)	High	High	Medium	Coverage + Dependency Graph
Doc2Vec + HDBSCAN [15]	Clustering	Medium	Medium	Medium	Requirements
RL Prioritization [16]	RL-Based Prioritization	Medium	Medium	Medium	User Logs
RLFUZZ [18]	Fuzzing + DDPG	Low	High	High	Execution Trace
XCSF + Replay Buffer [20]	RL-based Classifier	Medium	Medium-High	Medium	Logs + Coverage
RL + DGC [21]	RL for TSP Optimization	Medium	Medium	Medium	Coverage + History
LSTM [24,25]	Deep Learning	Medium	High	Low	Historical Logs
ANFIS [26]	Neural Network	Medium	Medium	Low	Meta Data
SBLA-AdaBoost-CNN [27]	DL + Boosting	High	High	Medium	Execution History
LoM-Score [28]	Heuristic + Slicing	Medium	Medium	Medium	Change Impact + Slice
Our Method	RL-based + Fine-Grained TSM	High	Medium	Medium	Coverage + API Trace

Delta debugging is highly effective for isolating failure-inducing input fragments and has been shown to reduce test cases to minimal failure-inducing subsets, but it operates at a coarse level and lacks support for semantic preservation in API-based scenarios [9]. Slicing-based methods such as HSFal are precise and maintain only semantically relevant code segments yet are dependent on source-level analysis and are infeasible when source code is unavailable [10]. Dynamic slicing-based methods, such as the one proposed in [11], support fine-grained test suite minimization by identifying and removing non-essential instructions or operations. Their evaluation on five real-world Java projects demonstrated that up to 52% of partially redundant test cases could be eliminated without reducing fault detection capability, confirming the potential of instruction-level optimization for improving test suite maintainability. Reinforcement learning, particularly Q-learning, has shown strong results in test prioritization [16,17], fuzzing [19], and debugging [21]. Other RL-based classifiers, such as the XCSF approach combined with a replay buffer [20], show promise in TSP by using experience replay to improve policy stability and generalization. While methods like RLFUZZ [16,18] demonstrate success in specific domains, most operate at the level of whole test cases or byte streams, lacking support for API-level structural reasoning. Deep learning-based methods demonstrate high fault-detection precision, particularly when modeling test behavior with temporal dependencies [24,25,27], but are constrained by data availability and scalability, especially for sequence-based models like LSTM. Reinforcement learning, particularly its deep variants, has shown potential in test prioritization and fuzzing, but fine-grained TSM as test case synthesis remains largely unexplored.

This analysis highlights that few existing techniques simultaneously achieve high precision, reasonable computational cost, and applicability in low-information environments. This defines the research gap our work addresses: the need for a reinforcement learning-based method capable of synthesizing compact and effective test cases without relying on formal specifications, and capable of scaling to open-source C++ libraries with diverse API structures and incomplete documentation.

1.3. Objectives and Contributions

The aim of this research is to develop an intelligent method for synthesizing TCs for C++ libraries based on a Q-learning agent, which achieves the reduction of the original test suite size without compromising the predefined characteristics of testing effectiveness. The method operates in the absence of formal specifications for valid API usage. In this research, branch code coverage, in particular, is used as the primary metric for evaluating testing effectiveness. However, the method is not limited to a specific metric and can be adapted to alternative criteria, such as line coverage or fault detection capabilities.

To achieve this aim, the following objectives have been defined:

1. To formalize the model of a Q-learning agent for the task of test case synthesis, considering the specific structure of states, actions, and reward function, and to implement a training algorithm based on classical Q-learning.
2. To develop a test case synthesis algorithm that rebuilds the original TCs at the API level using the memory of a trained agent, aiming to reduce TC length while preserving target branch coverage.
3. To design an intelligent method for test cases synthesis for C++ libraries that integrates the Q-learning agent with the test case synthesis algorithm.
4. To conduct an experimental evaluation of the method's effectiveness by applying it to two open-source C++ libraries, including an analysis of the agent's learning dynamics, the characteristics of the synthesized test cases, comparison with uniform random distribution-based TC synthesis as baseline, and assessment of the method's scalability.

This work contributes an intelligent test case synthesis method for C++ libraries, capable of minimizing test suite size while preserving branch coverage and operating without specifications, validated through experiments on open-source C++ codebases. The remainder of the paper is organized as follows. Section 2 presents the formalization of the Q-learning agent model and reinforcement learning components implementation. In Section 2.10. the test case synthesis algorithm is presented. In Section 3, we provide an experimental evaluation, comparing the proposed approach with uniform random distribution-based TC synthesis as baseline across two open-source libraries. Section 4 discusses the results, including stability, effectiveness, limitations, and threats to validity of the method, and outlines directions for future work. Section 5 concludes the paper.

2. Materials and Methods

2.1. Q-Learning Agent Model

We formalize a model in which the agent incrementally constructs a sequence of actions, guided by feedback from the environment in the form of rewards. The process of constructing an action sequence can be viewed as a Markov Decision Process (MDP) [29], which is defined by a 5-tuple:

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

- $\mathcal{S}$—The set of states representing the current state s;

- $\mathcal{A}$—The set of admissible actions in state s;
- $\mathcal{P}(s'|s,a)$—The state transition probability function, defining the probability of moving to a new state s' after making action a in state s;
- $\mathcal{R}(s,a)$—The reward function, assigning a numerical reward for performing action a in state s;
- $\gamma \in (0,1)$—The discount factor.

In our case, the state space is explored by a Q-learning agent. During the interaction process, at each step $t \in \mathbb{N}$, the agent is in a particular state $s_t \in S$, selects an action $a_t \in \mathcal{A}$, appends the action to the partially synthesized TC, executes it via the environment and receives a reward r_t, with further transition to a new state s_{t+1}.

Since classical Q-learning is a model-free method, the agent does not explicitly use the state transition probability function $\mathcal{P}(s_{t+1}|s_t,a_t)$. Instead, it learns based on the Q-function update algorithm [30]. This update rule follows the standard Temporal Difference (TD) learning approach, where the agent refines value estimates by combining observed and expected rewards over time [31]. The utility function $Q(s,a)$ is approximated as a Q-table that stores the expected reward value for each state–action pair. This enables the agent to improve its test generation behavior through interaction, without relying on specifications or a model. Accordingly, the Q-table values are updated using the classical Bellman equation:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \cdot \left[r_t + \gamma \cdot \max_{a_{t+1}} Q(s_{t+1},a_{t+1}) - Q(s_t,a_t) \right] \quad (1)$$

$\alpha \in (0,\ 1]$ is the learning rate.

2.2. Reinforcement Learning Components

This section provides a systematic overview of the RL components implemented in this study. All core components of the setup are summarized in Table 2. A flowchart of the RL process is shown in Figure 1.

Table 2. Definitions and implementations of RL components used in the test case synthesis.

RL Component	Implementation
Agent and Environment	The Q-learning agent interacts with a virtual execution environment instrumented with code coverage tracing tools. The environment is represented as an execution engine that runs Lua scripts and provides key execution metrics as feedback.
State	Each agent state is represented as a suffix of the currently constructed API calls sequence, limited to the last fixed number of elements. This abstraction can be seen in Section 2.3.
Action Space	The action space $\mathcal{A}$ is constructed from traced function calls in the original test suite. It includes only valid API calls with arguments extracted during the execution. The available action in the space is updated dynamically to prevent invalid combinations during test synthesis. API instrumentation is described in Section 2.11.2.
Policy	The agent follows an exploration policy implemented as ε-greedy strategy. The action is selected based on estimated Q-values with stochasticity controlled by the exploration rate ε (Section 2.6).
Exploration Strategy	The agent starts with a high exploration rate ε to encourage diverse action selection and gradually reduces ε using a decay function over training episodes (Section 2.8).

Table 2. *Cont.*

RL Component	Implementation
Value Function	The agent maintains a Q-table that estimates the expected cumulative reward for each state–action pair (Section 2.3).
Reward	After executing a test case, the agent observes a reward computed as a combination of code coverage increase, redundancy reduction, and length penalty. Our reward function is presented in Section 2.5.
Learning algorithm	The learning process follows the Q-learning paradigm. The agent maintains a Q-table which stores expected values of actions in given states. At the beginning of training, the Q-table is initialized as an empty map. At each step, the agent selects an action using ε-Greedy policy Equation (4), updates the TC, and applies the Bellman equation Equation (1) to update Q-values. The Q-table is updated after every step, ensuring fast feedback propagation. The learning algorithms are presented in Section 2.4.
Learning Phases	The learning consists of two phases: The Pretraining phase. For each TC from original TS, the agent performs imitation learning by replaying the sequence of recorded API calls using Algorithm 1. This allows the initializing of the Q-table with meaningful state–action estimates and capturing common action patterns observed during prior executions. The pretraining procedure is applied sequentially to each test case in the original suite. The Training phase. After pretraining, the agent performs reinforcement learning using Algorithm 2. For each TC from original TS, a fixed number of training episodes is performed, during which the agent interacts with the environment and updates its Q-function. The same Q-table is retained and reused between different test cases, allowing the agent to transfer learned experience across structurally similar sequences.
Invalid Action Handling	When the agent selects an invalid action that leads to an incorrect or semantically incompatible test case, a rollback is performed, and another action is selected. This process is repeated up to a predefined rollback limit (Algorithm 2).
Convergence Criteria	Training continues until a predefined number of episodes is reached or until the loss function, defined as the mean squared Bellman error, Equation (8) stabilizes. The total reward R_{total} is also tracked per episode as an auxiliary signal of learning progress. Details are in Section 2.9.

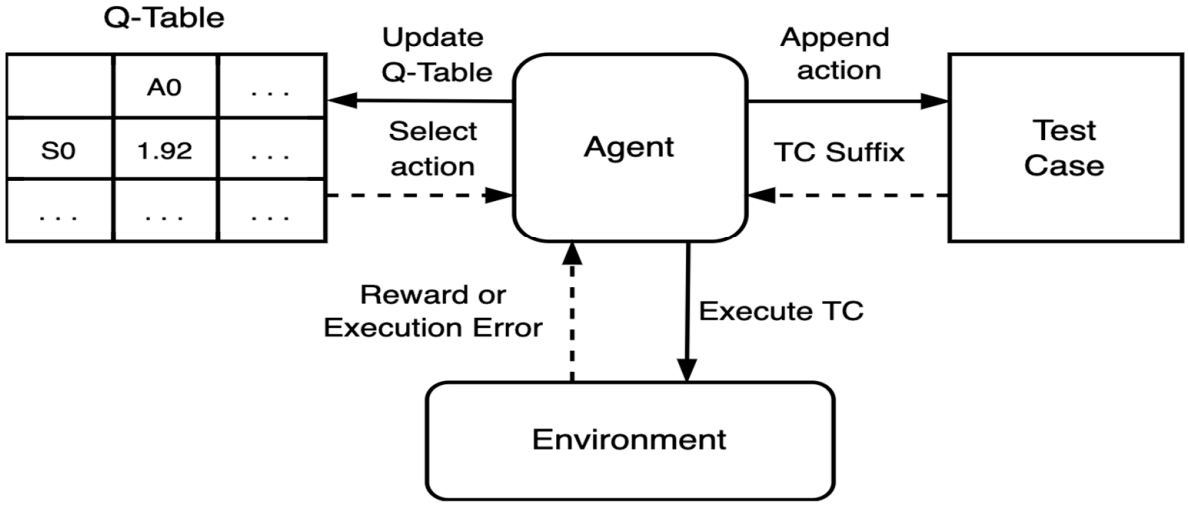

Figure 1. Q-learning-based agent reinforcement learning flowchart.

2.3. Agent's State Representation

In reinforcement learning, the quality of decisions made depends on the representation of the state, which serves as the basis for estimating the utility of actions. However, in tasks involving complex software environments, such as test case synthesis for C++ libraries, the curse of dimensionality becomes a critical concern, as the number of potential states can grow exponentially with the complexity of API usage which a phenomenon often referred to as the state explosion problem [6]. To reduce the complexity, the state s_t is limited as follows.

TC Suffix Function. Let there be a current sequence of API calls of the tested library representing a partially synthesized TC at step t:

$$TC^t = \langle a_1, a_2, \ldots, a_t \rangle, \quad a_i \epsilon \mathcal{A}$$

Let $k \in \mathbb{N}$ be the depth of the call history, then the abstract agent state is determined solely based on the last k actions using the test case suffix function:

$$s_t = Suffix_k(S^t) = \begin{cases} \oslash, & if\ t = 0 \\ \langle a_1, \ldots, a_t \rangle & if\ 0 < t \leq k \\ \langle a_{t-k+1}, \ldots, a_t \rangle & if\ t > k \end{cases} \quad (2)$$

Each call is encoded as a record that abstracts away syntactic details but retains the type of the object, function name, call category, and argument signature. Given a partially synthesized test case in state s_t, the state abstraction is computed using a suffix function of the TC. This limited representation is not fully Markovian in the classical sense. Specifically, it does not contain complete information about argument values, internal object states, or the global execution context that may influence the behavior of a function. Two sequences with the same function names but different arguments will be represented by the same state key. The lack of detailed information on object state changes also leads to the loss of part of the dynamic context. Despite the loss of full Markovian properties, empirical results show that this abstraction is sufficient for effective learning in the task of synthesizing TCs with limited length. Within such local heuristics, the compact representation may enable high learning speed, avoids excessive state-space branching, and prevents excessive memory usage. Also, unlike solutions that require full simulation of the program state or tracking object states, this representation can enable scaling to large APIs without a loss of efficiency. Moreover, such an abstraction makes the implementation stable, controllable, and interpretable, which is especially valuable in CI systems where the transparency of test suite synthesis is crucial. Therefore, this representation offers a rational compromise between accuracy, generalization, and computational efficiency, which is essential for practical tasks of test code generation for C++ libraries.

The Q-table is used as a model for approximating the utility function $Q(s, a)$. Each key in the Q-table is derived using the TC suffix function Equation (2) applied to the partially constructed TC during agent training. The action space $\mathcal{A}$ includes the set of admissible API calls determined based on the currently available objects and functions in the current state. Each action in the action space is associated with a numerical value of the Q-function representing the expected cumulative reward when this action is selected in the given state. Each element in the Q-table functions as a node from which edges emerge toward possible TC extensions through action selection from the $\mathcal{A}$.

2.4. The Agent Training

At the initial stage, pretraining takes place. The pretraining is simulated based on the function call sequences extracted from the original test cases. The agent does not select actions independently but instead replicates the actions that were observed in original TC, receiving rewards for reproducing all steps sequentially. The pseudocode of pretraining Algorithm 1 is shown in below.

Algorithm 1 Agent Pretraining Algorithm

Input:	
TC_0	Original TC
$k > 0$	Call history depth
$\alpha \in (0, 1]$	Learning rate
$\gamma \in (0, 1)$	Discount factor
eps	Number of pretraining episodes
Output:	
Q	Pretrained Q-Table
Body:	
$Q \leftarrow \oslash$	1: Initialize Q-Table
for all *eps* **do**	2: For all pretraining episodes
local $TC \leftarrow \oslash$	3: Initialize local TC
local $\mathcal{A}_{ep} \leftarrow$ GetActionSpace(TC_0)	4: Extract action space from original TC
for each $a_t \in \mathcal{A}_{ep}$ **do**	5: For each instruction/action in local action space
local $s_t \leftarrow Suffix_k(TC)$	6: Compute current state using Equation (2)
$TC \leftarrow TC \shortparallel a_t$	7: Append action to local TC
local $s_{t+1} \leftarrow Suffix_k(TC)$	8: Compute next state using Equation (2)
local $r_t \leftarrow$ Execute(TC)	9: Execute local TC and get reward
$Q \leftarrow Update(Q,\ r_t,\ a_t, s_t,\ s_{t+1},\ \alpha,\ \gamma)$	10: Update the Q-Table using Equation (1)
end for	11: End nested for
end for	12: End for
return Q	13: Return pretrained Q-Table

Next, we present the generalized agent training Algorithm 2. The Q-table is updated during the process of constructing a TC by the agent. At each step, the agent is in a specific state s_t, selects an action a_t, appends it to the TC, and the resulting TC is then executed in a virtual environment. Execution results are evaluated using the reward r_t, and the Q-table is updated according to Equation (1). Unlike classical episodic updates performed after complete execution, in our implementation, the Q-table is updated after each appended action of the TC, enabling faster adaptation of the policy to new action combinations. Rollback is used to handle invalid action sequences during test case construction. The agent may roll back and retry action selection up to a predefined number of steps.

The central component of the agent's learning process is the reward function, which defines which actions are considered beneficial. In the following, we describe its structure and the logic behind assigning positive and negative feedback.

Algorithm 2 Agent Training Algorithm

Input:

Q	Q-Table
TC_0	Original TC
$k > 0$	Call history depth
$\alpha_0, \alpha_{final} \in (0, 1]$	Learning rate
$\varepsilon_0, \varepsilon_{final} \in (0, 1)$	Exploration probability
$\gamma \in (0, 1)$	Discount factor
eps	Number of training episodes
maxRollback	Max failed executions before stop an episode

Output:

Q	Trained Q-Table

Body:

1:	**for** $ep = 1$ **to** *eps* **do**	1: For all training episodes
2:	local $TC \leftarrow \oslash$	2: Initialize local TC
3:	local $\mathcal{A}_{ep} \leftarrow \text{GetActionSpace}(TC_0)$	3: Extract action space from original TC
4:	local $C_{orig} \leftarrow GetCoverage(TC_o)$	4: Initialize original TC coverage
5:	local $C_t \leftarrow \oslash$	5: Initialize current TC coverage
6:	local $\alpha_{ep} \leftarrow \text{LinearDecay}\left(eps,\ ep, \alpha_0, \alpha_{final}\right)$	6: Compute decayed learning rate using Equation (6)
7:	local $\varepsilon_{ep} \leftarrow \text{LinearDecay}\left(eps,\ ep, \varepsilon_0, \varepsilon_{final}\right)$	7: Compute decayed exploration probab. using Equation (6)
8:	local $R_{total} \leftarrow 0$, $\mathcal{L}\text{oss} \leftarrow 0$, $s_t \leftarrow \oslash$, $a_t \leftarrow \oslash$	8: Initialized total reward, loss, current state, and action
9:	**while** $C_t < C_{orig}$ **and not** Over $(\mathcal{A}_{ep})$ **do**	9: While $C_t < C_{orig}$ or action space is not over
10:	local $rollback \leftarrow 0$	10: Initialize rollback counter
11:	**while** $rollback < maxRollback$ **do**	11: While rollback counter less than threshold
12:	$a_t \leftarrow \varepsilon Greedy\left(Q, s_t, \mathcal{A}_{ep}, \varepsilon_{ep}\right)$	12: Selection eGreedy action from action space
13:	$TC \leftarrow TC \# a_t$	13: Append action to TC
14:	**if** $isValid(TC)$ **then**	14: Execute and check if TC runs normally
15:	$\mathcal{A}_{ep} \leftarrow \mathcal{A}_{ep} / a_t$	15: Remove the action from the space
16:	**break**	16: Exit while loop
17:	**else**	17: Else
18:	$TC \leftarrow TC / a_t$	18: Rollback the action from TC
19:	$rollback \leftarrow rollback + 1$	19: Increment the rollback counter
20:	**endif**	20: End If
21:	**end while**	21: End while loop
22:	**if** $rollback > maxRollback$ **then**	22: If maxRollback reached than
23:	**break**	23: Terminate the episode
24:	**endif**	24: End If
25:	local $s_{t+1} \leftarrow Suffix_k(TC)$	25: Compute the next state using Equation (2)
26:	local $r_t = \text{Execute}(TC)$	26: Execute the TC and get reward
27:	$Q \leftarrow Update\left(Q,\ r_t,\ a_t, s_t,\ s_{t+1},\ \alpha_{ep},\ \gamma\right)$	27: Update the Q-Table using Equation (1)
28:	$R_{total} \leftarrow R_{total} + r_t$	28: Accumulate local reward Equation (6)
29:	$\mathcal{L}\text{oss} \leftarrow \mathcal{L}\text{oss} + \mathcal{L}(Q, r_t, a_t, s_t,\ s_{t+1},\ \gamma)$	29: Compute Loss Function Equation (7)
30:	$s_t \leftarrow s_{t+1}$	30: Update current state
31:	$C_t \leftarrow GetCoverage(TC)$	31: Update current coverage
32:	**end while**	32: End while loop
33:	$\text{Log}(\mathcal{L}\text{oss}, R_{total})$	33: Log Loss Function and Total Reward per episode
34:	**end for**	34: End for loop
35:	**return** Q	35: Return Q-Table

2.5. Reward Function

In reinforcement learning-based TC synthesis, the structure of the reward function plays a crucial role. It determines which action sequences the agent will consider beneficial and

which are undesirable. In our method, the reward function is multi-level and includes both an immediate reward after each action and a final reward for the entire completed TC. The first level governs local optimal growth, encouraging the agent to explore new API calls. This design allows the agent to learn after each step, rather than only at the end of the episode, which can be particularly effective when working with complex APIs. Moreover, it is important not only to achieve higher code coverage but also to ensure the structural efficiency of the resulting TC by eliminating redundant calls. We adapt the idea of semantic similarity from AFLNET [19], where function names and argument types are used as a heuristic to detect repetition. In our method, similarity is considered as equality of function names and the number of arguments, without comparing the actual values of the arguments.

Depending on the outcome of executing a partially constructed TC, the stepwise reward is assigned as follows:

- Increase in coverage $\rightarrow$ +0.5;
- New execution traces discovered $\rightarrow$ +0.5;
- Two semantically similar actions in a row $\rightarrow -0.15$;
- Three semantically similar actions in a row $\rightarrow -0.3$;
- No improvement in coverage $\rightarrow -0.1$.

Upon completing the TC, an additional global evaluation is applied based on the results achieved by the entire TC. This evaluation considers the relative increase in coverage or discovery of new traces compared to the original TC.

The TC compression coefficient ϑ is computed as

$$\vartheta = \frac{N - N'}{N} \tag{3}$$

N is the original TC length and N' is the length of the synthesized TC. After the TC is finalized, the agent's final reward per episode is computed using the compression coefficient ϑ as follows:

- Increased coverage $\rightarrow +5.0 \cdot \vartheta$;
- New execution traces discovered $\rightarrow +5.0 \cdot \vartheta$;
- No coverage improvement $\rightarrow +0.0 \cdot \vartheta$;
- Coverage degradation $\rightarrow -5.0 \cdot \vartheta$.

Thus, positive rewards are assigned for increased code coverage, discovery of new execution traces, and synthesizing shorter TCs compared to the original ones. Penalties are imposed for repeating the same actions consecutively, stagnating coverage, or producing semantically equivalent yet inefficient operations. The reward is scaled based on the TC length, encouraging the agent to generate shorter and more efficient TC. Given how the reward is computed, we now proceed to examine the action selection policies, which govern the balance between exploration and exploitation during both training and application phases.

2.6. Action Selection Policy

Once the agent has formed an internal representation of the expected utility of actions in various states via the Q-table, a strategy must be defined for selecting the next action. This strategy, or action selection policy, is a core component of the agent's model. During training, it regulates the trade-off between exploring new paths and exploiting the learned knowledge. During the TC synthesizing phase, the Q-values learned by the agent help avoid actions that do not contribute new traces or coverage, thus promoting the construction of effective TC.

In our implementation, two alternative action selection policies are supported: the ε-Greedy policy and the Boltzmann (Softmax) policy. These are detailed below.

- ε-Greedy policy. With probability ε, the agent selects a_t—random action from the available action set $\mathcal{A}(s_t)$ for current state s_t; with probability $(1-\varepsilon)$, it selects the action with the highest Q-value:

$$a_t = \begin{cases} \underset{a' \in \mathcal{A}(s_t)}{\arg\max} Q(s_t, a') & (1-\varepsilon) \\ \text{rand}\mathcal{A}(s_t) & \varepsilon \end{cases} \quad (4)$$

- Boltzmann policy (Softmax) [31]. In this strategy, actions are selected probabilistically based on their Q-values scaled exponentially, according to the softmax distribution:

$$P(a_t|s_t) = \frac{e^{\frac{Q(s_t, a_{tagent})}{\tau}}}{\sum_{a' \in \mathcal{A}(s_t)} e^{\frac{Q(s_t, a')}{\tau}}} \quad (5)$$

τ is the temperature parameter that controls the influence of stochasticity. A high τ encourages exploration by making all actions nearly equally likely, while as $\tau \to 0$ the behavior approaches that of a greedy policy. In addition to basic training, an important aspect is the reuse of acquired experience. The next subsection describes the limited knowledge transfer between TCs.

2.7. Limited Experience Transfer Strategy During Agent Training

In the context of reinforcement learning, the concept of transfer learning (TL) has been explored in works [32,33], where the possibility of transferring Q-functions or accumulated experience between similar tasks or environments was demonstrated, with the goal of reusing knowledge in substantially different settings. Although our case does not involve generalization across significantly different state spaces, preserving the Q-table between TC based on the same TS may result in an internal knowledge transfer effect functionally close to TL. Specifically, using a ready TC or its fragment as a starting point can help the agent improve coverage more quickly during training.

Thus, for agent training, it may be beneficial to employ a sequential strategy, in which the agent processes the original set of TCs one by one. During training, the agent retains its accumulated Q-table. In this way, the knowledge acquired from one TC is partially transferred to the next. Moreover, before each new training session on a TC, it is useful to allow the agent to make a single pass through the original sequence of API calls. This initial step serves as a form of pretraining and can potentially help the agent orient itself more quickly in the new action space.

2.8. Adaptive Scheduling of Exploration and Learning Rate

At the beginning of training, the agent requires a higher exploration rate and moderate learning updates. As training progresses, it is beneficial to reduce exploration to consolidate the learned policy and to stabilize updates by decreasing the learning rate. This allows the agent to initially stabilize its policy and later adapt to newly discovered trajectories more effectively [34]. In particular, the learning rate α from Equation (1) can be dynamically adjusted during training based on the progress of training episodes. Additionally, exploration strategies can be applied in parallel, where the ε parameter in the ε-Greedy action selection policy gradually decreases over time. This ensures a balance between exploring new actions and exploiting previously acquired knowledge. This dynamic adjustment is performed using the linear decay function. The general formula for linear decay is

$$Decay\left(t,\ T, v_o, v_{final}\right) = v_o - \frac{\left(v_o - v_{final}\right)}{T} \cdot t \quad v_o > v_{final} \quad (6)$$

v_o is the initial parameter value, v_{final} is the lower bound, t is the step, and T is the total number of steps. To quantitatively evaluate the training progress, appropriate performance metrics must be introduced. The next subsection presents the loss function and the total reward, which are used to assess the agent's learning success.

2.9. Training Evaluation Using Total Reward and Q-Value Loss

The effectiveness of the agent's learning process can be evaluated by calculating the total reward per training episode. The total reward refers to the cumulative number of reward units that the agent accumulates during a single episode of training. Formally,

$$R_{total} = \sum_{t=0}^{T} r_t \tag{7}$$

r_t is the reward received at step t, and T is the total number of steps in the episode. This metric reflects the agent's ability to generate efficient sequences that are rewarded positively under the defined reward function. It also helps identify stagnation or regressions in learning dynamics, especially in later epochs. This metric is widely used to evaluate the effectiveness of an agent's policy in reinforcement learning [31]. To monitor the quality of the agent's training, we use a loss function. The loss is defined as the mean squared Bellman error (Mean Squared Loss, MSL):

$$\mathcal{L}(Q) = \frac{1}{2}\left(Q(s_t, a_t) - \left[r_t + \gamma \cdot \max_{a' \in \mathcal{A}(s_{t+1})} Q(s_{t+1}, a')\right]\right)^2 \tag{8}$$

The loss function is computed immediately after the Q-function update Equation (1) and indicates how far the current estimate $Q(s_t, a_t)$ deviates from the expected target which consists of the immediate reward and the estimated maximum future return. This function is standard in TD learning and is widely used in both classical Q-learning and DQN-based approaches [35,36]. The loss function serves as a quantitative indicator of the following:

- How quickly the agent's behavior stabilizes, which is reflected by the reduction of fluctuations in the loss curve.
- Simultaneous reduction in the loss function and increase in the total return R_{total} indicates convergence toward an effective Q-value.

Unlike typical reinforcement learning problems where agents often use deep neural networks, in our case, the Q-table is discrete, and the loss function is used solely as an indicator of stability, not as a target function for optimization. Nevertheless, it is used for tuning training parameters and determining whether training was successful. After training is complete, the agent can be used to synthesize new TC. In the next section, we describe the algorithm for TC synthesis based on the knowledge accumulated in the Q-table.

2.10. Test Case Synthesis Algorithm

The test case synthesis Algorithm 3 uses the memory of a previously trained agent in the form of a Q-table. After the training phase is completed, the Q-table contains the expected utility values for each state–action pair, accumulated over multiple episodes of interaction with the environment. The next step is to synthesize TCs in the nearly exploitation mode using the previously described policies. In this mode, the agent no longer updates the Q-values. The Q-table is used to synthesize sequences of actions. The action selection policy may be ε-greedy Equation (4) or Boltzmann (Softmax, Equation (5)). The synthesis process stops once the achieved coverage exceeds that of the original test case, or when no further actions are available in the current state, or the rollback limit is exceeded.

Algorithm 3 Test Case Synthesis Algorithm

Input:

Q	Trained Q-Table
TC_o	Original TC
$maxRollback$	Maximum number of fails before exit
$\tau > 0$	Boltzmann temperature parameter (Optional)
$\varepsilon \in [0;1]$	eGreedy probability (Optional)

Output:

TC	Out TC

Body:

1:	$TC \leftarrow \oslash$	1: Initialize local TC
2:	$k \leftarrow \text{Extract}(Q)$	2: Extract maximal call history depth from Q-Table
3:	$C_{orig} \leftarrow GetCoverage(TC_o)$	3: Initialize Coverage of the original TC
4:	$\mathcal{A} \leftarrow \text{GetActionSpace}(TC_o)$	4: Extract action space from original TC
5:	$C_t \leftarrow \oslash$	5: Initialize current TC coverage
6:	**while** $C_t < C_{orig}$ **and not** $\text{isOver}(\mathcal{A})$ **do**	6: While action space is not over
7:	local $rollback \leftarrow 0$	7: Initialize rollback counter
8:	local $s_t \leftarrow Suffix_k(TC)$	8: Initialize current state using Equation (2)
9:	local $a_t \leftarrow 0$	9: Initialize current action
10:	**while** $rollback < maxRollback$ **do**	10: While rollback counter less than threshold
11:	**if** ε ! = 0 **then**	11: If policy is eGreedy
12:	$a_t \leftarrow \varepsilon Greedy(Q, s_t, \mathcal{A}, \varepsilon)$	12: Selection eGreedy action from action space
13:	**elseif** $\tau! = 0$ ***then***	13: If policy is Softmax
14:	$a_t \leftarrow Softman(Q, s_t, \mathcal{A}, \tau)$	14: Selection Softmax action from action space
15:	**else**	15: Else
16:	$a_t \leftarrow Rand(\mathcal{A})$	16: Selection random action from action space
17:	**endif**	17: End if
18:	$TC \leftarrow TC \ddagger a_t$	18: Append action to TC
19:	**if** $isValid(TC)$ **then**	19: Execute and check if TC runs normally
20:	$\mathcal{A}_{episode} \leftarrow \mathcal{A}_{episode} / a_t$	20: If TC is broken remove the action from the space
21:	**break**	21: Exit nested while loop
22:	**else**	22: Else
23:	$TC \leftarrow TC / a_t$	23: Rollback the action from TC
24:	$rollback \leftarrow rollback + 1$	24: Increment the rollback counter
25:	**endif**	25: End if
26:	**end while**	26: End while loop
27:	**if** $rollback > maxRollback$ **then**	27: If maxRollback reached, then
28:	**break**	28: Exit while loop
29:	**endif**	29: End if
30:	$C_t \leftarrow GetCoverage(TC)$	30: Compute local coverage of TC
31:	**if** $C_t > C_{orig}$ **then**	31: If local coverage is bigger than coverage of original TC
32:	**break**	32: Exit while loop
33:	**endif**	33: End if
34:	**end while**	34: End while loop
35:	**return** TC	35: Return TC

2.11. Method of Intelligent Test Case Synthesis

2.11.1. Conceptual Pipeline

This study presents an intelligent method for synthesizing TC for C++ libraries, which integrates a Q-learning agent and the test case synthesis Algorithm 3. The method consists of the following conceptual stages:

1. API Instrumentation. At the preparatory stage, which precedes the execution of the pipeline, static instrumentation of the target C++ library is performed as described in Section 2.11.2.
2. Function Call Tracing. During the execution of the original TCs, we log the sequences of API calls, including function signatures, objects, call contexts, and arguments. These traces are then used as the basic action space for training the agent. The tracing mechanism, as well as API instrumentation, is described in [37].
3. Q-learning Agent Training. Based on the collected action space, the agent undergoes multiple training epochs in a simulated environment. At each step, the agent learns to interact with the environment, selects an instruction, and adds it to the current TC, after which it receives feedback on the execution of the TC. The agent model and training Algorithms 1 and 2 are described starting from Section 2.1.
4. TC Synthesis. Upon completion of training, the agent's memory (Q-table) is used to automatically synthesize new TC using Algorithm 3.
5. Final Test Suite Construction. The synthesized TCs with the best characteristics are sequentially aggregated into a new test suite.

The scheme presented in Figure 2 depicts the core components and stages of execution in the form of a pipeline. It should be noted, however, that the API instrumentation step is not explicitly shown in the diagram, although it serves as a critical prerequisite for the operation of the entire system.

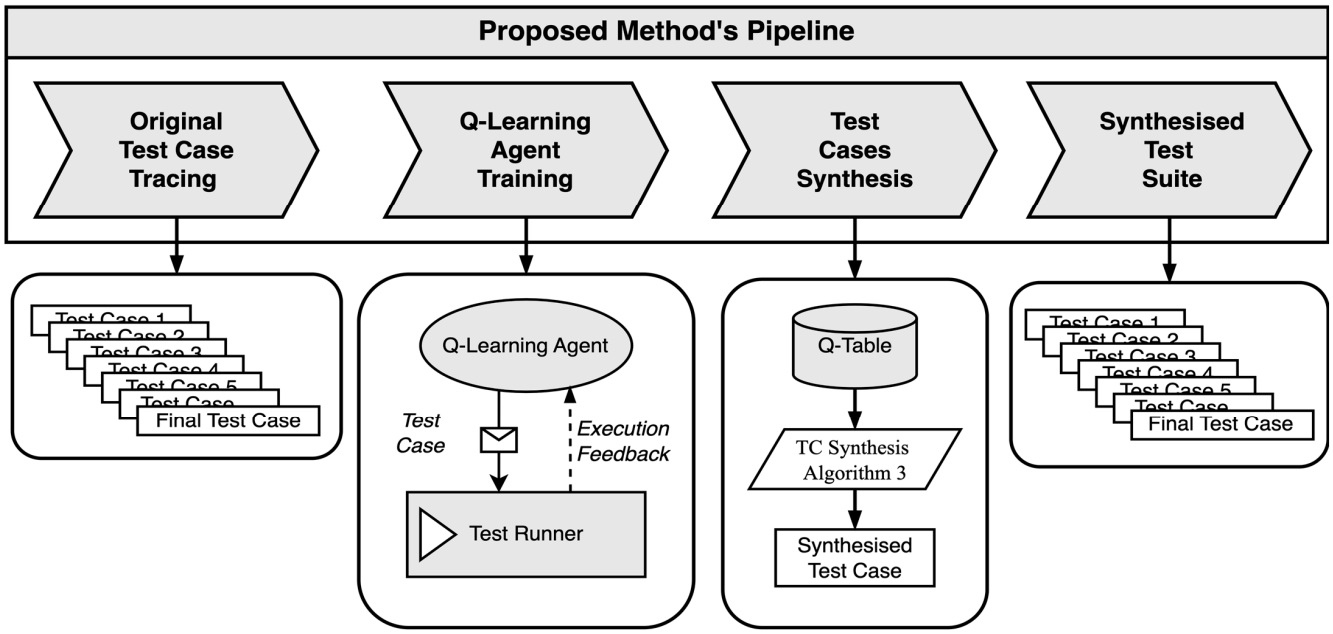

Figure 2. General scheme of the proposed intelligent test case synthesis method, illustrating its pipeline structure as well as the implementation details of the main components.

2.11.2. API Instrumentation

The API of the target C++ library is instrumented using the SWIG tool, which automatically generates wrappers to enable calling C or C++ code from other programming languages such as Python, Java, C#, Lua, Ruby, Tcl, and Perl [38]. In our case, Lua bindings

to the C++ library are generated, allowing its API to be executed in a simulated environment. This execution mode enables validation of test case behavior without recompiling the source code or accessing internal object states. During execution, key behavioral metrics such as branch code coverage, instruction count, and trace novelty are collected. This process is particularly effective for open-source libraries, which often lack formal specifications or comprehensive documentation, since it allows for safe exploration of API usage solely through runtime observation. Also, such approach significantly increases training speed and simplifies integration with the execution environment and library runtime context. In addition, SWIG provides support for binding to other interpreted languages such as Python or JavaScript, offering flexibility in maintaining the synthesized TCs by enabling involvement of developers with diverse technical backgrounds.

3. Experimental Results

3.1. Expiremental Subjects

Two open-source C++ libraries were selected as experimental subjects for evaluating the proposed method as shown in Table 3. The selection was guided by the need to cover a range of API complexity, test suite structure, and realistic usage scenarios where formal specifications are absent.

Table 3. Subject libraries and their characteristics.

Library	Public API Functions Count	Sources (LoC)	Test Suites (LoC)	Test Suites Count	Instructions Count	Original TS Branch Coverage (%)
BitmapPlusPlus	33	760	380	7	71	27
Hjson	122	3936	1970	54	1259	33.9

Hjson is a feature-rich parser for the JSON format, designed for human-readable serialization. It represents a high-complexity case, with approximately 3.9 k lines of code, over 122 public API functions, and widespread use of overloaded methods, recursive data structures, and stateful objects. Its original test suite exhibits structural diversity, including nested object manipulations and complex branching behavior. This makes it a representative subject for testing scenarios involving deep state space and high API variability.

In contrast, BitmapPlusPlus is a lightweight image processing library focused on basic operations such as pixel access and file I/O. It contains around 0.75 k lines of code and exposes 33 public API functions. The existing test suite is shallow, with minimal inter-test dependencies and mostly linear sequences of API calls. This makes BitmapPlusPlus a good representative of simpler libraries with low state coupling and flat API surfaces.

3.2. Experimental Setup

3.2.1. Training Environment

The agent training was conducted on a workstation, which should be considered when interpreting the results. The workstation was equipped with a 2.6 GHz 6-Core Intel Core i7 processor and 16 GB of RAM. Lua-based environment executables were compiled using Clang 15.0 with basic block coverage instrumentation enabled via the compiler flag (fsanitize-coverage = trace-pc-guard). The training process was visualized using the Matplot library [39].

3.2.2. Training Parameters

The agent training procedure consists of two distinct phases: pretraining and main Q-learning-based training. The parameters used during the pretraining stage are summarized in Table 4.

Table 4. Parameters values used in Algorithm 1 for Q-learning agent pretraining.

Parameter	Symbol	Value
Call history depth	k	5
Number of episodes	*eps*	100
Max rollback steps	*maxRollback*	30
Learning rate	α	0.2
Discount factor	γ	0.85

After pretraining, the agent enters the main reinforcement learning stage, detailed in Algorithm 2, where it autonomously explores the action space and refines the Q-values. In each training session for every original TC, the agent performs up to 1000 episodes of guided exploration using the ε-Greedy policy. The exploration probability ε parameter is decayed linearly from 1.0 to 0.2 to ensure a smooth transition from exploration to exploitation. Simultaneously, the learning rate α is decayed from 0.3 to 0.1, allowing the agent to stabilize its updates in later episodes. The full list of hyperparameters and their values for both libraries is presented in Table 5.

Table 5. Parameter values used in Algorithm 2 for Q-learning agent training.

Parameter	Symbol	Value
Call history depth	k	5
Number of episodes	*eps*	1000
Max rollback steps	*maxRollback*	30
Initial Exploration probability	ε_0	1.0
Final Exploration probability	ε_{final}	0.2
Initial Learning rate	α_0	0.3
Final Learning rate	α_{final}	0.1
Discount factor	γ	0.85

3.3. Agent Training Dynamics and Policy Evaluation

Figure 3 illustrates the evolution of the Q-learning agent's performance during training across 1000 episodes for two representative test cases from the BitmapPlusPlus (top row) and Hjson (bottom row) libraries. Each plot corresponds to one of two learning indicators: accumulated reward and loss value.

Figure 3a,c show the episode-wise total reward, computed using Equation (7), which captures the trade-off between instruction efficiency, coverage gains, and redundancy. The reward signal is highly dynamic in early stages due to exploration but begins to stabilize after ~400–500 episodes, indicating the agent is converging to a policy that consistently improves branch coverage while minimizing redundancy. In both libraries, the average episode reward rises over time. Figure 3b,d display the loss function Equation (8), computed during Q-value updates. The loss reflects the discrepancy between estimated Q-values and the temporal-difference targets. A monotonic decrease in average loss is observed in both libraries, demonstrating that the agent's Q-table becomes increasingly accurate. In particular, the Hjson agent (Figure 3d) shows effective minimization of loss by episode 900, confirming that policy learning has almost stabilized. Together, these plots validate the policy convergence and effectiveness of the reward shaping strategy. The decreasing

loss and stabilizing rewards indicate that the agent successfully learns an effective test generation policy under constrained exploration budgets.

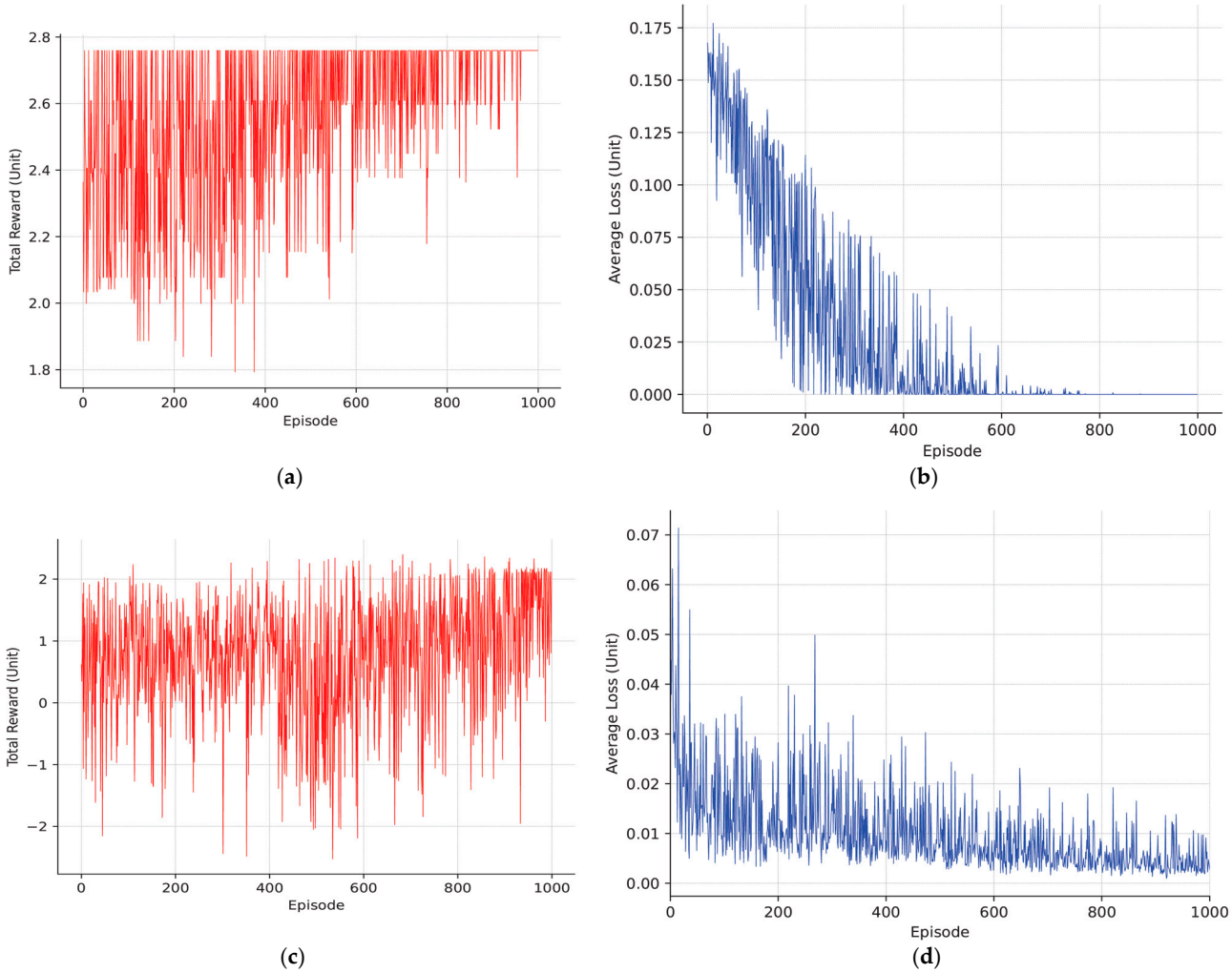

Figure 3. Evolution of the agent's performance over 1000 training episodes for two representative test cases: BitmapPlusPlus and Hjson. Subfigures (**a**,**c**) show the total accumulated reward over episodes for BitmapPlusPlus and Hjson, respectively, reflecting execution quality, coverage gains, and redundancy penalties. Subfigures (**b**,**d**) depict the corresponding average loss per episode, indicating Q-value convergence and policy stability over time.

3.4. Scalability Evaluation Based on Q-Table Resource Consumption

Table 6 presents a comparative analysis of training time and the resulting Q-table size for the tested libraries.

Table 6. Resource consumption metrics for the Q-learning agent across the two target libraries. The table compares the number of training sessions, total number of unique states stored in the Q-table, cumulative training time, and the final memory footprint of the learned Q-table.

Library	Number of Training Sessions (TC)	Number of Unique States	Training Time (Hours)	Final Q-Table Size (Mb)
BitmapPlusPlus	7	~7000	~1	~3.1
Hjson	46	~17,000	~20	~8.5

Figure 4 illustrates the growth dynamics of the Q-table size during training sessions for the evaluated libraries. Each data point on the plot corresponds to the completion of

a training session for a specific TC. In case of BitmapPlusPlus library training sessions (Figure 4a), the Q-table size stabilizes quickly after the fourth session, reaching approximately 7000 entries. In contrast, for the Hjson case (Figure 4b), a more irregular and prolonged growth is observed, eventually reaching around 17,000 entries. After a sharp increase in the Q-table size for a particular TC, the growth dynamics begin to stabilize.

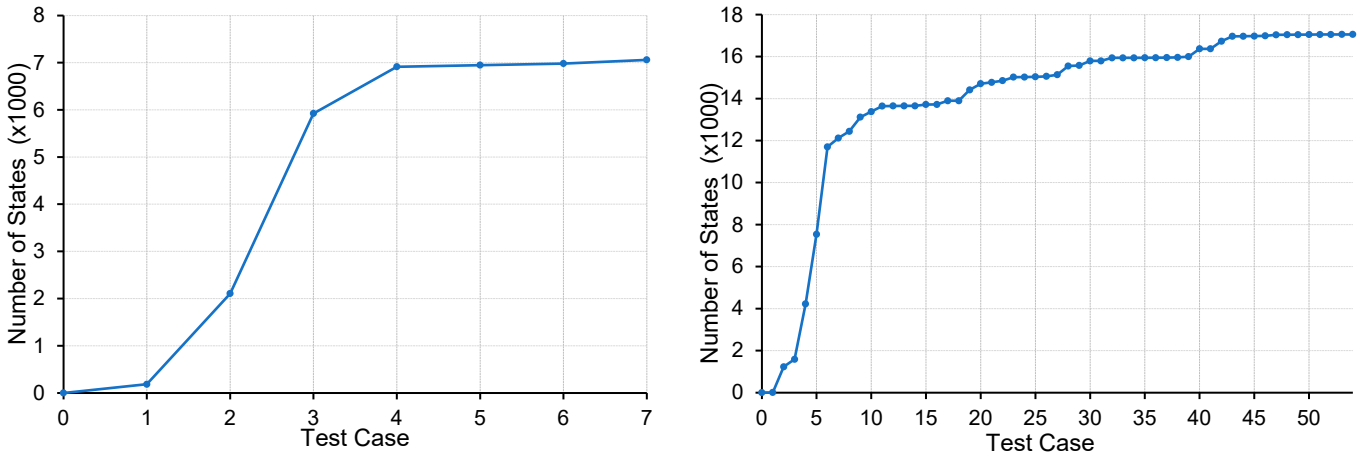

Figure 4. Growth dynamics of the number of unique states stored in the Q-table across training sessions. Subfigure (**a**) shows state accumulation over 7 test cases for BitmapPlusPlus, where saturation is reached quickly. Subfigure (**b**) shows the growth pattern across 54 training sessions for Hjson, which exhibits prolonged and more gradual Q-table expansion due to higher API complexity and state diversity.

3.5. Q-Table Exploitation Phase

3.5.1. Stability Analysis of Branch Coverage Under Different Action Selection Policies

To assess the impact of different action selection strategies on test case quality, we performed test case synthesis using the trained Q-table obtained after the reinforcement learning phase is complete. For each action selection policy configuration, 150 test cases were independently generated using Algorithm 3 and *maxRollback* parameter equal 30. The synthesized test cases were executed and evaluated based on branch coverage, allowing us to analyze the stability and effectiveness of different policy settings. The parameters of the evaluated action selection policies are summarized in Table 7.

Table 7. Parameters of the action selection policies used in Algorithm 3 for test case synthesis.

Configuration	Policy Type	Parameters	Value
QLEG1	*εGreedy*	ε	0.1
QLEG2			0.15
QLEG3			0.2
QLB1	Boltzmann (Softmax)	τ	0.8
QLB2			1.5
QLB3			3.0
RAND (Baseline)	Uniform Random Distribution	-	-

Figure 5a shows the distribution of branch coverage achieved by the ε-Greedy policy series for TC synthesis in the BitmapPlusPlus library. As evident from the plot, all three policy configurations exhibit consistently higher median coverage values compared to the RAND baseline policy. QLEG1 achieves the highest median coverage of 28.2%, whereas

the median for RAND is only 23.2%, with significantly higher variance. Figure 5b presents results for the Boltzmann (Softmax) selection policy, which also maintains stable median coverage values. The QLB3 configuration reaches a median of 28.5%.

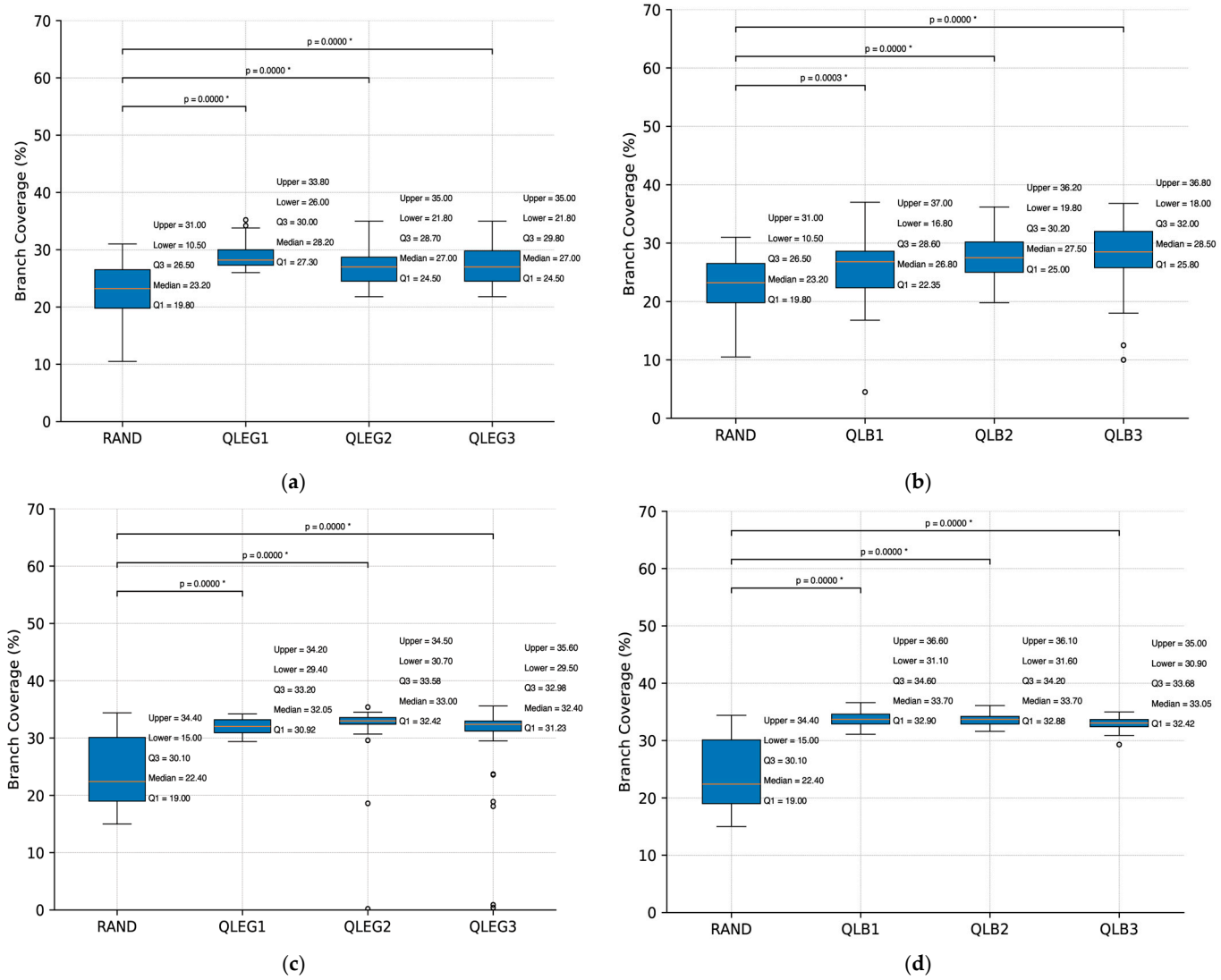

Figure 5. Branch coverage distributions obtained using different action selection policies compared to the RAND baseline. For each configuration, 150 synthesized test cases were evaluated, and branch coverage was computed. The boxplots display the median, interquartile range, and outliers (°). Horizontal brackets indicate pairwise differences assessed by the Mann-Whitney U-test. An asterisk (*) indicates that the difference is statistically significant ($p < 0.05$). (**a**) BitmapPlusPlus ε-Greedy. (**b**) BitmapPlusPlus Boltzmann. (**c**) Hjson ε-Greedy. (**d**) Hjson Boltzmann.

Figure 5c shows branch coverage distributions for Hjson under the ε-Greedy policy. The QLEG configurations yield consistent performance in the range from 30.92% to 35.6%, surpassing the RAND baseline. The highest median, achieved by QLEG2, is 33%. In Figure 5d, the Boltzmann (Softmax) policy demonstrates high stability, with branch coverage consistently ranging between 32.42% and 36.6%. The highest median achieved is 33.7%.

3.5.2. Comparison of Action Selection Policies by Branch Coverage Growth Dynamics

Figure 6 illustrates the progression of branch coverage during the execution of TCs synthesized using QLEG2 and QLB2 policies in Algorithm 3, in comparison with the RAND (baseline), for the BitmapPlusPlus and Hjson libraries, respectively.

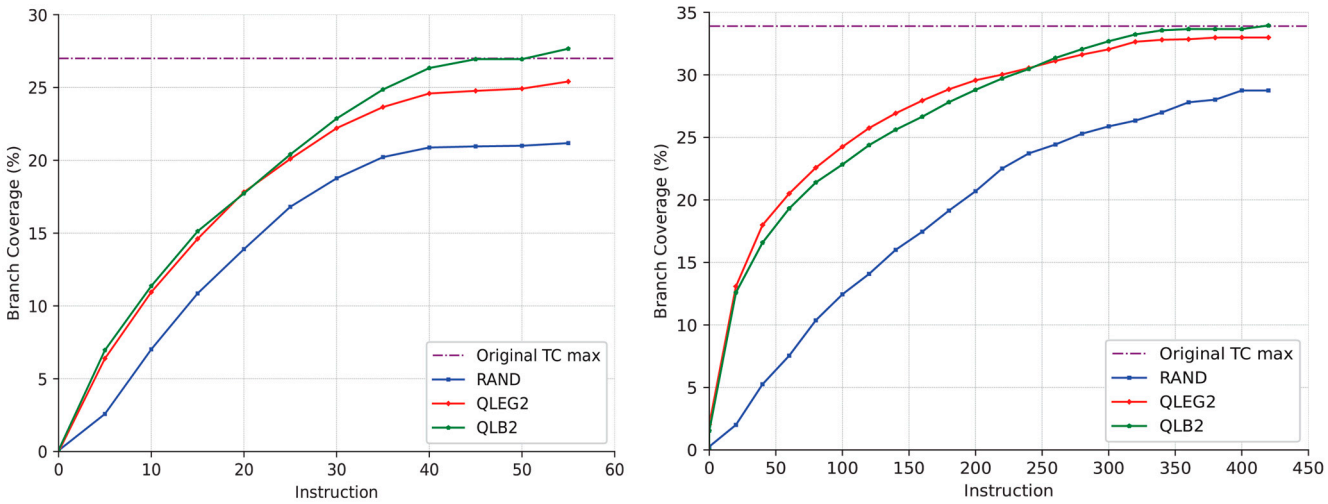

Figure 6. The dynamics of branch coverage growth as a function of the number of executed instructions for test cases synthesized using the QLEG2 and QLB2 policies, in comparison to the RAND baseline. The instructions count axis is aligned. The curves show the mean branch coverage across 150 synthesized test cases per configuration. The dashed purple line represents the maximum coverage achieved by the original test case. Subfigure (**a**) corresponds to the BitmapPlusPlus library and (**b**) to Hjson.

In Figure 6a, which corresponds to the BitmapPlusPlus library, the learned policies QLEG2 and QLB2 exhibit a rapid increase in coverage during the first 40 instructions. Both policies significantly outperform RAND across all instruction ranges. QLB2 achieves the highest coverage, reaching the original TC's maximum around 50 instructions, whereas RAND saturates around 24% and fails to reach the reference line. The coverage curves for QLEG2 and QLB2 are closely aligned, yet QLB2 consistently shows a slight advantage. Although QLB2 ultimately outperforms, RAND does not fall far behind and demonstrates moderate success. This behavior can be attributed to the lower API complexity and more uniform coverage opportunities in the BitmapPlusPlus library, which increase the chances for even random strategies to hit effective action sequences without exhaustive search.

In Figure 6b for Hjson library, the differences become even more pronounced. QLEG2 and QLB2 reach over 30% branch coverage within the first 200–250 instructions, while the RAND policy requires more than 400 instructions to approach 29%. The original TC's maximum is surpassed by QLB2 and nearly matched by QLEG2. The learned policies demonstrate higher efficiency and stability, especially in the plateau region after 300 instructions, where RAND continues to grow slowly and remains below the optimal level.

The weaker performance of RAND in this case is a consequence of higher API complexity, which leads to frequent invalid action sequences and early termination due to the maxRollback constraint. Since invalid actions cause full resets and exhaust the retry budget, RAND is unable to form long, coherent sequences that effectively explore deep control-flow paths.

3.5.3. Instruction Count Reduction as a Result of Learned Policy Execution

Figure 7 presents a comparative graph of the instruction counts in the original TCs and those synthesized using the QLB2 and QLEG2 policies.

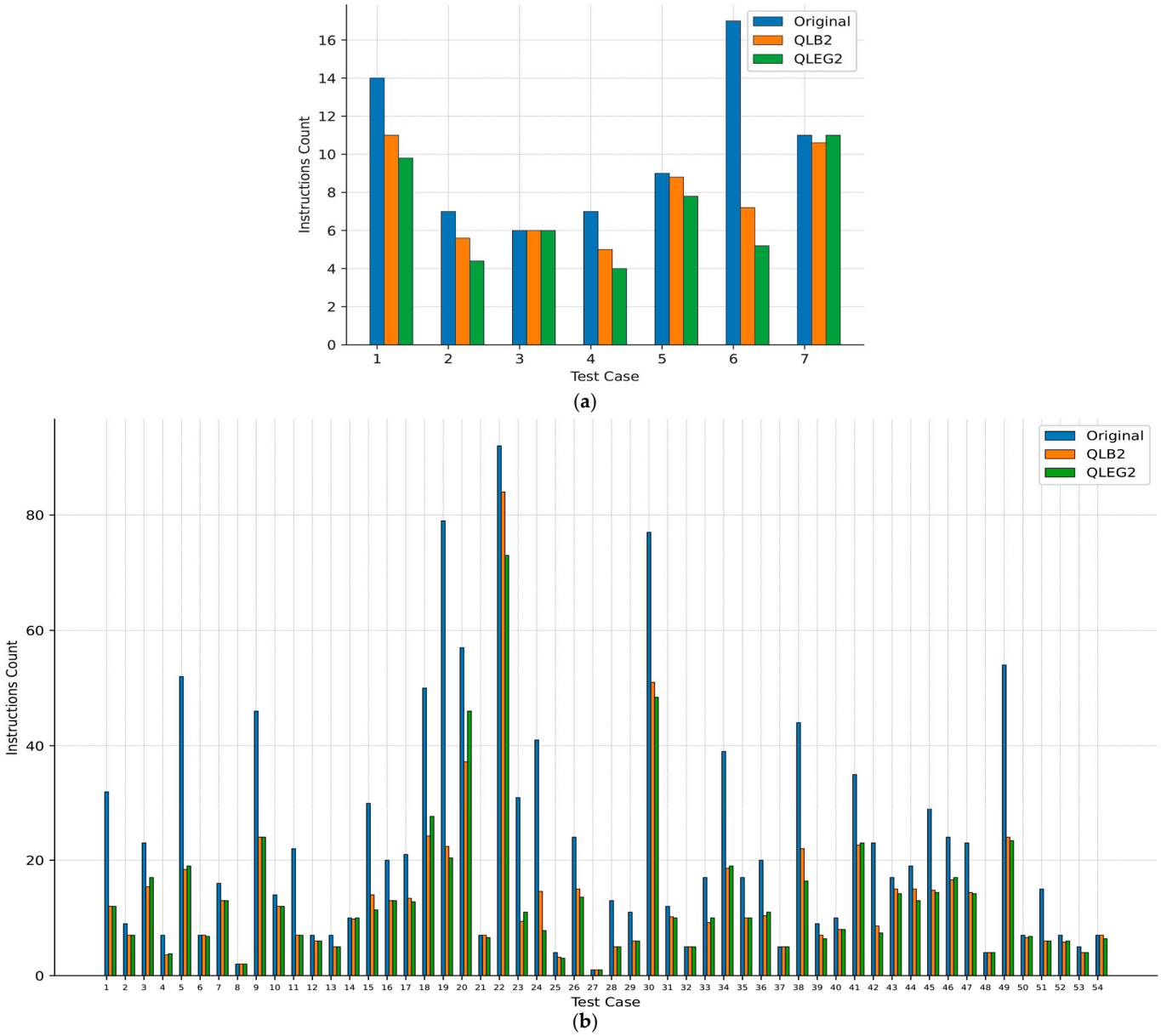

Figure 7. Comparative analysis of instruction counts per test case before and after test suite optimization for (**a**) BitmapPlusPlus and (**b**) Hjson libraries. Bars for QLB2 and QLEG2 represent the mean number of instructions across 150 independently synthesized test cases for each original test case.

For the BitmapPlusPlus library (Figure 7a), the results demonstrate a significant reduction in the number of instructions in most original TCs. In TC number 6, the number of instructions was reduced from 17 to 5. Similar reductions are observed in other tests where the method managed to maintain or even improve code coverage with fewer instructions. In certain cases, such as TCs 3 and 7, the number of instructions remained nearly unchanged. For the Hjson library (Figure 7b), it is evident that for nearly all TCs, both QLEG2 and QLB2 policies achieved a notable reduction in instruction count. This is especially apparent in the most complex original TC, where the number of instructions exceeded 90. In such cases, the method reduced the TC length by nearly half. In some TCs (e.g., 20 and 22), the

reduction is less pronounced, indicating limited optimization potential for shorter TCs where most actions are essential.

3.5.4. Comparative Summary

In Table 8, we present a comparative summary of the test suite synthesis results for the BitmapPlusPlus and Hjson libraries across three configurations: RAND (baseline), QLEG2 (ε-greedy), and QLB2 (Boltzmann). The evaluation metrics include mean + std branch coverage across 150 synthesized test cases, average instruction count, and compression coefficient, defined as the ratio between the instruction count of the synthesized test case and the original Equation (3). For BitmapPlusPlus, both QLEG2 and QLB2 significantly outperform RAND in terms of branch coverage, achieving 25.41% and 27.66% on average, compared to only 23.13% for RAND. These improvements are achieved while reducing the average instruction count from 71 (original) to 51.83 and 52.49, yielding compression coefficients of 0.27 and 0.26, respectively. For Hjson, QLB2 achieves the highest average branch coverage of 34.07%, closely followed by QLEG2 at 33.03%, both exceeding the original test suite's coverage of 33.9%. Additionally, the synthesized test suites are significantly shorter, with instruction counts reduced from 1259 to 419.36 for QLB2 and 405.40 for QLEG2, corresponding to compression coefficients of 0.67 and 0.68, respectively.

Table 8. Efficiency comparison of the test case synthesis policies for the BitmapPlusPlus and Hjson libraries in terms of branch coverage, average instruction count, and compression coefficient. The original test suites are used as a baseline. All synthesized instruction counts represent mean values across 150 independently generated test cases per configuration.

Configuration Name	Original TS Branch Coverage (%)	Synthesized TS Branch Coverage (%)	Original TS Instructions Count	Synthesized TS Instructions Count	Compression Coefficient
		BitmapPlusPlus			
RAND	27	23.13 ± 3.99	71	53.96 ± 6.96	0.24
QLEG2		25.41 ± 2.69		51.83 ± 2.86	0.27
QLB2		27.66 ± 2.49		52.49 ± 2.20	0.26
		Hjson			
RAND	33.9	24.6 ± 8.50	1259	213.92 ± 101.43	0.83
QLEG2		33.03 ± 1.39		405.40 ± 13.09	0.68
QLB2		34.07 ± 1.07		419.36 ± 7.33	0.67

4. Discussion

4.1. Summary

The obtained results demonstrate the ability of a classical Q-learning-based agent to effectively explore the space of C++ library API calls without requiring API specifications or documentation. Of particular importance is the fact that even in cases involving complex APIs, such as the Hjson library, the agent successfully stabilized the learning process and achieved an acceptable reward. Although fluctuations in rewards were observed for this case due to the high variability of TCs, the overall loss dynamics indicate effective accumulation of useful information in the Q-table.

Despite the absence of full Markovian properties, empirical results show that the state abstraction using a test case suffix is sufficient for rational agent training in TCs of limited length. Within such local heuristics, compact state representation allows for accelerated learning, avoids excessive branching of the state space in the Q-table, and prevents excessive memory usage. The scalability analysis showed that the Q-table size does not grow linearly for both libraries independent of the number of API functions they have. Rather, it is more influenced by the structural diversity of training TCs and the degree

of redundancy in the original TCs. For Hjson, despite having a larger API surface, the agent encountered a narrower state space due to high structural similarity among the original TCs, which limited the entropy of available sequences. Overall, we conclude that the size of the Q-table after training depends not only on the number of public functions in the library, but also on the variability of training TCs. Libraries with complex and extensive public interfaces create larger exploration spaces and require more memory, whereas libraries with lower combinatorial complexity allow the agent to stabilize more quickly. Although Hjson provided a larger number of original TCs, the lower entropy of API calls led to fewer unique states in the Q-table, despite a significantly longer training time.

The observed stability in synthesizing sequences across both libraries confirms the model's ability to transfer knowledge between different TCs within the same API. The pretraining strategy before each new training episode allowed the agent to adapt to a new environment, which is particularly beneficial in time-constrained training settings. Among the tested configurations, QLB2 demonstrates the most adequate performance, achieving the highest mean branch coverage for both libraries also maintaining a competitive compression coefficient. QLEG2 shows near-adequate results, reaching slightly lower coverage with comparable compression, suggesting it is a viable alternative when deterministic behavior is preferred. In contrast, the RAND baseline policy is inadequate, with significantly lower coverage indicating that random synthesis does not produce efficient test cases. The analysis of instruction counts, and coverage metrics demonstrates that synthesized TCs using QLB2 and QLEG2 policies consistently maintain or improve branch coverage while significantly reducing instruction count. For the Hjson library, both policies achieved coverage values comparable to the original TS with a nearly 2–3 factor compression (compression coefficient 0.67–0.68). For BitmapPlusPlus, the QLB2 policy matched the original 27% branch coverage with a compression coefficient of 0.26, confirming the ability of the approach to construct compact and effective TCs. However, the compression effect was less pronounced for certain configurations (e.g., QLEG2 on BitmapPlusPlus), where the reduced TC length did not lead to coverage improvement. This suggests that while the method is effective in optimizing longer and more redundant test cases, its benefit for already concise TCs is naturally bounded. These results underline the efficiency of policy-guided synthesis.

Overall, the proposed method demonstrates strong adaptability, effectiveness in synthesizing relevant TCs, and potential for scaling to new software libraries. The integration of Q-learning with dynamic API exploration is of value, as it eliminates the dependency on formal specifications.

4.2. Limitations and Threats to Validity

This study has several limitations that may affect the generalizability, reliability, and reproducibility of the proposed method.

4.2.1. Internal Validity

The performance of the Q-learning agent depends on multiple factors, including the reward function, training parameters, and exploration policy. In this study, we used fixed configurations during training for the reward function, learning rate, and discount factor. The ε-greedy strategy was used during training with a linear decay schedule for ε. These values were manually tuned for stability and remained non-adaptive throughout training. While this ensures comparability across test cases and reproducibility, it may limit the agent's ability to dynamically adjust to varying convergence speeds or difficulty levels. Furthermore, since learning is driven by dynamic execution feedback, inaccuracies in instrumentation or reward evaluation could introduce bias. The original test suites were

used as the primary source of structure and action space definition, which may introduce biases if the original tests are incomplete or non-representative.

4.2.2. External Validity

The experiments were conducted on only two open-source C++ libraries selected for their diversity and availability. While they differ in API structure and complexity, results may not generalize to larger or more complex software systems. Future work should include a broader range of libraries to better evaluate scalability and adaptability across different domains.

4.2.3. Construct Validity

Our approach evaluates test quality using code coverage, instruction count, and compression ratio. These metrics, although standard in the TSO literature, may not fully reflect fault detection ability or behavioral correctness. No explicit fault injection was performed, which limits assessment of the approach's effectiveness in detecting regressions or critical defects.

4.2.4. Methodological Limitations

A key limitation of the proposed method is its reliance on existing test suites for action space extraction. The agent is not capable of synthesizing novel API calls that were not previously observed. This dependency reduces autonomy in scenarios where original test traces are incomplete or unavailable. Moreover, we compared our approach only to a single baseline which is random action selection (RAND). The comparison provides a minimal reference point but does not reflect the full spectrum of possible strategies. Although no established method exists for test case prioritization and minimization in undocumented C++ libraries, comparison with additional baselines would strengthen the conclusions.

4.3. Future Work

1. The current solution focuses solely on the level of traced instructions. Integrating argument synthesis would allow for producing more variative TCs with deeper semantics and move toward a fully automatic generation of TCs.
2. The current evaluation is limited to two C++ libraries. To increase external validity, future experiments should incorporate a broader and more diverse set of libraries, including ones with larger APIs, deeper call hierarchies, and domain-specific constraints.
3. While random action sampling (RAND) was used as a minimal reference, future work should include comparisons with classical or hybrid techniques (e.g., greedy set cover, clustering, ILP, or classification-based approaches) to better position the proposed method within the TSM landscape.
4. The use of tabular Q-learning (Q-table) limits the scalability of the approach. Incorporating neural architectures such as Deep Q-Networks (DQN) or Actor-Critic methods could enable the handling of larger state and action spaces.
5. Combining the agent-based approach with classification, evolutionary, or statistical techniques may improve the quality of synthesized TCs and enhance adaptability to different types of APIs.
6. The feasibility of such enhancements is supported by prior research on formalizing testing stages through mathematical modeling [40,41] as well as on evaluating the quality of obfuscated code [42], which provide foundational insights into structured testing processes and test case evaluation.

7. Future research may extend the method toward distributional reinforcement learning [43,44], which would allow the agent to consider not only the expected utility of an action but also the distribution of its outcomes.

5. Conclusions

This study introduces an intelligent method of test case synthesis, specifically designed for C++ libraries lacking formal specifications or API documentation. The method employs a classical Q-learning agent to dynamically construct sequences of API calls based solely on execution feedback and coverage metrics. In contrast to traditional test suite optimization (TSO) techniques that operate by selecting or minimizing existing test cases, our method synthesizes new, compact test cases guided by a learned policy derived from observed execution behavior. To the best of our knowledge, this is the first work to explore Q-learning for fine-grained TSM as test case synthesis at the API sequence level in undocumented C++ environments. While prior studies have leveraged reinforcement learning for tasks such as test case prioritization [16,17], fuzzing [18], and test reduction via mutation coverage [8], these approaches typically assume the availability of structured test artifacts, specifications, or mutation operators. Our method, by contrast, operates in black-box conditions and does not rely on predefined specifications or instrumentation beyond coverage tracing.

Experimental evaluation on two open-source C++ libraries, Hjson and BitmapPlusPlus, demonstrates that the proposed method achieves substantial instruction-level compression while preserving or even improving original branch coverage. Specifically, the QLB2 policy reduced the instruction count of test cases by up to 67% in Hjson while slightly increasing coverage from 33.9% to 34.07%. In BitmapPlusPlus, QLB2 achieved the maximum branch coverage 27.66% with 26% reduction. These results outperform a random baseline and approach the reduction rates observed in classical methods. For example, authors [8] reported that greedy TSM techniques reduce test suites to approximately 70% on average without compromising fault detection. Moreover, dynamic slicing methods such as that proposed in [11] achieved up to a 52% reduction in redundant test operations across real-world Java projects, whereas our learned policies achieved even greater compression across libraries with varying complexity. The ability to synthesize compact, semantically valid test cases without human guidance or specification makes the proposed method particularly attractive for automated black-box testing of modern C++ APIs.

Future work will include benchmarking against additional state-of-the-art TSO methods such as ILP-based [12,13], clustering [19], or classification-driven approaches [20], as well as extending the model to support argument synthesis, deep reinforcement learning, and possibly multi-agent coordination for improving exploration efficiency and scalability of our method. We believe our findings contribute to closing the gap in automated test generation for undocumented systems and confirm the viability of reinforcement learning as a foundation for scalable, adaptive test case synthesis in resource-constrained or specification-limited environments.

Author Contributions: Conceptualization, O.K. and M.H.; methodology, M.H. and S.S.; software, M.H.; validation, S.S. and O.K.; formal analysis, M.H.; investigation, M.H. and S.S.; resources, P.M.; data curation, O.C.; writing—original draft preparation, M.H.; writing—review and editing, S.S.; visualization, M.H. and O.C.; supervision, S.S. and P.M.; project administration, O.K.; funding acquisition, P.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The raw data supporting the conclusions of this article will be made available by the authors on request.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

API	Application Programming Interface
CI	Continuous Integration
TS	Test Suite
TC	Test Case
TSO	Test Suite Optimization
TSM	Test Suite Minimization
TSR	Test Suite Reduction
ILP	Integer Linear Programming
TSP	Test Suite Prioritization
RL	Reinforcement Learning

References

1. Coviello, C.; Romano, S.; Scanniello, G.; Marchetto, A.; Corazza, A.; Antoniol, G. Adequate vs. inadequate test suite reduction approaches. *Inf. Softw. Technol.* **2020**, *119*, 106224. [CrossRef]
2. Pan, R.; Bagherzadeh, M.; Ghaleb, T.A.; Briand, L. Test Case Selection and Prioritization Using Machine Learning: A Systematic Literature Review. *Empir. Softw. Eng.* **2022**, *27*, 29. [CrossRef]
3. Mehmood, A.; Ilyas, Q.M.; Ahmad, M.; Shi, Z. Test Suite Optimization Using Machine Learning Techniques: A Comprehensive Study. *IEEE Access* **2024**, *12*, 168645–168671. [CrossRef]
4. Yoo, S.; Harman, M. Regression Testing Minimization, Selection and Prioritization: A Survey. *Softw. Test. Verif. Reliab.* **2012**, *22*, 67–120. [CrossRef]
5. Khan, S.U.R.; Lee, S.; Javaid, N.; Abdul, W. A systematic review on test suite reduction: Approaches, experiment's quality evaluation, and guidelines. *IEEE Access* **2018**, *6*, 27865–27890. [CrossRef]
6. Cruciani, E.; Miranda, B.; Verdecchia, R.; Bertolino, A. Scalable Approaches for Test Suite Reduction. In Proceedings of the 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 419–429. [CrossRef]
7. Nayab, S.; Wotawa, F. Testing and reinforcement learning: A structured literature review. In Proceedings of the 24th IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), Cambridge, UK, 1–5 July 2024; pp. 326–335. [CrossRef]
8. Jehan, S.; Wotawa, F. An Empirical Study of Greedy Test Suite Minimization Techniques Using Mutation Coverage. *IEEE Access* **2023**, *11*, 65427–65442. [CrossRef]
9. Zeller, A.; Hildebrandt, R. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* **2002**, *28*, 183–200. [CrossRef]
10. Ju, X.; Jiang, S.; Chen, X.; Wang, X.; Zhang, Y.; Cao, H. HSFal: Effective Fault Localization Using Hybrid Spectrum of Full Slices and Execution Slices. *J. Syst. Softw.* **2014**, *90*, 3–17. [CrossRef]
11. Vahabzadeh, A.; Stocco, A.; Mesbah, A. Fine-Grained Test Minimization. In Proceedings of the 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 27 May–3 June 2018; pp. 210–221. [CrossRef]
12. Ma, X.; Sheng, B.; Ye, C. Test-Suite Reduction Using Genetic Algorithm. In Proceedings of the 6th International Conference on Advanced Parallel Processing Technologies, APPT 2005, Hong Kong, China, 27–28 October 2005; Cao, J., Nejdl, W., Xu, M., Eds.; Lecture Notes in Computer Science. Springer: Berlin/Heidelberg, Germany, 2005; Volume 3756, pp. 253–261. [CrossRef]
13. Xue, Y.; Li, Y.-F. Multi-objective Integer Programming Approaches for Solving the Multi-criteria Test-suite Minimization Problem: Towards Sound and Complete Solutions of a Particular Search-based Software-engineering Problem. *ACM Trans. Softw. Eng. Methodol.* **2020**, *29*, 20. [CrossRef]
14. Gu, S.; Mesbah, A. Scalable Similarity-Aware Test Suite Minimization with Reinforcement Learning. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 172. [CrossRef]
15. Tahvili, S.; Hatvani, L.; Felderer, M.; Afzal, W.; Bohlin, M. Automated Functional Dependency Detection Between Test Cases Using Doc2Vec and Clustering. In Proceedings of the 2019 IEEE International Conference on Artificial Intelligence Testing (AITest), Newark, CA, USA, 4–9 April 2019; pp. 19–26. [CrossRef]
16. Waqar, M.; Imran; Zaman, M.; Muzammal, M.; Kim, J. Test Suite Prioritization Based on Optimization Approach Using Reinforcement Learning. *Appl. Sci.* **2022**, *12*, 6772. [CrossRef]

17. Bagherzadeh, M.; Kahani, N.; Briand, L. Reinforcement Learning for Test Case Prioritization. *IEEE Trans. Softw. Eng.* **2022**, *48*, 2836–2856. [CrossRef]
18. Zhang, Z.; Cui, B.; Chen, C. Reinforcement Learning-Based Fuzzing Technology. In Proceedings of the 14th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS-2020), Lodz, Poland, 1–3 July 2020; Barolli, L., Poniszewska-Maranda, A., Park, H., Eds.; Springer: Cham, Switzerland, 2021; Volume 1195, pp. 217–226. [CrossRef]
19. Pham, V.-T.; Böhme, M.; Roychoudhury, A. AFLNET: A Greybox Fuzzer for Network Protocols. In Proceedings of the 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), Porto, Portugal, 24–28 October 2020; pp. 460–465. [CrossRef]
20. Rosenbauer, L.; Stein, A.; Pätzel, D.; Hähner, J. XCSF with Experience Replay for Automatic Test Case Prioritization. In Proceedings of the 2020 IEEE Symposium Series on Computational Intelligence (SSCI), Canberra, ACT, Australia, 1–4 December 2020; pp. 1307–1314. [CrossRef]
21. Durmaz, E.; Tümer, M.B. Intelligent Software Debugging: A Reinforcement Learning Approach for Detecting the Shortest Crashing Scenarios. *Expert Syst. Appl.* **2022**, *198*, 116722. [CrossRef]
22. Sharif, A.; Marijan, D.; Liaaen, M. DeepOrder: Deep Learning for Test Case Prioritization in Continuous Integration Testing. In Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), Luxembourg, 27 September–1 October 2021; pp. 525–534. [CrossRef]
23. Marijan, D.; Gotlieb, A.; Sapkota, A. Neural Network Classification for Improving Continuous Regression Testing. In Proceedings of the 2020 IEEE International Conference on Artificial Intelligence Testing (AITest), Oxford, UK, 25–27 May 2020; pp. 123–124. [CrossRef]
24. Xiao, L.; Miao, H.; Shi, T.; Huang, Y. LSTM-Based Deep Learning for Spatial-Temporal Software Testing. *Distrib. Parallel Databases* **2020**, *38*, 687–712. [CrossRef]
25. Saidani, I.; Ouni, A.; Mkaouer, M.W. Improving the Prediction of Continuous Integration Build Failures Using Deep Learning. *Autom. Softw. Eng.* **2022**, *29*, 21. [CrossRef]
26. Joseph, A.K.; Radhamani, G. Hybrid Test Case Optimization Approach Using Genetic Algorithm with Adaptive Neuro-Fuzzy Inference System for Regression Testing. *J. Test. Eval.* **2017**, *45*, 2283–2293. [CrossRef]
27. Raamesh, L.; Jothi, S.; Radhika, S. Test Case Minimization and Prioritization for Regression Testing Using SBLA-Based AdaBoost Convolutional Neural Network. *J. Supercomput.* **2022**, *78*, 18379–18403. [CrossRef]
28. Ufuktepe, E.; Tuglular, T. Application of the Law of Minimum and Dissimilarity Analysis to Regression Test Case Prioritization. *IEEE Access* **2023**, *11*, 57137–57157. [CrossRef]
29. Meleshko, Y.; Raskin, L.; Semenov, S.; Sira, O. Methodology of Probabilistic Analysis of State Dynamics of Multi-Dimensional Semi-Markov Dynamic Systems. *East.-Eur. J. Enterp. Technol.* **2019**, *6*, 6–13. [CrossRef]
30. Han, D.; Mulyana, B.; Stankovic, V.; Cheng, S. A Survey on Deep Reinforcement Learning Algorithms for Robotic Manipulation. *Sensors* **2023**, *23*, 3762. [CrossRef]
31. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*, 2nd ed.; MIT Press: Cambridge, MA, USA, 2018.
32. Taylor, M.E.; Stone, P. Transfer Learning for Reinforcement Learning Domains: A Survey. *J. Mach. Learn. Res.* **2009**, *10*, 1633–1685.
33. Pan, S.J.; Yang, Q. A Survey on Transfer Learning. *IEEE Trans. Knowl. Data Eng.* **2010**, *22*, 1345–1359. [CrossRef]
34. Ćorović, A.; Ilić, V.; Đurić, S.; Marijan, M.; Pavković, B. The Real-Time Detection of Traffic Participants Using YOLO Algorithm. In Proceedings of the 2018 26th Telecommunications Forum (TELFOR), Belgrade, Serbia, 20–21 November 2018; pp. 1–4. [CrossRef]
35. Kaelbling, L.P.; Littman, M.L.; Moore, A.W. Reinforcement Learning: A Survey. *J. Artif. Intell. Res.* **1996**, *4*, 237–285. [CrossRef]
36. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; Riedmiller, M. Human-Level Control through Deep Reinforcement Learning. *Nature* **2015**, *518*, 529–533. [CrossRef]
37. Hulevych, M. CIDER: Assisted Automation Tool for C++ Libraries Testing. *Control Navig. Commun. Syst.* **2024**, *2*, 74–77. [CrossRef]
38. Beazley, D.M. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++. In Proceedings of the 4th Annual Tcl/Tk Workshop, Monterey, CA, USA, 6–10 July 1996; Available online: https://www.swig.org/papers/Tcl96/tcl96.html (accessed on 24 June 2025).
39. Hunter, J.D. Matplotlib: A 2D Graphics Environment. *Comput. Sci. Eng.* **2007**, *9*, 90–95. [CrossRef]
40. Semenov, S.; Liqiang, Z.; Weiling, C. Penetration Testing Process Mathematical Model. In Proceedings of the 2020 IEEE International Conference on Problems of Infocommunications. Science and Technology (PIC S&T), Kharkiv, Ukraine, 6–9 October 2020; pp. 142–146. [CrossRef]
41. Semenov, S.; Liqiang, Z.; Weiling, C.; Davydov, V. Development of a Mathematical Model for the Software Security Testing First Stage. *East.-Eur. J. Enterp. Technol.* **2021**, *3*, 24–34. [CrossRef]
42. Semenov, S.; Davydov, V.; Voloshyn, D. Obfuscated Code Quality Measurement. In Proceedings of the 2019 XXIX International Scientific Symposium "Metrology and Metrology Assurance" (MMA), Sozopol, Bulgaria, 7–11 October 2019; pp. 1–6. [CrossRef]

43. Bellemare, M.G.; Dabney, W.; Munos, R. A Distributional Perspective on Reinforcement Learning. In Proceedings of the 34th International Conference on Machine Learning (ICML 2017), Sydney, Australia, 6–11 August 2017; pp. 449–458. Available online: https://proceedings.mlr.press/v70/bellemare17a.html (accessed on 18 June 2025).
44. Son, K.; Kim, D.; Kang, W.J.; Hostallero, D.E.; Yi, Y. QTRAN: Learning to factorize with transformation for cooperative multi-agent reinforcement learning. In Proceedings of the 36th International Conference on Machine Learning, Long Beach, CA, USA, 9–15 June 2019; pp. 5887–5896.