Journal of Trends in Computer Science and Smart Technology (ISSN: 2582-4104)
www.irojournals.com/tcsst/

Graph Neural Network and Reinforcement Learning Framework for Test Case Prioritization, Selection, and Reduction

Sowmyadevi S.[1], Shashi Mehrotra[2]

Department of Computer Science and Engineering, SRM Institute of Science and Technology, Delhi NCR Campus, Modinagar, Ghaziabad, India.

Email: [1]ss2860@srmist.edu.in, [2]shashim@srmist.edu.in

Abstract

The process of regression testing is generally done under critical time and resource constraints. In the existing approaches, Test Case Prioritization (TCP), Test Case Selection (TCS), and Test Suite Reduction (TSR) have been treated as different optimization problems. This is based on simple heuristics that do not use valuable information that can be extracted from the program under test. Moreover, they do not allow for making internally consistent budget-conscious decisions. This paper introduces a unified approach that combines various variants of Graph Neural Networks (GNNs) and Reinforcement Learning (RL) for the solutions of TCP, TCS, and TSR. In our proposal, the representation of regression artifacts such as test cases, code entities, and faults is used as nodes in the typed graph, while the edges are used to show the relations between these artifacts. The relation-aware GNN is used for generating test case embeddings, reflecting the distance between the test cases and the changed areas as well as the areas known to have issues in the past. The actor-critic RL agent uses these test case embeddings and the budget to decide whether to run, skip, or discard test cases. The performance of our proposal outstrips coverage-based heuristics, history-based ranking, a GA-based search, an enhanced QPSO, and two learning-based ablations, as validated by our experiments on four Java projects from Defects4J. In terms of various programs and budgets, using GNN–RL has led to an improvement in the Average Percentage of Faults Detected (APFD) and Cost-cognizant Average Percentage of Faults Detected (APFDc) by approximately 6–9 percentage points on lower budgets. In addition, it is possible to obtain a 42% suite reduction and a 48% cost reduction while still retaining 98% of all fault-revealing tests in the final suite. The results show that it is possible to obtain promising solutions for adaptive regression testing using budget-aware reinforcement learning and graph-based representation learning.

Keywords: Regression Testing, Test Case Prioritization, Test Suite Reduction, Test Case Selection, Graph Neural Networks, Reinforcement Learning, Search-Based Software Engineering.

1. Introduction

In today's complex software systems, changes are constantly being made, including the addition of new features, bug fixes, and new releases in the CI/CD pipeline. However, the risk of regression is always present with each change. In complex industrial systems, there are

DOI: https://doi.org/10.36548/jtcsst.2026.1.007
Received: 29.01.2026, received in revised form: 25.02.2026, accepted: 10.03.2026, published: 19.03.2026
© 2026 Inventive Research Organization. This is an open access article under the Creative Commons Attribution-NonCommercial International (CC BY-NC 4.0) License

thousands of tests with different costs and fault detection capabilities. Hence, it is not possible to run all tests with each change. Therefore, in the context of regression testing, the focus is mainly on the selection of test cases to be executed and the order in which they should be executed, given the budget constraints in terms of time and resources [1],[5].

In this problem setting, three problems have been identified: Test Case Prioritization (TCP), Test Case Selection (TCS), and Test Suite Reduction (TSR). In Test Case Prioritization, the goal is to find an ordering of test cases to reveal faults as early as possible, typically by optimizing the Average Percentage of Faults Detected (APFD) and Cost-cognizant Average Percentage of Faults Detected (APFDc). TCP aims to detect faults as soon as possible. This is often expressed in terms of Average Percentage of Faults Detected (APFD) and Cost-cognizant Average Percentage of Faults Detected (APFDc). TCS selects a small number of tests that are most important for a specific change or budget. On the other hand, TSR aims to eliminate tests that are not needed without making it too difiicult to find faults [4],[7]–[9]. Recent tertiary and live reviews emphasized that these three approaches are complementary, yet many published techniques are challenging to implement in practice.

For the most effective formulation of the problem, TCP/TCS/TSR can be solved as an optimization problem. Software engineering approaches to regression testing have traditionally used search-based optimization to optimize a cost function that trades off coverage, fault detection, and cost, given permutations (for TCP) or selection vectors (for TCS/TSR) [11,13,12]. Other approaches have attempted to jointly solve the minimization and prioritization problem, e.g., with the use of deep models, and have shown that ranking-based approaches can achieve similar performance to minimization approaches when prior signals are reliable [2]. Learning-based approaches can still use flattened feature representations but may require additional logic to handle budgets or long-term reductions.

Reinforcement learning (RL) has been employed to model regression testing as a sequential decision-making problem where the agent uses a policy that strikes a balance between fault detection and resource usage. In a number of studies, TCP using RL was found to have greater adaptivity compared to traditional heuristics. Nevertheless, some of these RL techniques have not been able to exploit more advanced structural relationships between tests, code entities, and faults, as they maintain their state representation in a feature vector. Recently, Graph Neural Networks (GNNs), which are powerful models that can be trained to learn graph data, have been proposed. Several types of GNNs have been proposed for abstract syntax graphs, control flow graphs, code call graphs, and dependency graphs for code quality analysis and reasoning. Studies in mapping have demonstrated that graph learning algorithms can be applied in software engineering, where the behavior of a defect is affected by the dependency of software elements [19]-[23]. Nevertheless, to our best knowledge, no works have been done in developing a common logical formalism that combines heterogeneous software graphs with sequential decision-making in test coverage statements, test coverage triggers, and test coverage requirements, even though they have some affinities.

These gaps are addressed by proposing a unified framework using heterogeneous graphs and reinforcement learning to support a single decision process whose output is test case prioritization, test case selection, and test suite reduction. A heterogeneous graph is used to represent regression artifacts such as tests, program elements, and faults. This graph is then encoded using an encoder GNN, which allows information to be transferred across the graph. Test embeddings that represent information such as coverage, impact of changes, structure, and fault histories of individual artifacts are also constructed in the process. This is the RL agent's decision on whether to execute, skip, or permanently discard a test at each testing budget. The

reward allows the agent to adapt across versions while maintaining tests that reveal faults in the test pool.

2. Literature Review

Regression testing has gradually shifted from simple re-execution of legacy suites to a family of optimization problems that explicitly balance cost, coverage, and fault detection. Recent research evidence and CI-focused syntheses emphasize that test case prioritization, test case selection, and test suite reduction are best viewed as complementary mechanisms for controlling regression cost while preserving effectiveness [1],[3],[4]. TCP typically aims to maximize early fault detection (e.g., APFD, APFDc), TCS selects a subset of tests under budget constraints, and TSR removes redundant tests subject to adequacy or fault-detection criteria [4],[7]-[9]. Traditional techniques are largely based on coverage and execution history. Structural methods rank or select tests according to the code elements they exercise often using total or additional coverage of modified or fault-prone components, while history-based methods reorder tests that have failed recently or frequently [7]. To cope with the combinatorial nature of TCP, TCS, and TSR, search-based software engineering introduced metaheuristic optimization, where permutations or subsets of tests are encoded as candidate solutions and optimized using algorithms such as genetic algorithms, PSO, and their variants [11]–[13]. For example, QPSO has been adapted to jointly handle TCP, TCS, and TSR within a single optimization model [13]. Recent optimizers, including improved whale-based search and QPSO variants, continue to show that search-based techniques can deliver strong trade-offs between early detection and cost, although systematic reduction reviews still note common concerns related to scalability and repeated evaluation overhead in fast CI cycles [8],[12],[13].

Recent interest has turned to the application of Machine Learning (ML) and reinforcement learning to enhance the adaptiveness of regression testing. ML-based techniques, as described in [14], [15], use coverage, code metrics, change history, and past outcomes to obtain scores for selecting and prioritizing tests. Classifiers, regressors, or learning-to-rank models are trained on these scores. Models of this type can achieve better results than simple heuristics if feature engineering is effective. They generally optimize a surrogate loss instead of optimally APFD, APFDc or cost. The performance may degrade as systems evolve [14],[15]. RL-based approaches view TCP and TCS as sequential decision problems in which, given the current state (e.g., partial result, remaining budget), the agent chooses the next test and receives rewards for revealing faults or cost-aware utility [16]–[18]. Various studies on Deep RL frameworks reported improved APFD and fault detection latency and studied issues such as cost-aware reward, scalability and stability of the policy in the industry [16]–[18]. The majority of these approaches, however, are based on vectorized features or naive graph encodings, which largely treat tests as independent.

At the same time, software engineering is starting to gain interest in graph-based techniques and GNN-based techniques. Program artifacts like abstract syntax trees, control-flow graphs, call graphs, and dependency graphs form graphs naturally, and recent surveys on using deep learning for source code highlight that graph representations capture important syntactic and semantic structure [19],[20]. Graph neural networks and graph-based encodings have been increasingly used in tasks such as code classification, clone detection, defect prediction, bug localization, and vulnerability detection [21],[22],[23]. GNN generalize neural message passing to arbitrary graphs and have proven effective at modelling long-range dependencies in code and related artifacts. However, prior works have not yet optimized

regression tests through them: the above-mentioned works use graphs for static analysis or representation learning and not directly for driving regression test optimization, while most existing RL-based TCP/TCS works also rely on handcrafted feature vectors or simplified adjacency matrices, failing to account for full relations between tests, code entities, faults, and changes [5],[10],[14]-[18].

There is an obvious gap in the existing literature. Numerous TCP/TCS/TSR methods continue to manipulate flattened representations, such as coverage matrices, aggregated metrics, and feature vectors, which lose important relational structure. In addition, search-based approaches optimize explicit objectives but don't learn from historical campaigns. The focus of supervised ML lies on surrogate losses. Additionally, RL-based regression testing methods do not typically employ expressive graph encoders nor treat prioritization, selection, and long-term reduction in a unified manner [5],[8],[10],[11],[14]–[18]. Moreover, scalability and rapid adaptability remain challenging in CI/CD environments, where budgets, codebases, and test suites evolve continuously [14]–[18]. This paper addresses these gaps by proposing a graph-centric, RL-guided framework in which heterogeneous graphs represent tests, code entities, faults, and changes; a GNN learns test embeddings from these graphs; and an RL agent uses these embeddings to make sequential decisions about test ordering, selection, and reduction.

3. Preliminaries and Problem Formulation

This section introduces the notation, metrics, and formal models used throughout the paper. It first introduces the basic regression testing setting and the effectiveness and cost metrics used in the paper, namely APFD and APFDc, and then introduces the heterogeneous graph representation of regression artifacts. Finally, we formulate the reinforcement learning problem and provide the optimization objective of the proposed GNN-RL framework.

3.1 Notation and Basic Definitions

Let a software system evolve through a sequence of versions$P^{(1)}, P^{(2)}, \dots, P^{(V)}$, where $P^{(v)}$denotes the v-th version of the program under test. For this system, let $T = \{t_1, t_2, \dots, t_n\}$ be the global regression test suite, and let $T^{(v)} \subseteq T$denote the subset executed for version $P^{(v)}$. We assume that each test case $t_j$has an associated execution cost $c_j > 0$, and that each program version may expose a set of faults. Let $F^{(v)} = \{f_1^{(v)}, f_2^{(v)}, \dots, f_{m_v}^{(v)}\}$ be the set of faults that are known to be detectable in version $P^{(v)}$, and let $F = \bigcup_{v=1}^{V} F^{(v)}$ be the union of faults across all versions considered in the study. For a given version $P^{(v)}$, executing test $t_j$yields an outcome $y_j^{(v)} \in \{0,1\}$, where $y_j^{(v)} = 1$indicates that $t_j$reveals at least one fault in $F^{(v)}$, and 0otherwise. When fault-level information is available, we write $y_{j,i}^{(v)} \in \{0,1\}$ to denote whether $t_j$reveals a specific fault $f_i^{(v)}$. A test ordering (or prioritization) for a version $P^{(v)}$is a permutation $\pi^{(v)} = (\pi_1^{(v)}, \pi_2^{(v)}, \dots, \pi_{|T^{(v)}|}^{(v)})$, where $\pi_k^{(v)}$is the index of the test executed at position k. A selected subset is any $T_{\text{sel}}^{(v)} \subseteq T^{(v)}$, and a reduced suite is a subset $T_{\text{red}} \subseteq T$retained across multiple regression cycles according to a reduction strategy [4],[6]-[9].

3.2 Regression Testing Objectives and Metrics

To quantify the effectiveness and cost of regression testing techniques, we use standard metrics widely adopted in TCP, TCS, and TSR studies [1],[5],[14],[18].

3.2.1 Average Percentage of Faults Detected (APFD)

Consider a version $P^{(v)}$with test suite $T^{(v)}$of size n and a set $F^{(v)}$of m detectable faults. Let $\pi^{(v)}$be a prioritized ordering of $T^{(v)}$. For each fault $f_i \in F^{(v)}$, define TF_ias the position in $\pi^{(v)}$of the first test that reveals fault f_i. The Average Percentage of Faults Detected (APFD) for ordering $\pi^{(v)}$is then defined as [1],[5]:

$$\text{APFD}(\pi^{(v)}) = 1 - \frac{1}{nm}\sum_{i=1}^{m} TF_i + \frac{1}{2n} \quad (1)$$

APFD values range from 0 to 1, with higher values representing earlier average fault detection in the test execution order. APFD values are often multiplied by 100 to express them as percentages. In our setting, the policy outputs an executed ordering $\pi^{(v)}$for the subset actually run in a cycle (budgeted execution), while skipped/discarded tests do not appear in $\pi^{(v)}$. We compute APFD (and APFDc) on this executed ordering at each budget level: faults not detected within the executed subset are assigned the worst-case position $\text{TF}_i = |\pi^{(v)}| + 1$(and, for APFDc, the corresponding cumulative cost is set to the total executed cost), so aggressive skipping/discarding is penalized rather than rewarded. We additionally report INC alongside APFD/APFDc to summarize how many faults remain detectable after reduction.

3.2.2 Cost-Cognizant APFD (APFDc)

Equal test costs and equal fault severities are rarely realistic assumptions in practice. To account for unequal test costs and unequal fault severities, cost-sensitive variants of APFD add test costs and fault severities to the metric [5],[18].Let $c_j$denote the cost of the test at position $j$in ordering $\pi^{(v)}$, and let $C_{\text{tot}} = \sum_{j=1}^{n} c_j$ be the total cost of executing the entire suite. For each fault f_i, let $w_i > 0$denote its severity (or business impact weight), and let CC_ibe the cumulative cost incurred up to and including the first test that reveals fault f_i. A widely used cost- and severity-aware variant of APFD is then:

$$\text{APFDc}(\pi^{(v)}) = 1 - \frac{1}{C_{\text{tot}} \sum_{i=1}^{m} w_i} \sum_{i=1}^{m} w_i \, CC_i \quad (2)$$

Higher APFDc indicates that severe faults tend to be revealed earlier while incurring less cumulative cost.

3.2.3 Reduction and Cost Metrics

For a reduced suite $T_{\text{sel}}^{(v)} \subseteq T^{(v)}$, we are also interested in the degree of reduction and the resulting cost savings. Let $n = |T^{(v)}|$, $n_{\text{sel}} = |T_{\text{sel}}^{(v)}|$, and denote total cost of a set $X \subseteq T^{(v)}$by $C(X) = \sum_{t_j \in X} c_j$

We use the following metrics:

Reduction ratio (RR): $$\mathrm{RR}(T_{\mathrm{sel}}^{(v)}) = 1 - \frac{n_{\mathrm{sel}}}{n} \quad (3)$$

Cost reduction (CR): $$\mathrm{CR}(T_{\mathrm{sel}}^{(v)}) = 1 - \frac{C(T_{\mathrm{sel}}^{(v)})}{C(T^{(v)})}. \quad (4)$$

Inclusivity of fault-revealing tests (INC): if $F^{(v)}$is the set of faults detectable by $T^{(v)}$, and $F_{\mathrm{sel}}^{(v)}$is the set actually revealed by $T_{\mathrm{sel}}^{(v)}$, then $$\mathrm{INC}(T_{\mathrm{sel}}^{(v)}) = \frac{|F_{\mathrm{sel}}^{(v)}|}{|F^{(v)}|}. \quad (5)$$

High RR and CR indicate strong reduction and cost savings, but they must be interpreted together with APFD/APFDc and INC to ensure that fault detection capability is not compromised [3],[4],[8],[9]. In the remainder of the paper, we use APFD and APFDc primarily to quantify early fault detection, RR and CR to quantify resource savings, and INC to summarize how many faults remain detectable after reduction.

3.3 Heterogeneous Graph Representation of Regression Artefacts

Traditional regression testing techniques usually operate on matrices or handcrafted feature vectors, such as coverage matrices C and historical failure counts. To allow the GNN–RL framework to exploit richer structural relationships, we represent regression artefacts as a heterogeneous graph. For a given version $P^{(v)}$, we define a directed graph $G^{(v)} = (V^{(v)}, E^{(v)}, \tau_V, \tau_E)$, where $V^{(v)}$is the set of nodes, $E^{(v)} \subseteq V^{(v)} \times V^{(v)}$is the set of edges, $\tau_V: V^{(v)} \to$ {test,code,fault}assigns a node type, and $\tau_E: E^{(v)} \to \mathcal{R}$assigns an edge type from a finite set $\mathcal{R}$(e.g., “covers”, “detects”, “belongs-to-module”, “changed-in-version”).

We distinguish three primary node sets: $V_T^{(v)} = \{u_j: t_j \in T^{(v)}\}$, representing test cases, $V_C^{(v)} = \{u_k: s_k^{(v)} \in S^{(v)}\}$, representing code entities, and $V_F^{(v)} = \{u_i: f_i^{(v)} \in F^{(v)}\}$, representing faults. We encode key relations through typed directed edges $\tau_E(\cdot) \in \mathcal{R}$.if test $t_j$executes entity $s_k^{(v)}$, add $(u_j, u_k)$with $\tau_E(u_j, u_k) = \mathrm{covers}$. we do not assume oracle fault–test mappings; for each faulty version $P^{(v)}$, we obtain fault-revealing tests from observed outcomes by running the JUnit suite on the buggy version. Each defect instance is represented as a fault node $u_i \in V_F^{(v)}$, and for every such test $t_j$we add $(u_j, u_i)$with $\tau_E(u_j, u_i) = \mathrm{detects}$(optionally weighted by failure consistency). if $s_k^{(v)} \in \Delta^{(v)}$, we annotate its node features or connect it to a special “change” node. module/class nodes and call-graph edges can be added when available; the framework only requires a well-typed graph in which each test node connects to relevant code and fault context. In Section 4, a GNN encoder operates on $G^{(v)}$to produce test embeddings $\mathrm{h}_j^{(v)}$for each test node $u_j \in V_T^{(v)}$, aggregating information from its local neighborhood and, through message passing, from structurally related regions of the graph [19]–[23]. These embeddings constitute a compact, learned representation of the test’s context, which will feed the RL decision process.

3.4 Reinforcement Learning Formulation

We cast the joint TCP/TCS/TSR problem for a given regression cycle as a Markov decision process (MDP) [18,24]. An MDP is specified by a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma)$, where $\mathcal{S}$is the state space, $\mathcal{A}$is the action space, $P$is the state transition kernel, $R$is the reward function,

and $\gamma \in [0,1]$is a discount factor.For a specific version $P^{(v)}$, one episode corresponds to executing a regression cycle under a budget constraint. At each decision step k, the RL agent observes a state $s_k \in \mathcal{S}$, chooses an action $a_k \in \mathcal{A}$, receives a reward $r_k = R(s_k, a_k)$, and transitions to the next state s_{k+1}according to P.In the proposed framework, the state $s_k$encodes at least: the GNN-based embeddings of remaining test cases $\{\mathrm{h}_j^{(v)}\}$,indicators of which tests have already been executed or removed,the remaining budget (e.g., residual cost allowance) and cumulative cost so far, and summary statistics of faults detected so far (or proxies such as coverage achieved).

The action space $\mathcal{A}$is defined to support TCP, TCS, and TSR in a unified way. One practical design is to let actions take the form $a_k = (j_k, d_k)$, where $j_k$indexes a test t_{j_k}among those not yet decided upon, and $d_k \in$ {execute,skip,discard}specifies whether the test is (i) scheduled for execution in the current cycle (TCP/TCS), (ii) temporarily skipped for this cycle, or (iii) marked as a candidate for permanent removal from the suite (TSR). Alternative formulations can separate execution and reduction decisions across episodes; the specific choice in Section 4 aims to balance expressiveness and tractability. The reward signal $r_k$is designed to encourage early detection of important faults while penalizing excessive cost and harmful reduction. A generic form is

$$r_k = \alpha_1 \, \Delta\mathrm{FD}_k - \alpha_2 \, \Delta C_k - \alpha_3 \, \Delta\mathrm{Loss}_k, \tag{6}$$

where: $\Delta\mathrm{FD}_k$measures the incremental fault detection gain attributable to action a_k(e.g., number or severity-weighted sum of new faults revealed by executing t_{j_k}), $\Delta C_k$is the incremental cost incurred (typically c_{j_k}if the test is executed, or 0 otherwise), $\Delta\mathrm{Loss}_k$captures the long-term fault detection loss associated with discarding tests (e.g., an estimate of expected future faults that might be missed if $t_{j_k}$is permanently removed), and $\alpha_1, \alpha_2, \alpha_3 \geq 0$ are tunable coefficients that trade off these objectives. Given a policy $\pi_\theta(a \mid s)$parameterized by θ(e.g., neural network parameters), the expected return of the agent from an initial state distribution ρ is

$$J(\theta) = \mathbb{E}_{s_0 \sim \rho,\, a_k \sim \pi_\theta}\left[\sum_{k=0}^{K-1} \gamma^k r_k\right], \tag{7}$$

where $K$is the episode length (e.g., the number of decisions until the budget is exhausted or all tests are classified). The reinforcement learning objective is to find parameters $\theta^\star$that maximize $J(\theta)$. In Section 4, we instantiate this with a GNN-based state encoder and a deep RL algorithm (e.g., actor–critic), and detail how APFD/APFDc-related quantities influence $r_k$and the training loss [18],[24].

3.5 Combined TCP/TCS/TSR Optimization Problem

We now summarize the combined optimization goal that the GNN–RL framework is intended to approximate. For a given regression cycle on version $P^{(v)}$, the practitioner typically specifies a budget $B^{(v)}$(time or cost units) and possibly a minimum acceptable inclusivity level $\underline{\mathrm{INC}} \in (0,1)$. Let $\pi^{(v)}$denote the ordering of tests that are actually executed in this cycle (the TCP/TCS outcome), and let T_{red}denote the global reduced suite after applying TSR decisions across multiple cycles. For clarity, let $T_{\mathrm{exec}}^{(v)} \subseteq T^{(v)}$be the subset executed in cycle v, and let $\pi^{(v)}$be the ordering of this subset. We are interested in maximizing early fault detection while respecting budget and maintaining long-term suite quality.

At the level of a single cycle, a natural scalarized objective is

$$\max_{\pi^{(v)},\, T_{\text{exec}}^{(v)}} \Phi^{(v)} = \lambda_1 \,\text{APFD}(\pi^{(v)}) + \lambda_2 \,\text{APFDc}(\pi^{(v)}) + \lambda_3 \,\text{CR}(T_{\text{exec}}^{(v)}), \tag{8}$$

subject to

$$C(T_{\text{exec}}^{(v)}) \le B^{(v)}, \text{INC}(T_{\text{exec}}^{(v)}) \ge \underline{\text{INC}}, \tag{9}$$

where $\lambda_1, \lambda_2, \lambda_3 \ge 0$weight the relative importance of each component. Explicitly solving Equation (8)–(9) as a constrained combinatorial optimization problem across cycles is computationally prohibitive. Instead, the GNN–RL framework learns a policy $\pi_\theta$that, given the heterogeneous graph $G^{(v)}$, current budget status, and historical context, approximates decisions that yield high values of $\Phi^{(v)}$over time. The multi-objective structure is reflected in the reward design in Equation (6) and in the way APFD/APFDc and reduction metrics are aggregated during training.

4. Proposed GNN–RL Framework

This section presents the proposed framework that combines a heterogeneous graph neural network with a reinforcement learning agent to address test case prioritization, test case selection, and test suite reduction in a unified way. Building on the notation and formulation in Section 3, we first provide a high-level overview of the pipeline, then describe the construction of the heterogeneous graph, the GNN encoder, the RL decision process, the training procedure, and finally discuss complexity and implementation issues.

4.1 Overview of the Framework

For each regression cycle over the version $P^{(v)}$, the framework consists of four major stages:

1. **Graph Construction:** The available regression artefacts, such as test suite $T^{(v)}$, code entities $S^{(v)}$, known faults $F^{(v)}$, coverage matrix $C^{(v)}$, and change information $\Delta^{(v)}$) are represented as a heterogeneous graph $G^{(v)}$ as described in Section 3.3. The nodes of the graph represent the tests, code entities, and faults, while the edges represent the coverage, fault detection, and change information.

2. **Graph Embedding via GNN:** The heterogeneous GNN applies an operation $G^{(v)}$ o obtain the low-dimensional embeddings $\text{h}_j^{(v)}$ for each test node u_j. The process considers both the local information, such as coverage over changed code, and global information, such as proximity to fault-prone code.

3. **RL-Based Decision Making:** An RL agent, represented by neural networks that take in test embeddings and budget state, makes a sequence of "execute", "skip", or "discard" decisions for each test. The sequence of "execute" decisions induces a prioritized ordering $\pi^{(v)}$and a selected subset $T_{\text{exec}}^{(v)}$.

4. **Execution and Feedback:** The executed test suite consists of the selected tests, whose outcomes update the fault detection indicators $y_{j,i}^{(v)}$, and rewards are then computed based on Equation (6), which captures the APFD-like reward for early

fault detection, cost, and reduction-related penalties. These reward signals update the RL policy over episodes to improve the policy over a sequence of regression cycles. In summary, the architecture of the proposed framework is shown in Figure 1.

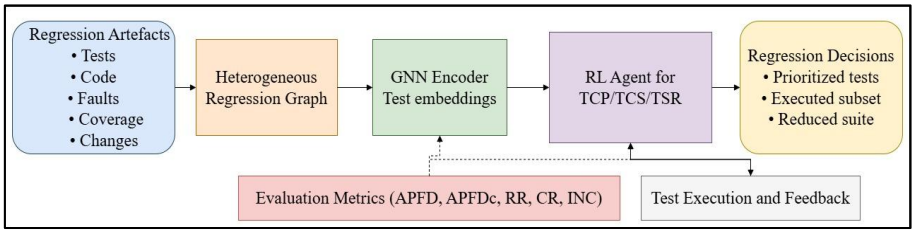

Figure 1. Overall GNN–RL-BSased Regression Testing Framework

4.2 Construction of the Heterogeneous Test Artefact Graph

The graph $G^{(v)}$introduced in Section 3.3 is instantiated in a way that balances representational richness with computational cost. We identify three types of nodes and several types of edges.

4.2.1 Node Types and Features

- **Test Nodes:** For each test case $t_j \in T^{(v)}$, we create a test node $u_j \in V_T^{(v)}$. Its initial feature vector $\mathrm{x}_j^{(T)}$can include: normalized execution cost $\tilde{c}_j$, historical failure rate over past versions (e.g., fraction of cycles where $t_j$revealed a fault), last execution outcome (pass/fail/unknown), simple static descriptors (e.g., test length or input size if available).

- **Code Nodes:** For each code entity $s_k^{(v)} \in S^{(v)}$, we create a code node $u_k \in V_C^{(v)}$. Its features $\mathrm{x}_k^{(C)}$may include: complexity metrics (e.g., cyclomatic complexity, lines of code), change indicator (binary flag for membership in $\Delta^{(v)}$), module or layer identifiers encoded as one-hot or learned embeddings.

- **Fault Nodes:** For each known detectable fault $f_i^{(v)} \in F^{(v)}$, we create a fault node $u_i \in V_F^{(v)}$with features $\mathrm{x}_i^{(F)}$capturing: severity weight w_i, type/category of the fault (e.g., logic error, API misuse), and historical persistence (e.g., number of versions where the fault was present).

4.2.2 Edge Types and Attributes

Edges encode relations among these nodes:

- **Test–Code Coverage Edges:** For each $C_{j,k}^{(v)} = 1$, we add an edge $e = (u_j, u_k)$of type "covers". Edge attributes may include: normalized execution frequency 1whether the coverage involves changed code ($s_k^{(v)} \in \Delta^{(v)}$).

- **Test–Fault Detection Edges:** When test $t_j$reveals fault $f_i^{(v)}$in historical runs, we add an edge of type "detects". Edge attributes can encode the version in which the detection occurred, or the time since last detection.

- **Code–Code Structural Edges:** Optional edges, such as "calls", "defines", "inherits-from", etc., connect code nodes according to the call graph or dependency graph. These edges represent structural proximity between tested entities covered by multiple tests, which enables the GNN to propagate the risk information. These edges form a heterogeneous graph, which can be processed using relation-aware GNN variants such as those described in [19]-[23]. The graph is created for each version, and the test and code node identities can be shared across versions, enabling knowledge sharing.

4.3 GNN Encoder for Test Case Representation

The GNN encoder maps the heterogeneous graph $G^{(v)}$to latent vectors $\mathrm{h}_u^{(\ell)}$for each node u, across $\ell = 0, \ldots, L$layers, where $\mathrm{h}_u^{(0)} = \mathrm{x}_u$is the initial feature vector. Given the heterogeneity of node and edge types, we adopt a relation-specific message-passing scheme inspired by relational graph convolutional networks (R-GCN) and heterogeneous graph attention networks [22],[23]. At each layer ℓ, the hidden state of node u is updated as:

$$\mathrm{h}_u^{(\ell+1)} = \sigma\Big(\sum_{r\in\mathcal{R}} \sum_{v\in\mathcal{N}_r(u)} \frac{1}{Z_{u,r}} \mathrm{W}_r^{(\ell)}\, \mathrm{h}_v^{(\ell)} + \mathrm{W}_0^{(\ell)}\, \mathrm{h}_u^{(\ell)}\Big) \tag{10}$$

Where $\mathcal{R}$is the set of edge types, $\mathcal{N}_r(u)$are the neighbors of $u$connected by relation r, $Z_{u,r}$is a normalization constant (e.g., number of neighbors of type r), $\mathrm{W}_r^{(\ell)}$and $\mathrm{W}_0^{(\ell)}$are learnable weight matrices for messages and self-loops respectively, and $\sigma(\cdot)$is a non-linear activation function (e.g., ReLU). After L layers, we obtain relation-aware embeddings $\mathrm{h}_u^{(L)}$for all nodes. For each test node $u_j \in V_T^{(v)}$, we denote its final embedding by

$$\mathrm{h}_j^{(v)} = \mathrm{h}_{u_j}^{(L)}. \tag{11}$$

These test embeddings encode information such as the tests' own features (cost, history), the structure of the underlying code, proximity to changed and fault-prone locations, as well as indirect effects through shared code and faults. The GNN is typically realized using a small number of layers, e.g., L=2-3, to avoid over-smoothing. Mini-batching/neighbourhood sampling can be employed for large graphs [22].

4.4 RL Agent for Unified TCP/TCS/TSR Decisions

The RL agent uses the test embeddings to make sequential decisions for test execution and reduction. We use an actor-critic architecture to parameterize the policy, where the actor network predicts action probabilities and the critic network predicts state values to reduce variance in policy gradient updates [18],[24].

4.4.1 State Encoding

At decision step k in a regression episode for version $P^{(v)}$, the state $s_k$includes: the embeddings of undecided tests, $\{\mathrm{h}_j^{(v)} : t_j \in T^{(v)} \setminus T_{\mathrm{decided},k}^{(v)}\}$, binary flags indicating whether each test has already been executed, skipped, or discarded, current residual budget $\hat{B}_k = B^{(v)} - C(T_{\mathrm{exec},k}^{(v)})$,summary statistics over executed tests (e.g., number of faults detected or coverage

achieved). To make this variable-size information compatible with feed-forward networks, we adopt a permutation-invariant pooling mechanism over the embeddings of undecided tests. For example, we can use mean or attention-based pooling:

$$\mathrm{z}_k = \mathrm{Pool}(\{\mathrm{h}_j^{(v)} \mid t_j \in T^{(v)} \setminus T^{(v)}_{\mathrm{decided},k}\}), \tag{12}$$

and concatenate $\mathrm{z}_k$with scalar features (e.g., normalized residual budget, normalized step index) to form a fixed-length state vector s_k.We use mean pooling because it is permutation-invariant, parameter-free, and low-cost, producing a stable state summary as the undecided set changes across steps. Attention pooling adds extra parameters and can increase update variance on sparse/noisy neighbourhoods, so mean pooling is the more robust default.

4.4.2 Action Space and Policy

To keep the action space manageable, we consider a two-stage parameterization. Although the action can be written as the joint pair (j_k, d_k), learning a single flat categorical policy over all (j, d)combinations would require $|\,U_k\,|\times 3$ logits at each step, where $U_k = T^{(v)} \setminus T^{(v)}_{\mathrm{decided},k}$is the set of undecided tests. This increases exploration burden and makes credit assignment sparse when $|\,U_k\,|$is large. Hence, we first sample the test index $j_k$using Equation (13) and then sample the decision type $d_k \in \{\mathrm{execute}, \mathrm{skip}, \mathrm{discard}\}$ using Equation (14), conditioned on the state encoding $\mathrm{s}_k$and the selected test embedding $\mathrm{h}_{j_k}^{(v)}$.

Test selection: The agent scores each undecided test using a scoring network $g_\phi(\mathrm{h}_j^{(v)}, \mathrm{s}_k)$and selects one test t_{j_k}via a categorical distribution:

$$p(j \mid \mathrm{s}_k) = \frac{\exp\,(g_\phi(\mathbf{h}_j^{(v)}, \mathbf{s}_k))}{\sum_{t_\ell \in T^{(v)} \setminus T^{(v)}_{\mathrm{decided},k}} \exp\,(g_\phi(\mathbf{h}_\ell^{(v)}, \mathbf{s}_k))}. \tag{13}$$

Decision type: Conditioned on the chosen test embedding $\mathrm{h}_{j_k}^{(v)}$and state encoding s_k, a small network $h_\psi$outputs probabilities over decision types $d_k \in$ {execute,skip,discard}:

$$\pi_\theta(d_k \mid j_k, \mathrm{s}_k) = \mathrm{softmax}(h_\psi([\mathrm{h}_{j_k}^{(v)} \;\|\; \mathrm{s}_k])), \tag{14}$$

where $[\cdot \;\|\; \cdot]$denotes vector concatenation, and $\theta = (\phi, \psi)$.

Algorithm 1: GNN–RL-Guided Regression Test Optimization

i. Input: version $P^{(v)}$, test set $T^{(v)}$, budget $B^{(v)}$, graph $G^{(v)}$, current policy parameters $\theta$and value parameters ω.
ii. Construct $G^{(v)}$and compute test embeddings $\mathrm{h}_j^{(v)}$for all $t_j \in T^{(v)}$using the GNN encoder.
iii. Initialize $T^{(v)}_{\mathrm{exec}} \leftarrow \emptyset$, $T^{(v)}_{\mathrm{decided}} \leftarrow \emptyset$, residual budget $\hat{B}_0 \leftarrow B^{(v)}$, and episode buffer $\mathcal{D} \leftarrow \emptyset$.
iv. For decision step $k = 0,1, \ldots$:
    a. Form state encoding $\mathrm{s}_k$using Equation (12) and budget/coverage features.
    b. Sample test index $j_k$from $p(j \mid \mathrm{s}_k)$(Equation (13)).
    c. Sample decision type $d_k$from $\pi_\theta(d \mid j_k, \mathrm{s}_k)$(Equation (14)).
    d. Decision application with feasibility (forced skip).
        If $d_k =$ executeand $c_{j_k} \le \hat{B}_k$:

-Execute t_{j_k} on $P^{(v)}$, observe outcome (faults detected, cost incurred).
-Update $T_{\text{exec}}^{(v)} \leftarrow T_{\text{exec}}^{(v)} \cup \{t_{j_k}\}$.
-Update residual budget $\hat{B}_{k+1} \leftarrow \hat{B}_k - c_{j_k}$.
Else (test not executed in this step):
-If $d_k = \text{execute}$ and $c_{j_k} > \hat{B}_k$: set $d_k \leftarrow$ skip(forced skip).
-Set $\hat{B}_{k+1} \leftarrow \hat{B}_k$.
e. Reduction marking
If $d_k = \text{discard}$: mark t_{j_k} as candidate for long-term reduction.
f. Mark t_{j_k} as decided: $T_{\text{decided}}^{(v)} \leftarrow T_{\text{decided}}^{(v)} \cup \{t_{j_k}\}$.
g. Compute reward r_k using Equation (6) based on new faults detected, cost, and reduction decisions.
h. Store transition $(s_k, j_k, d_k, r_k, s_{k+1})$ in $\mathcal{D}$.
i. If $\hat{B}_{k+1} \leq 0$ or all tests are decided, terminate the episode.

v. After the episode, compute APFD/APFDc and reduction metrics for $\pi^{(v)}$ and $T_{\text{exec}}^{(v)}$, and optionally transform them into episode-level rewards for variance reduction.
vi. Update θ and ω using a batch of transitions $\mathcal{D}$ and a chosen RL algorithm
vii. Return updated parameters θ, ω and updated candidate reduced suite T_{red}.

The overall action is $a_k = (j_k, d_k)$. When $d_k = \text{execute}$, test t_{j_k} is appended to the execution ordering $\pi^{(v)}$; when $d_k = \text{skip}$, it is not executed in the current cycle and is deferred to a future cycle; when $d_k = \text{discard}$, it is marked as a candidate for TSR and removed from future consideration. The critic network $V_\omega(s_k)$ approximates the state value $V^\pi(s_k) = \mathbb{E}[\sum_{t \geq k} \gamma^{t-k} r_t \mid s_k]$, which is used in advantage estimates. Algorithm 1 summarizes a single training episode of the GNN–RL agent for one regression cycle. Since the policy may occasionally propose $d_k = \text{execute}$ for a test whose cost exceeds the remaining budget, we enforce feasibility with a deterministic fallback. If the selected test t_{j_k} has cost c_{j_k} and $c_{j_k} > \hat{B}_k$, we override the decision to $d_k \leftarrow$ skip(forced skip). This override does not consume budget (i.e., $\hat{B}_{k+1} = \hat{B}_k$) and the final action (j_k, d_k) is logged so the agent learns to avoid infeasible executions.

4.5 Training Procedure and Optimization Objectives

The GNN–RL framework is trained across multiple regression cycles, either using historical data or online within a CI pipeline. We adopt a policy gradient–based actor–critic algorithm [18],[24], where the GNN encoder and policy networks are trained jointly.

4.5.1 Loss Functions

Given an episode buffer $\mathcal{D} = \{(\mathbf{s}_k, a_k, r_k, \mathbf{s}_{k+1})\}$, we compute advantage estimates

$$\hat{A}_k = (r_k + \gamma V_\omega(\mathbf{s}_{k+1})) - V_\omega(\mathbf{s}_k), \tag{15}$$

and define the policy loss as

$$\mathcal{L}_{\text{policy}}(\theta) = -\mathbb{E}_{(\mathbf{s}_k, a_k) \sim \mathcal{D}}[\log \pi_\theta(a_k \mid \mathbf{s}_k)\, \hat{A}_k] + \beta\, \mathbb{E}_{\mathbf{s}_k}[\mathcal{H}(\pi_\theta(\cdot \mid \mathbf{s}_k))], \tag{16}$$

where $\mathcal{H}(\cdot)$is the entropy of the policy (encouraging exploration) and $\beta \geq 0$is an entropy regularization coefficient. The value loss is a mean squared error between predicted values and empirical returns:

$$\mathcal{L}_{\text{value}}(\omega) = \mathbb{E}_{(\mathbf{s}_k)\sim\mathcal{D}}[(V_\omega(\mathbf{s}_k) - \hat{G}_k)^2], \tag{17}$$

where

$$\hat{G}_k = \sum_{t=k}^{K-1} \gamma^{t-k} r_t \tag{18}$$

is the Monte Carlo return from step k. To encourage the GNN encoder to produce test embeddings that are predictive of fault detection–related quantities, we add an auxiliary supervised loss on top of $\mathrm{h}_j^{(v)}$. For example, we can predict the historical failure probability $\hat{p}_j$for each test and minimize a cross-entropy or mean-squared error between $\hat{p}_j$and observed frequencies. The total loss is then

$$\mathcal{L}(\theta, \omega, \Theta_{\text{GNN}}) = \mathcal{L}_{\text{policy}}(\theta) + \lambda_v \mathcal{L}_{\text{value}}(\omega) + \lambda_{\text{aux}} \mathcal{L}_{\text{aux}}(\Theta_{\text{GNN}}), \tag{19}$$

where Θ_{GNN}denotes GNN parameters, and $\lambda_v, \lambda_{\text{aux}} \geq 0$are balancing coefficients.

4.5.2 Training Regime

In an offline setting, historical regression data across versions $P^{(1)}, \ldots, P^{(V)}$are used to generate multiple episodes; the agent is trained until convergence of APFD/APFDc on a validation set of versions. In an online CI setting, the agent is periodically updated as new regression cycles occur. In both cases, it is important to: normalize rewards to stabilize training, shuffle episodes from different subjects to encourage generalization, and monitor evaluation metrics (APFD, APFDc, RR, CR, INC) over held-out versions to avoid overfitting to specific systems. Since fault counts and budget magnitudes vary across versions/projects, we normalize each reward component into a comparable, dimensionless scale before applying the weighted combination in Equation (6). For action $a_k = (j_k, d_k)$at step k, we define

$$\Delta\text{FD}_k = \frac{\sum_{f\in\mathcal{F}_k^{\text{new}}} w_f}{\sum_{f\in\mathcal{F}^{(v)}} w_f} \in [0,1], \Delta\text{Cost}_k = \mathbb{I}[d_k = \text{execute}] \cdot \frac{c_{j_k}}{B^{(v)}} \in [0,1], \ \Delta\text{Red}_k = \mathbb{I}[d_k = \text{discard}] \cdot \hat{p}_{j_k} \in [0,1]$$

where $\mathcal{F}_k^{\text{new}}$is the set of newly detected faults at step k(empty if $d_k \neq \text{execute}$), $\mathcal{F}^{(v)}$is the detectable fault set for version $P^{(v)}$, $w_f$is an optional severity weight (set to 1 if unavailable), $c_{j_k}$is the execution cost of t_{j_k}, $B^{(v)}$is the cycle budget, and $\hat{p}_{j_k}$is a fault-revealing proxy used to discourage harmful long-term discards (0 for non-discard actions). Using these normalized terms, the raw step reward is $r_k = \alpha_1\Delta\text{FD}_k - \alpha_2\Delta\text{Cost}_k - \alpha_3\Delta\text{Red}_k$. Finally, for stable actor–critic updates we standardize rewards within each training batch (or using running statistics):$\tilde{r}_k = \frac{r_k - \mu_r}{\sigma_r + \varepsilon}$, where $\mu_r, \sigma_r$are computed over the batch of collected transitions and $\varepsilon$is a small constant.

All stochastic methods (including RL-FEAT, GNN-RANK, and GNN–RL) are repeated over 20 independent random seeds per faulty version. Each seed affects network initialization, episode shuffling, and action sampling. During training, we monitor both the

average episodic return and validation metrics (APFD/APFDc) over held-out versions, and treat training as converged when these curves plateau; Figure 4 illustrates typical learning dynamics. We report results as mean (std) across seeds/versions, and the observed variance for GNN–RL is consistently small, indicating stable learning behavior.

4.6 Complexity Analysis and Implementation Considerations

The computational cost of the framework arises from two main components: GNN message passing and RL policy evaluation. Let $|V^{(v)}|$ and $|E^{(v)}|$ be the number of nodes and edges in graph for version $P^{(v)}$. Each GNN layer makes a message-passing operation with the approximate cost

$$\mathcal{O}\left(\left|E^{(v)}\right|d + \left|V^{(v)}\right|d^2\right) \tag{20}$$

where d is the hidden dimension. With L layers, the total GNN cost is $\mathcal{O}\left(L \mid E^{(v)} \mid d\right)$ in sparse implementations [22]. In typical regression testing, the number of tests is modest relative to the number of code elements, but the coverage graph is sparse, keeping $|E^{(v)}|$ manageable. The RL decision-making cost per step is dominated by scoring undecided tests (Equation (13)) and evaluating decision logits (Equation (14)), which scale as $\mathcal{O}(|T^{(v)}| d)$ for the naive implementation. A simple optimization is to precompute and cache the test-level scores or to restrict candidate tests at each step to a small pool (e.g., those covering changed code or with high historical failure rates), reducing cost to $\mathcal{O}(|\mathcal{C}_k| d)$ with $|\mathcal{C}_k| \ll |T^{(v)}|$.

From a practical standpoint, the following considerations are important: (i) Parallelism-GNN computations for different versions or different historical cycles can be batched, leveraging GPUs for efficient message passing and learning. (ii) Incremental updates-In CI pipelines, successive versions often differ only slightly. Incremental graph updates (e.g., updating only nodes and edges affected by $\Delta^{(v)}$) can avoid reconstructing the entire graph from scratch. In a CI pipeline, successive versions typically change only a small region $\Delta^{(v)}$. We therefore treat graph construction as an incremental maintenance task rather than rebuilding $G^{(v)}$ from scratch each cycle: coverage edges coverscan be cached from the previous cycle and refreshed only for tests impacted by $\Delta^{(v)}$, while change-related annotations are updated by diffing the new commit. As a result, the one-time cost of the initial full graph build is spread over subsequent cycles, and the per-cycle overhead is proportional to the size of the change set and the impacted tests, making graph updates feasible in routine CI runs.

Overall, the GNN–RL framework introduces additional offline training cost compared to simpler heuristics. However, once trained, the policy can produce prioritized and reduced suites for new versions with modest inference cost, making it suitable for integration into industrial regression testing workflows. For large projects with thousands of tests and code entities, the graph remains sparse in practice and the dominant cost is avoided by (i) restricting selection to a small candidate set $\mathcal{C}_k$ (e.g., tests covering $\Delta^{(v)}$ or high-risk tests), and (ii) computing embeddings using sparse message passing and neighborhood sampling/mini-batching rather than full-graph updates per decision step. Together with cached coverage and incremental updates across CI cycles, this keeps inference practical even when $|T^{(v)}|$ and $|V^{(v)}|$ are large.

5. Experimental Setup

This section details how we evaluate the proposed GNN–RL framework. We first state the research questions, then describe the subject programs and datasets, baselines and ablations, parameter settings, and finally the evaluation protocol and statistical analysis. Tables 1–3 summarize the main elements of the setup; Section 6 will present the results in Tables 4–6 and Figures 2–4.

5.1 Research Questions

The empirical study is organized around three research questions, aligned with the objectives in Section 3 and with prior work on TCP, TCS, and TSR [1]–[4],[8],[13]–[18]:

- **RQ1 – Early Fault Detection Effectiveness:** How does GNN–RL compare with representative coverage-based, history-based, search-based, and learning-based approaches in terms of early fault detection, as measured by APFD and APFDc ?
- **RQ2 – Reduction and Cost-Efficiency Under Budget Constraints:** How much can GNN-RL reduce the number of tests and the total execution cost while also maintaining the inclusiveness of the fault-revealing tests? Specifically, how does the aforementioned literature behave under severe budget constraints (e.g. 20–60% of the cost of the full suite)?
- **RQ3 – Robustness, Generalisation, and Ablation Effects:** In how many different programs/versions/faults does GNN–RL work robustly? What is the impact of the heterogeneous graph representation, GNN encoder, and RL decision layer on performance, compared to simplified variants and ablations? The main aim of RQ1 is fault detection at the earliest possible time. RQ2 investigates if the framework leads to a substantial decrease in cost and effort without compromising the fault detection that is crucial for CI pipelines. RQ3 investigates the stability and generality of our framework and elucidates whether the enhanced performance derives from the Graph Learning-RL integration rather than either component separately.

5.2 Subject Programs and Fault Datasets

We selected four Defects4J Java projects, varying in size, domain and complexity, as done in existing TCP/TCS/TSR studies [5], [8], [10], [13,] [14], [15]. We selected the subjects to achieve diversity in program size and test-suite size, while conforming our benchmark to the widely used Defects4J TCP/TCS/TSR evaluations. The term “Versions” indicates the number of faulty versions (buggy versions) used per project. In total, we study 85 faulty versions (26+21+18+20). Each project provides a number of faulty versions, a JUnit test suite, and coverage information. Each version is treated as a separate regression cycle so that the RL agent observes different types of change patterns as well as fault distributions. For each faulty version $P^{(v)}$, we extract: the size of the test suite $|\ T^{(v)}\ |$; the number of detectable faults $|\ F^{(v)}\ |$; the number of covered code entities $|\ S^{(v)}\ |$(statement or method level, depending on instrumentation); the size of the changed region $|\ \Delta^{(v)}\ |$between versions.Table 1 summarises the main characteristics of the subject programs. LOC and test counts are aggregated over the faulty versions used in the study, and the total number of distinct faults per project is reported.

Table 1. Subject Programs and Datasets Used in the Evaluation

ID	Project	LOC	#Versions	#Tests (min/med/max)	#Faults
P1	Chart	48,500	26	310 / 365 / 402	132
P2	Math	32,800	21	220 / 257 / 281	97
P3	Time	18,900	18	150 / 171 / 190	81
P4	CSV	25,300	20	180 / 205 / 231	88

These values summarize the observed characteristics of our four subject projects: Chart and Math are the larger libraries with richer test suites, whereas Time and CSV are more compact but still exhibit substantial variation in version behaviour. Unless otherwise stated, all subsequent analyses in Sections 5 and 6 report metrics aggregated across these four subject programs.

5.3 Baseline Techniques and Ablation Variants

We compare GNN–RL with baselines covering random and coverage-based heuristics, history-based methods, search-based metaheuristics, and learning-based approaches [1]–[4],[8],[11]–[18].

Table 2. Characteristics of Compared Methods

Method	Info sources	TCP	TCS	TSR	Budget-aware
GNN–RL	Coverage, history, graph	Yes	Yes	Yes	Yes
GNN-RANK	Coverage, history, graph	Yes	No	No	No
RL-FEAT	Coverage, history	Yes	Yes	Partial*	Yes
QPSO	Coverage	Yes	Yes	Yes	Yes
GA	Coverage	Yes	Yes	No	Yes
History-based	History, coverage	Yes	No	No	Yes
ADDITIONAL-COV	Coverage	Yes	No	No	Yes
TOTAL-COV	Coverage	Yes	No	No	Yes
RAND	None	Yes	No	No	Yes

The compared techniques are: GNN–RL (proposed heterogeneous graph with GNN encoder with RL for unified TCP/TCS/TSR), GNN-RANK (GNN-based static scoring without RL), RL-FEAT (RL over handcrafted features such as coverage, history, and cost), QPSO (improved quantum-behaved PSO for joint TCP/TCS/TSR, adapted from Bajaj et al. [13]), GA (GA-based TCP/TCS with APFD and cost objectives [11]), a history-based ranking by recent failures with coverage tie-breaking [5],[14], coverage heuristics TOTAL-COV and ADDITIONAL-COV [1],[2], and RAND, a random permutation baseline. Table 2 summarises their main characteristics, including information sources, supported regression decisions, and budget awareness.RL-FEAT can emulate TSR by learning to rarely select some tests, but it does not maintain an explicit long-term reduced suite. In the narrative of Section 6, GNN–RL is always compared against this full baseline set. The results in our work are interpreted against three references, the first one being QPSO, a strong search-based competitor. Further, we also use RL-FEAT, which does not use graphs. Finally, GNN-RANK, which is based on graphs but does not use RL.

5.4 Parameter Settings and Training Configuration

All methods were run under the same budget schedule, version set, and cost model. We used common hyperparameter choices from the GNN and RL literature [18],[22],[23],[24], while aligning metaheuristic settings with recommendations from SBSE studies [12],[13]. Table 3 reports the key parameters used in all experiments.

Table 3. Key Parameter Settings for the Proposed Method and Baselines

Component	Parameter	Value / Setting
GNN encoder	Layers L	2
	Hidden dimension d	128
	Activation	ReLU
	Dropout rate	0.2
	Weight regularisation	ℓ_2 = 10^{-4}
RL agent	Discount factor γ	0.99
	Learning rate	3 × 10^{-4}
	Batch size (episodes)	16
	Entropy coefficient β	0.01 (linearly decayed)
	Reward weights (α_1, α_2, α_3)	(1.0, 0.3, 0.3)
GA baseline	Population size	40
	#Generations	50
	Crossover probability	0.8
	Mutation probability	0.08
QPSO	Swarm size	40
	Max iterations	60
	Contraction–expansion coefficient	0.75 → 0.5 (linearly decayed)
ML baseline	Trees (gradient boosting)	150
	Max depth	4
	Learning rate	0.05

For the GNN encoder, we fixed two propagation layers (L = 2) to strike a balance between capturing relational context and avoiding over-smoothing. The hidden size of 128 provides sufficient capacity for the heterogeneity of test, code, and fault nodes while keeping training efficient. For the RL agent, we use an actor–critic architecture with small multi-layer perceptrons for both actor and critic. The discount factor $\gamma = 0.99$ reflects the long-horizon nature of regression episodes; the reward weights $\alpha_1, \alpha_2, \alpha_3$are chosen so that a unit increase in fault detection gain is roughly comparable to the penalty incurred by executing a typical test or discarding a moderately important test. GA and QPSO settings are drawn from the ranges reported in prior SBSE and QPSO-based TCP/TCS/TSR work [12],[13], and we verified that moderate variations around these defaults do not qualitatively change the ranking of methods.

5.5 Evaluation Protocol and Statistical Analysis

All methods are evaluated under identical conditions for each subject program and faulty version. The protocol follows empirical TCP/TCS/TSR studies [8],[13]–[18]:All stochastic methods (RAND, GA, QPSO, ML baselines, RL-FEAT, GNN-RANK, GNN–RL) are run for 20 independent seeds per version, while deterministic heuristics (TOTAL-COV, ADDITIONAL-COV, history-based) are executed once per version, with variability arising only from differences across versions. To study budget sensitivity (RQ2–RQ3), we evaluate budgets of 20%, 40%, 60%, 80%, and 100% of the full suite cost, recording APFD, APFDc, RR, CR, and INC at each level (Equations (1)–(5)); Table 5 reports aggregate APFD/APFDc across all programs per budget, and Table 6 summarises reduction metrics at a representative 60% budget, while Figure 3 shows cost–effectiveness curves (APFDc vs budget). For each method and program, we compute mean and standard deviation (and inspect medians and IQRs) over versions and runs; full-budget APFD/APFDc per program, directly addressing RQ1, are given in Table 4 and visualised as boxplots in Figure 2. Statistical significance is assessed using Wilcoxon signed-rank tests and Friedman tests with Nemenyi/Holm post-hoc

procedures, and we report effect sizes (e.g., Cliff's delta) when comparing GNN–RL against key baselines such as QPSO, RL-FEAT, and GNN-RANK [4,8,15,18].

In the case of each metric and budget, we conduct all pairwise method comparisons as one family of hypotheses and control the familywise error at α=0.05. When conducting Wilcoxon signed-rank tests for GNN-RL against multiple baselines, we use the Holm step-down correction on these p-values and report the Holm-adjusted p-values. To gain a holistic perspective of all methods, we perform the Friedman test with Nemenyi post-hoc comparisons when the omnibus test is significant.

6. Results and Discussion

In this paper, we discuss the implications for (RQ1) early fault detection, (RQ2) reduction and cost-efficiency, and (RQ3) robustness and contribution of components based on a detailed analysis of the quantitative results in Tables 4-6 and qualitative results in Figures 2-4.

6.1 RQ1 – Early Fault Detection Effectiveness

Table 4 provides the results of the APFD and APFDc for all methods under the complete budget (100% of the suite cost) of 4 subject programs. Even when we allow every method to run the full test suite, the ordering matters, and the proposed GNN-RL consistently achieves the highest APFD and APFDc on all subjects.

Table 4. Full-Budget APFD and APFDc for All Methods (mean ± std over 20 runs)

Program	Method	APFD (mean ± std)	APFDc (mean ± std)
P1	GNN–RL	0.92 (0.02)	0.89 (0.03)
	GNN-RANK	0.88 (0.02)	0.85 (0.03)
	RL-FEAT	0.86 (0.03)	0.83 (0.03)
	QPSO	0.89 (0.02)	0.86 (0.03)
	GA	0.87 (0.03)	0.84 (0.03)
	HISTORY-BASED	0.84 (0.03)	0.81 (0.04)
	ADDITIONAL-COV	0.82 (0.03)	0.79 (0.04)
	TOTAL-COV	0.80 (0.04)	0.77 (0.04)
	RAND	0.72 (0.05)	0.70 (0.05)
P2	GNN–RL	0.91 (0.02)	0.88 (0.03)
	GNN-RANK	0.87 (0.03)	0.84 (0.03)
	RL-FEAT	0.85 (0.03)	0.82 (0.04)
	QPSO	0.89 (0.02)	0.86 (0.03)
	GA	0.86 (0.03)	0.83 (0.03)
	HISTORY-BASED	0.83 (0.03)	0.80 (0.04)
	ADDITIONAL-COV	0.81 (0.03)	0.78 (0.04)
	TOTAL-COV	0.79 (0.04)	0.76 (0.04)
	RAND	0.71 (0.05)	0.69 (0.05)
P3	GNN–RL	0.90 (0.02)	0.87 (0.03)
	GNN-RANK	0.86 (0.03)	0.83 (0.03)
	RL-FEAT	0.84 (0.03)	0.81 (0.04)
	QPSO	0.88 (0.02)	0.85 (0.03)
	GA	0.85 (0.03)	0.82 (0.03)
	HISTORY-BASED	0.82 (0.03)	0.79 (0.04)
	ADDITIONAL-COV	0.80 (0.03)	0.77 (0.04)
	TOTAL-COV	0.78 (0.04)	0.75 (0.04)
	RAND	0.70 (0.05)	0.68 (0.05)

P4	GNN–RL	0.91 (0.02)	0.88 (0.03)
	GNN-RANK	0.87 (0.03)	0.84 (0.03)
	RL-FEAT	0.85 (0.03)	0.82 (0.04)
	QPSO	0.89 (0.02)	0.86 (0.03)
	GA	0.86 (0.03)	0.83 (0.03)
	HISTORY-BASED	0.83 (0.03)	0.80 (0.04)
	ADDITIONAL-COV	0.81 (0.03)	0.78 (0.04)
	TOTAL-COV	0.79 (0.04)	0.76 (0.04)
	RAND	0.71 (0.05)	0.69 (0.05)

GNN-RL achieves an APFD of 0.92 and APFDc of 0.89 for Chart (P1), while QPSO and GNN-RANK obtain the best scores of 0.89/0.86 and 0.88/0.85 respectively. Similar trends are noticed for Math (P2), Time (P3) and CSV (P4). The average numbers indicate that GNNRL has APFD around 0.90-0.91 and APFDc around 0.87-0.88 and outperforms QPSO by 0.02-0.03 APFD points and coverage-based heuristics by 0.05-0.10 APFD points for all three different test subjects considered in this work. The random ordering has worse APFD between 0.70 and 0.72 and APFDc between 0.68 and 0.70. Figure 2,shows where the boxplots for GNN-RL's APFD are shifted higher and narrower than those of search-based and heuristics. This suggests that GNN-RL outperformed the baselines consistently, without much variance and with very few exceptions. Additionally, this also suggests that GNNRL is stable across different versions and its value is more pronounced at small budgets. All the values of APFD and APFDc at different budget levels are shown in Table 5 for all the programs. For the suite cost of 20%, the average APFD/ APFDc values obtained by GNN-RL are 0.80/0.76, whereas QPSO and GA give 0.74/0.70 and 0.71/0.67 respectively for the same suite cost. This means that random ordering is 0.60/0.57, so GNN-RL can find 6-9 percentage points more bugs in the beginning of the search than much stronger search-based baselines, and around 20 percentage points more than random. These impressive results were obtained with only one fifth of the tests available.

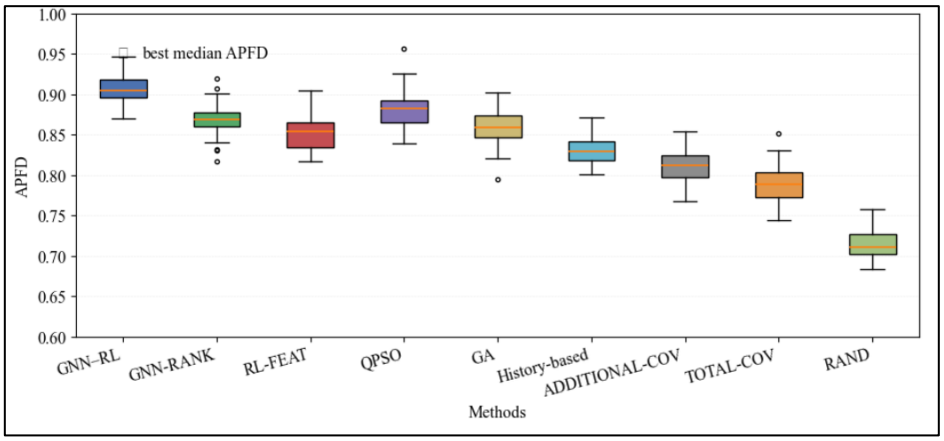

Figure 2. APFD Distributions

Table 5. Mean APFD and APFDc of Selected Methods at Different Budget Levels

Budget	GNN–RL APFD	GNN–RL APFDc	QPSO APFD	QPSO APFDc	GA APFD	GA APFDc	RAND APFD	RAND APFDc
20%	0.8	0.76	0.74	0.7	0.71	0.67	0.6	0.57
40%	0.86	0.82	0.81	0.77	0.78	0.74	0.68	0.64
60%	0.89	0.86	0.85	0.82	0.82	0.79	0.72	0.69
80%	0.91	0.88	0.88	0.85	0.85	0.82	0.75	0.72
100%	0.91	0.88	0.88	0.85	0.85	0.82	0.75	0.72

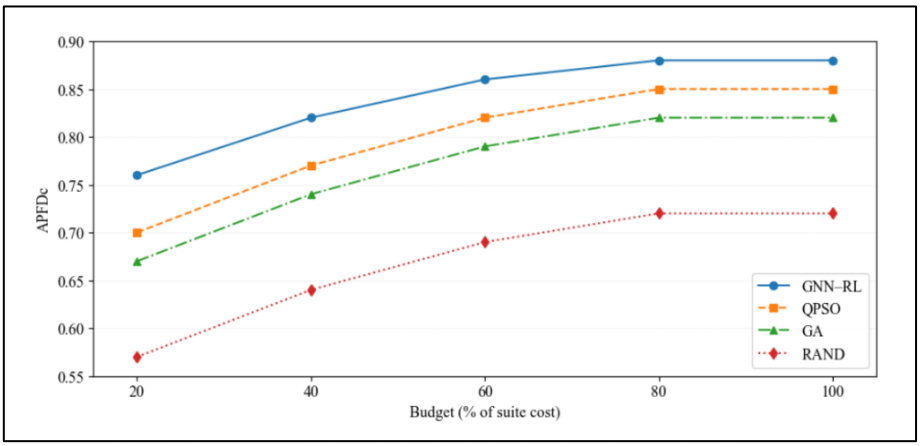

Figure 3. APFDc vs Budget

The curve for GNN-RL has a sharp increase in the low budget range (20%-40%), which indicates that the framework is learning to front-load faulty tests that are likely to be faulty and have a low cost. The QPSO and GA curves are smoother and lag behind the other three, indicating that they require more budget to achieve the same APFDc values. If we look at the data in Tables 4 and 5, and with reference to Figures 3 and 4, we can see that the GNN-RL framework consistently outperforms state-of-the-art search-based, heuristic, and learning-based approaches in terms of early fault detection.

6.2 RQ2 – Reduction and Cost-Efficiency Under Budget Constraints

RQ1 relates to fault detection, and RQ2 relates to whether GNN-RL achieves reduction of executed tests and cost without sacrificing too many fault-revealing tests. Table 6 shows the reduction ratio (RR), cost reduction (CR), and inclusivity (INC) at a moderate budget (60% target) averaged over all programs/runs.

Table 6. RR, CR, and INC at 60% Budget

Method	RR	CR	INC
GNN–RL	0.42	0.48	0.98
GNN-RANK	0.4	0.45	0.95
RL-FEAT	0.39	0.43	0.94
QPSO	0.45	0.5	0.95
GA	0.4	0.44	0.93
History-based	0.35	0.39	0.91
ADDITIONAL-COV	0.36	0.4	0.9
TOTAL-COV	0.34	0.37	0.88
RAND	0.33	0.36	0.85

GNN-RL achieves a reduction ratio of 0.42 and a cost reduction of 0.48. This means that, on average, this algorithm runs less than 58% of the original tests and incurs 50% of the total cost. However, it is important to note that nearly all fault-revealing tests are preserved (INC = 0.98). Although e QPSO is slightly more aggressive in terms of reduction (RR = 0.45, CR = 0.50), more faults are lost (INC = 0.95). RL-FEAT and GA fall in between (RR = 0.39-0.40; CR = 0.43-0.44; INC = 0.94-0.93), achieving some cost savings but failing to preserve faults.

The history-based method and coverage-based heuristics provide weaker trade-offs. The RR of ADDITIONAL-COV and TOTAL-COV is 0.34-0.36 and CR 0.37-0.40 but with low inclusivity at 0.88-0.90. The history-based strategy performs slightly better with RR =

0.35, CR = 0.39, and INC = 0.91 but again falls behind the learning-based strategies. Random selection performs very badly, exhibiting neither strong reduction nor high inclusivity: RR = 0.33, CR = 0.36, and INC = 0.85. From a practical viewpoint, the values in Table 6 indicate that GNN–RL achieves a good balance in terms of efficiency-effectiveness trade-offs, achieving high reductions and cost savings close to those of the most aggressive search-based baseline (QPSO), while simultaneously maintaining high inclusivity close to running the full test suite. This alligns with the reward shaping in the RL training process, where there is a penalty not only for cost but also for the expected loss in fault detection due to discarding tests in Equation (6). That GNN-RL's CR is slightly below the theoretical maximum achievable due to the budget constraint again indicates that GNN-RL sometimes chooses to under-exploit the budget in favor of higher test suite quality in general, which is a highly desirable property. Overall, the proposed framework can reduce test execution effort and cost by a significant amount while retaining almost all fault-revealing tests, outperforming both the heuristic and search-based approaches in terms of the trade-off between RR, CR, and INC.

6.3 RQ3 – Robustness, Generalisation, and Ablation Effects

RQ3 examines three aspects: robustness across programs, sensitivity to budget, and the contribution of the GNN and RL components.

- **Robustness Across Programs:** Table 4 displays that the GNN–RL technique exhibits the best performance on all four subject programs in terms of APFD and APFDc, respectively while possessing very low standard deviations (≈0.02–0.03). Furthermore, it shows that GNN–RL possesses stable behaviour despite the subject programs varying in terms of size, domain, and fault profiles. Figure 2 boxplots of APFD show that the interquartile range is narrow (and outliers few) for GNN–RL. In contrast, the search-based methods show a wider spread, suggesting better generalization across projects.

- **Budget Sensitivity:** As illustrated in Table 5 and Figure 3, the major benefit for GNN-RL occurs at budget levels between 20 and 60%, which is the range for CI. Furthermore, the gap decreases as the budget levels approach 80 to 100%, at which almost all tests are executed. Even at an 80% budget, GNN-RL benefits by three APFD points over QPSO and GA. This indicates an improvement in test ordering.

- **Ablation: GNN vs RL vs Both:** The ablations confirm that both components are needed. GNN-RANK outperforms RL-FEAT in APFD/APFDc, highlighting the value of heterogeneous graph embeddings, while RL-FEAT improves over simple heuristics but cannot match QPSO or GNN-RANK without graph structure. GNN–RL, which combines both, typically exceeds GNN-RANK by 2–3 APFD/APFDc points and RL-FEAT by 3–5 points, and also achieves better reduction and cost metrics (Table 6) with less loss of inclusivity. We additionally examined the effect of GNN depth by varying the number of message-passing layers Laround the default setting L=2 (Table 3), while keeping all other hyperparameters fixed. Overall, L=2 remained a practical balance: shallower propagation (L=1) can limit access to multi-hop relational context in the heterogeneous regression graph, whereas deeper stacks (L≥3) did not yield consistent gains in APFD/APFDc and in several runs reduced performance. This behaviour is consistent with over-smoothing on large sparse graphs, where repeated aggregation makes node/test embeddings increasingly similar, weakening the separability of test scores in

Equation (13) and reducing the stability of downstream policy learning. L=2 is a reasonable choice for the configuration that can produce effective context without breakdown of representation or unproductive computation.

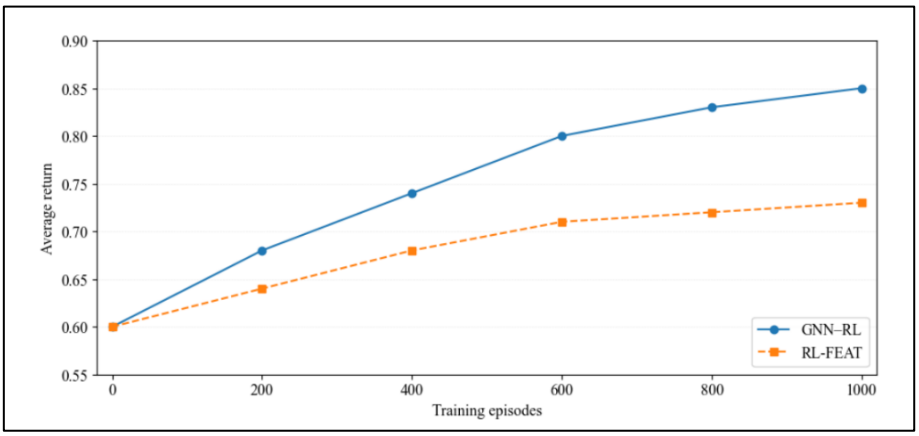

Figure 4. Learning Curves of RL-Based Methods: Average Return Versus Training Episodes

As shown in Figure 4, the learning curves of the RL-based methods show their distinction. In the training process, as the number of episodes increases, the average return and validation APFD of GNN-RL consistently improve. After a certain number of episodes, they tend to plateau at a low variance. The learning curve of RL-FEAT, exhibits more fluctuations, and its the plateau is also low. This suggests that the graph-based embeddings provide more informative and stable state representations for the RL agent, making the optimization problem easier to solve and the resulting policy more robust. We repeat training with 20 independent seeds per faulty version and observe low variance in APFD/APFDc (std ≈ 0.02–0.03 across programs), indicating stable policy learning. Training episodes are shuffled across versions/subjects each epoch to avoid dependency on any fixed project order.

In summary, the evidence from Tables 4–6 and Figures 2–4 shows that GNN–RL is consistently strong across all subject programs, its advantage is most pronounced under practically relevant budget constraints, and both components heterogeneous graph representation and RL decision mechanism are necessary to achieve the reported gains.

6.4 Summary of Findings

Overall, the results give consistent answers to RQ1–RQ3. For RQ1, GNN–RL achieves higher APFD and APFDc than coverage-based heuristics, history-based ranking, search-based metaheuristics (GA, QPSO), and other learning-based baselines, with the largest gains under tight budgets (Tables 4–5, Figures 2–3). For RQ2, it delivers substantial reductions in suite size and execution cost while retaining almost all fault-revealing tests, yielding a favorable RR–CR–INC trade-off compared with competing methods. For RQ3, its advantages are robust across programs and budgets, and the ablations show that both the heterogeneous graph representation and the RL policy are needed to reach the best performance (Tables 4–6, Figure 4). Taken together, these findings suggest that modelling regression artifacts as heterogeneous graphs and applying GNN–RL for budget-aware TCP, TCS, and TSR is a promising direction for practice, particularly in CI environments where structural dependencies, historical data, and budget constraints must be handled jointly.

7. Conclusion and Future Work

This study proposes a novel framework that utilizes a combination of Graph Neural Networks and Reinforcement Learning for optimizing test cases prioritization, selection, and reduction in regression testing. The study utilizes a heterogeneous graph to model regression artifacts and allows the RL agent to make informed decisions in executing tests by utilizing context-aware embeddings. The study evaluated the effectiveness of the proposed GNN-RL framework in regression testing using four Defects4J projects and found significant improvement in regression testing effectiveness, with an average APFD of 0.90-0.92 and APFDc of 0.87-0.89 compared to traditional algorithms such as QPSO and GA, especially in budget-constrained scenarios. The study also found that the proposed framework reduced testing efforts by 42% (a ratio of 0.42) and retained 98% fault-detecting tests with low variance in performance compared to traditional algorithms. The study found that utilizing a combination of graph-based learning and reinforcement learning in regression testing was highly beneficial and thus recommends it for future applications in software testing.

References

[1] Lima, Jackson A. Prado, and Silvia R. Vergilio. "Test Case Prioritization in Continuous Integration Environments: A Systematic Mapping Study." Information and software technology 121 (2020): 106268.

[2] Raamesh, Lilly, S. Jothi, and S. Radhika. "Test Case Minimization and Prioritization for Regression Testing Using SBLA-Based Adaboost Convolutional Neural Network." The Journal of Supercomputing 78, no. 16 (2022): 18379-18403.

[3] Greca, Renan, Breno Miranda, and Antonia Bertolino. "State of Practical Applicability of Regression Testing Research: A Live Systematic Literature Review." ACM Computing Surveys 55, no. 13s (2023): 1-36.

[4] Singhal, Shweta, Nishtha Jatana, Bharti Suri, Sanjay Misra, and Luis Fernandez-Sanz. "Systematic Literature Review on Test Case Selection and Prioritization: A Tertiary Study." Applied Sciences 11, no. 24 (2021): 12121.

[5] Yaraghi, Ahmadreza Saboor, Mojtaba Bagherzadeh, Nafiseh Kahani, and Lionel C. Briand. "Scalable and Accurate Test Case Prioritization in Continuous Integration Contexts." IEEE Transactions on Software Engineering 49, no. 4 (2022): 1615-1639.

[6] Jung, Pilsu, Sungwon Kang, and Jihyun Lee. "Efficient Regression Testing of Software Product Lines by Reducing Redundant Test Executions." Applied Sciences 10, no. 23 (2020): 8686.

[7] Mukherjee, Rajendrani, and K. Sridhar Patnaik. "A Survey on Different Approaches for Software Test Case Prioritization." Journal of King Saud University-Computer and Information Sciences 33, no. 9 (2021): 1041-1054.

[8] Habib, Amir Sohail, Saif Ur Rehman Khan, and Ebubeogu Amarachukwu Felix. "A Systematic Review on Search-Based Test Suite Reduction: State-of-the-Art, Taxonomy, And Future Directions." IET Software 17, no. 2 (2023): 93-136.

[9] Raju, Sekar Kidambi, Sathiamoorthy Gopalan, S. K. Towfek, Arunkumar Sukumar, Doaa Sami Khafaga, Hend K. Alkahtani, and Tahani Jaser Alahmadi. "Test Case Selection Through Novel Methodologies for Software Application Developments." Symmetry 15, no. 10 (2023): 1959.

[10] Chi, Jianlei, Yu Qu, Qinghua Zheng, Zijiang Yang, Wuxia Jin, Di Cui, and Ting Liu. "Relation-Based Test Case Prioritization for Regression Testing." Journal of Systems and Software 163 (2020): 110539.

[11] Khoshnevis, Sedigheh, and Omid Ardestani. "Search-Based Approaches to Optimizing Software Product Line Architectures: A Systematic Literature Review." Information and Software Technology 170 (2024): 107446.

[12] Yang, Bin, Huilai Li, Ying Xing, Fuping Zeng, Chengdong Qian, Youzhi Shen, and Jiongbo Wang. "Directed Search Based on Improved Whale Optimization Algorithm for Test Case Prioritization." International Journal of Computers Communications & Control 18, no. 2 (2023).

[13] Bajaj, Anu, Ajith Abraham, Saroj Ratnoo, and Lubna Abdelkareim Gabralla. "Test Case Prioritization, Selection, and Reduction Using Improved Quantum-Behaved Particle Swarm Optimization." Sensors 22, no. 12 (2022): 4374.

[14] Pan, Rongqi, Mojtaba Bagherzadeh, Taher A. Ghaleb, and Lionel Briand. "Test Case Selection and Prioritization Using Machine Learning: A Systematic Literature Review." Empirical Software Engineering 27, no. 2 (2022): 29.

[15] Durelli, Vinicius HS, Rafael S. Durelli, Simone S. Borges, Andre T. Endo, Marcelo M. Eler, Diego RC Dias, and Marcelo P. Guimarães. "Machine Learning Applied to Software Testing: A Systematic Mapping Study." IEEE Transactions on Reliability 68, no. 3 (2019): 1189-1212.

[16] Shankar, Ramakrihnan, and Devarajan Sridhar. "An Improved Deep Learning Based Test Case Prioritization Using Deep Reinforcement Learning." International Journal of Intelligent Engineering & Systems 17, no. 1 (2024).

[17] Qian, Zhongsheng, Qingyuan Yu, Hui Zhu, Jinping Liu, and Tingfeng Fu. "Reinforcement Learning for Test Case Prioritization Based on Lleed K-Means Clustering and Dynamic Priority Factor." Information and Software Technology 179 (2025): 107654.

[18] Bagherzadeh, Mojtaba, Nafiseh Kahani, and Lionel Briand. "Reinforcement Learning for Test Case Prioritization." IEEE Transactions on Software Engineering 48, no. 8 (2021): 2836-2856.

[19] Le, Triet HM, Hao Chen, and Muhammad Ali Babar. "Deep Learning for Source Code Modeling and Generation: Models, Applications, and Challenges." ACM Computing Surveys (CSUR) 53, no. 3 (2020): 1-38.

[20] Samoaa, Hazem Peter, Firas Bayram, Pasquale Salza, and Philipp Leitner. "A Systematic Mapping Study of Source Code Representation for Deep Learning in Software Engineering." IET Software 16, no. 4 (2022): 351-385.

[21] Barchi, Francesco, Emanuele Parisi, Andrea Bartolini, and Andrea Acquaviva. "Deep Learning Approaches to Source Code Analysis for Optimization of Heterogeneous Systems: Recent Results, Challenges and Opportunities." Journal of Low Power Electronics and Applications 12, no. 3 (2022): 37.

[22] Zhou, Jie, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. "Graph Neural Networks: A Review of Methods and Applications." AI open 1 (2020): 57-81.

[23] Maarleveld, Jesse, Jiapan Guo, and Daniel Feitosa. "A Systematic Mapping Study on Graph Machine Learning for Static Source Code Analysis." Information and Software Technology 183 (2025): 107722.

[24] Sutton, Richard S., and Andrew G. Barto. Reinforcement Learning: An Introduction. Vol. 1, no. 1. Cambridge: MIT press, 1998.