Journal of Intelligent Systems and Internet of Things
Vol. 18, No. 01, PP. 194-206, 2026
DOI: https://doi.org/10.54216/JISIoT.180114

An Adaptive Mutation-Aware Test Case Ordering Framework Using Deep Learning and Quantum-Behaved Multi-Objective PSO

S. Sowmyadevi[1,*], Anna Alphy[1]
[1]Department of CSE, SRMIST, Delhi-NCR campus, Ghaziabad, 201204, Uttar Pradesh, India
Emails: ss2860@srmist.edu.in; annaa@srmist.edu.in

Abstract

In regression testing, rapidly identifying defects is crucial for maintaining software quality amid frequent code changes. Traditional test case ordering methods, despite extensive research, often overlook the subtle but important relationship between test executions and mutations introduced during code modifications. This paper presents an adaptive mutation-aware test case ordering framework that integrates predictive modeling with swarm-based multi-objective optimization to address this gap. The approach begins by transforming test cases into enriched feature vectors, incorporating mutation coverage, historical performance, execution cost, and statement-level weighting. A supervised deep learning model is employed to predict the likelihood of each test case uncovering seeded defects. These predictions are subsequently fed into a Quantum-Behaved Particle Swarm Optimization (QPSO) engine, which generates an optimal execution sequence by jointly optimizing fault detection, execution cost, reuse potential, and coverage diversity. The proposed framework is demonstrated using a simple Java program and rigorously validated on real-world projects from the De-fects4J benchmark. Experimental results consistently show improvements in APFD, mutation scores, and execution efficiency, confirming the feasibility and scalability of the proposed system.

Received: March 16, 2025 Revised: June 09, 2025 Accepted: July 14, 2025

Keywords: Secure honeynet, Cloud IoT model; Machine learning algorithms; Health monitoring; Particle colony optimization

1. Introduction

Regression testing plays a critical role in software reliability under-changing versions. Frequent software changes within agile development iterations boost the chances of undesired side effects considerably. Preservation of current functions requires running perfectly selected and filtered test cases. Time and resource requirements of performing a full test suite with each update may be astronomically high, even for complex systems. Therefore, it is more crucial than ever to order fault-exposing tests early within the execution order with the help of advanced test case ordering algorithms.

Traditional test case ordering methods have utilized code coverage, past failure data, and randomly generated permutations but often fail to capture the strong relationship between tests and the kinds of faults that can occur in modified code. It is through the introduction of artificial faults (mutants) into the software to mimic real faults (mutation testing) that a richer, more defect-oriented view of test performance is gained. Combining mutation- based information with the ordering allows for a more penetrating exploration of interactions between tests and code anomalies. Nonetheless, mutation-aware methods are typically static and computationally intensive, often without the facility to adapt to the dynamic nature of change under evolving test environments.

This work introduces a mutation-aware test case ordering scheme with predictive learning along with a swarm-based optimization scheme that overcomes these constraints. It begins by transforming raw test cases into enriched feature vectors with the aid of mutation coverage, execution cost, past performance, and statement- level sensitivity. It utilizes

deep learning to predict fault-detection efficacy of test cases with regard to these features. It uses the predicted probabilities to input a Quantum-Behaved Particle Swarm Optimization (QPSO) algorithm, which delivers a most effective ordering with a tradeoff of various objectives of early fault detection, execution cost, past reliability, as well as diversity of code coverage.

The research is novel through its synergy of deep learning with quantum-inspired search as well as its ability to adaptively update test ordering algorithms as software changes its faults. The approach is first shown through a succinct Java program, with a straightforward depiction of each step, and then evaluated with real software from the Defects4J benchmark suite. Results show the method significantly improves early fault identification and reduces total testing cost compared to existing methodologies.

This method combines mutation-based testing, data-driven fault identification, and multi- objective search, offering a scalable, adaptive solution to a very computationally intensive problem of modern regression testing.

2. Related Work

Test case prioritization (TCP) is a rich research topic within regression testing, aiming to reorder test cases so that those with the greatest fault-revealing potential are executed first. Initially, methods were predominantly coverage-based, relying on metrics such as statement and branch coverage [1, 2]. Rothermel et al. [3] formally introduced TCP and established fundamental empirical investigations, illustrating how ordering impacts fault detection rate. Although such heuristics are time-efficient and simple to apply, they often result in suboptimal prioritization when code changes are semantically minor or dispersed.

To tackle these limitations, researchers explored history-based techniques, using previous test execution results—such as failure frequency, execution traces, or failure-predicting heuristics—to reorder test cases [4, 5]. Do et al. [6] demonstrated that combining historical failure data with coverage metrics significantly enhances prioritization. However, these methods degrade in performance when test behavior changes across versions or when historical data is noisy or sparse.

Mutation testing, introduced by DeMillo et al. [7], offers a fault-sensitive alternative. It works by generating small syntactic changes (mutants) in the source code to represent real faults. Test cases are evaluated on their ability to "kill" these mutants, yielding mutation scores that had better reflect fault detection potential. Greedy mutation-aware prioritization [8, 9] has proven competitive against coverage-based methods. Nevertheless, mutation analysis is computationally costly and requires careful tuning to balance overhead and quality. Selective mutation [10], mutant sampling [11], and operator-based reduction [12] have been proposed to mitigate these costs.

As mutation testing gained traction, researchers began integrating it with structural coverage and historical effectiveness [13, 14]. Gopinath et al. [15] combined mutation and coverage data to improve prioritization accuracy, while Zhang et al. [16] proposed mutant clustering to enhance diversity in mutant detection. Despite these improvements, mutation- aware techniques largely remained static, often relying on greedy selection mechanisms or manual interpretation.

The rise of machine learning has brought significant innovation to TCP. Supervised models have been used to learn fault-revealing patterns from static, historical, and dynamic features [17, 18]. Neural networks and tree-based classifiers predict test case failure probabilities, with some models incorporating semantic change analysis [19]. Deep learning models, particularly feedforward and recurrent architectures, have been employed to capture non- linear patterns in test behavior [20, 21]. However, most models optimize solely for fault probability, neglecting factors like execution cost, redundancy, and test reuse history.

To address multi-objective concerns, metaheuristic optimization algorithms have been applied. Genetic Algorithms (GA) [22], Ant Colony Optimization (ACO) [23], and Particle Swarm Optimization (PSO) [24] have all been utilized to reorder test suites by handling conflicting objectives such as fault detection versus execution time. Quantum-Behaved Particle Swarm Optimization (QPSO), introduced by Sun et al. [25], has recently demonstrated superior convergence speed and exploration capacity, although its application to regression testing remains nascent.

Hybrid frameworks that couple predictive models with optimization backends have emerged as powerful alternatives. For instance, Sharma et al. [26] used neural network-based fault predictors in tandem with PSO for test case prioritization. Others have proposed cost-aware reinforcement learning [27] and ensemble learning approaches with multi-objective fitness evaluation. Although effective, these hybrid methods often require extensive training data or handcrafted features.

Many of these approaches have been benchmarked on datasets like Defects4J, which provides real-world Java programs with seeded faults and validated test suites. Studies using Defects4J have established comparative baselines for APFD, APFDc, mutation score, and execution cost, making it an indispensable resource for TCP evaluation.

Building on these developments, this paper proposes a novel framework that integrates mutation-based static and dynamic features, deep learning-based fault detection prediction, and QPSO-based multi-objective optimization. Unlike prior methods, this approach dynamically adapts to changes in test behavior and mutation patterns, balances multiple evaluation metrics, and scales from illustrative case studies to industrial-scale benchmarks.

3. Background on QPSO

Quantum-Behaved Particle Swarm Optimization (QPSO) is a more advanced version of Particle Swarm Optimization (PSO), a nature-inspired metaheuristic that imitates the collective behavior of social animals like birds and fish. Traditional PSO algorithms rely on the movement of particles through a search space where each particle updates its position according to its personal experiences as well as those of its immediate neighborhoods. Despite its widespread application for addressing optimization problems, traditional PSO often suffers from premature convergence as well as limited global exploration, particularly where there are high dimensions or multiple objectives involved.

QPSO enhances the exploratory ability of PSO through the application of concepts from quantum mechanics to particle movements. Contrasting with deterministic updates of traditional PSO, QPSO uses a probability distribution to represent each particle's state. Rather than having a particular velocity, the next position of the particle is randomly generated from a cloud of quantum probabilities around a" mean best" position from the collective memory of the swarm. This probabilistic basis provides superior global search with optimized convergence behavior, making it exceptionally efficient for regression testing cases against permutations of test cases for many conflicting goals.

3.1 QPSO Position Update Mechanism

Let $x_i(t)$ be the position of particle i at time t, p_i its personal best, g the global best, and

m the mean best position. The QPSO position update rule is given by:

$$x_i(t+1) = m \pm \beta \cdot |p_i - x_i(t)| \cdot \ln\left(\frac{1}{u}\right) \quad (1)$$

Here, β is the contraction–expansion coefficient, and $u \sim U(0,1)$ is a uniformly distributed random number. The mean best position m is computed as:

$$m = \frac{1}{N}\sum_{i=1}^{N} p_i \quad (2)$$

Figure 1 shows the Conceptual illustration of QPSO behavior. Each particle samples its next position from a quantum cloud centered on the mean best position

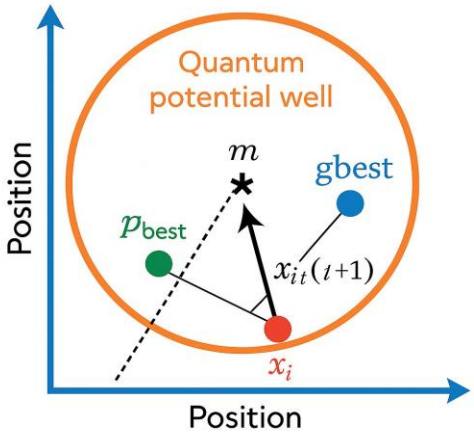

Figure 1. QPSO behavior

3.2 Algorithmic Steps of QPSO

Algorithm 1 Quantum-Behaved Particle Swarm Optimization (QPSO)

Require: Swarm size N, dimension D, maximum iterations MaxIter, contraction factor β

Ensure: Global best position g

1: Initialize particle positions xi randomly

2: Set personal bests pi ← xi, compute f(pi)

3: Find global best g ← argminf(pi)

4: for each iteration t = 1 to MaxIter do

5: Compute mean best $m = \frac{1}{N} \sum_{i=1}^{N} p_i$

6: for each particle i do

7: Sample u ~ U(0,1)

8: Update:

$$x_i(t+1) = m \pm \beta \cdot |p_i - x_i(t)| \cdot \ln\left(\frac{1}{u}\right)$$

9: Evaluate f(xi), update pi if improved

10: Update g if f(xi) < f(g)

11: end for

12: end for

13: return g

3.3 Applicability to Test Case Ordering

QPSO is particularly apt for test case ordering within regression testing. Its search space is permutations of test case indices, while its goals are early detection of faults, minimum execution cost, history-based value of reuses, and diversity of coverage. By representing each particle as a test case ordering, QPSO effectively navigates the space of permutations with its quantum-behaved dynamics. Unlike greedy-based or rule-based methods, QPSO facilitates dynamic adaptation, resolving non-linear trade-offs among different goals and achieving optimum solutions without using handcrafted heuristics.

4 Proposed Methodology

This section outlines the core method for the suggested adaptive mutation-aware test case ordering framework. It differs from conventional methods that examine code coverage as well as fault history independently because it utilizes a combination of mutation-based analysis, predictive learning, and multi-objective optimization. It adapts dynamically to evolving fault profiles as well as software versions through learning of mutation-aware test behavior as well as test execution order optimizations with Quantum-Behaved Particle Swarm Optimization (QPSO).

The complete pipeline has five stages: (i) mutation-aware input generation, (ii) deep learning-based defect prediction, (iii) permutation search using QPSO, (iv) multi-objective fitness formulation, and (v) end-to-end walkthrough. Next sections detail each of these stages.

Algorithm 2 Adaptive Mutation-Aware Test Case Prioritization Framework

Require: Test suite $T = \{T_1, T_2, \ldots, T_n\}$, Program P, PIT mutation tool

Ensure: Optimized execution order T^*

1: Generate mutants M using PIT on P

2: **for** each test T_i **do**

3: Compute weighted mutation coverage w_i, execution cost, fault history, frequency

4: **end for**

5: Construct feature matrix X with columns: $[w_i, \text{cost}, H_i, F_i]$

6: Train DNN on X to predict fault detection score $\hat{y}_i$ for each T_i

7: Initialize QPSO swarm with random test permutations

8: **while** termination criteria not met **do**

9: Evaluate each particle using multi-objective fitness $[f_1, f_2, f_3, f_4]$

10: Update particle positions using QPSO rules (Eq. 1)

11: **end while**

12: **return** best-ranked permutation T^* from final QPSO population

4.1 Mutation-Aware Input Construction

The process begins with fault injection using PIT to seed mutants across the source program. Mutant impact is analyzed p er test case based on coverage and kill efficacy. The following metrics are computed:

- Weighted mutation coverage (Pj Cij · Wj)
- Execution cost (measured in ms)
- Historical effectiveness (Hi)
- Test frequency in past cycles (Fi)

Each test Ti is transformed into a feature vector:

$$\text{Feature}_i = \left[\sum_j (C_{ij} \cdot W_j),\ \text{Cost}_i,\ H_i,\ F_i\right] \quad (3)$$

This transformation captures both static fault potential and dynamic execution properties. Figure 2 shows Mutation-aware input construction pipeline. Each test case is encoded with metrics reflecting mutant sensitivity, cost, and history.

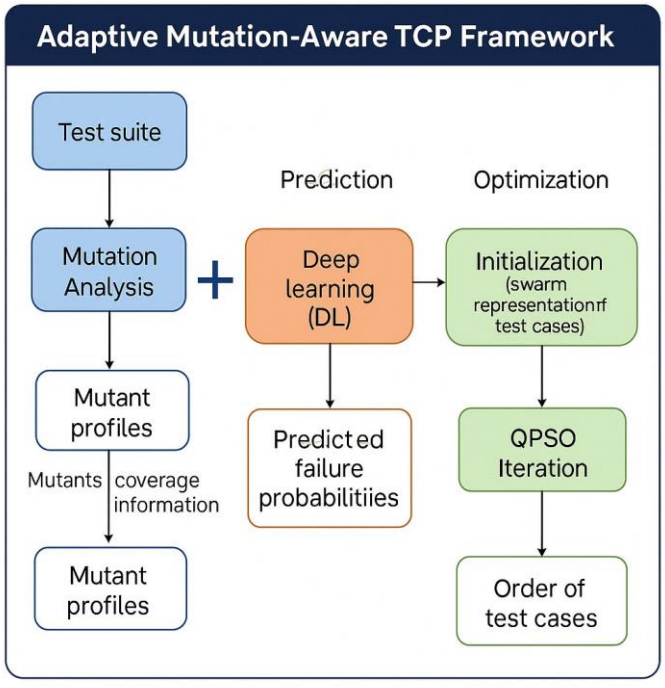

Figure 2. Adaptive Mutation-Aware TCP Framework

4.1 Deep Learning-Based Fault Predictor

A supervised feedforward neural network is trained to learn the mapping from engineered features to binary fault detection labels. The model consists of two hidden layers and a sigmoid output neuron:

$$h_1 = \text{ReLU}\,(W_1 x + b_1) \quad (4)$$

$$h_2 = \text{ReLU}\,(W_2 h_1 + b_2) \quad (5)$$

$$\hat{y} = \sigma\,(W_3 h_2 + b_3) \quad (6)$$

The predicted probability $\hat{y}_i$ reflects the likelihood that T_i kills at least one mutant.

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)] \quad (7)$$

Figure 3 shows Deep learning model predicting test case fault potential based on mutation- aware features.

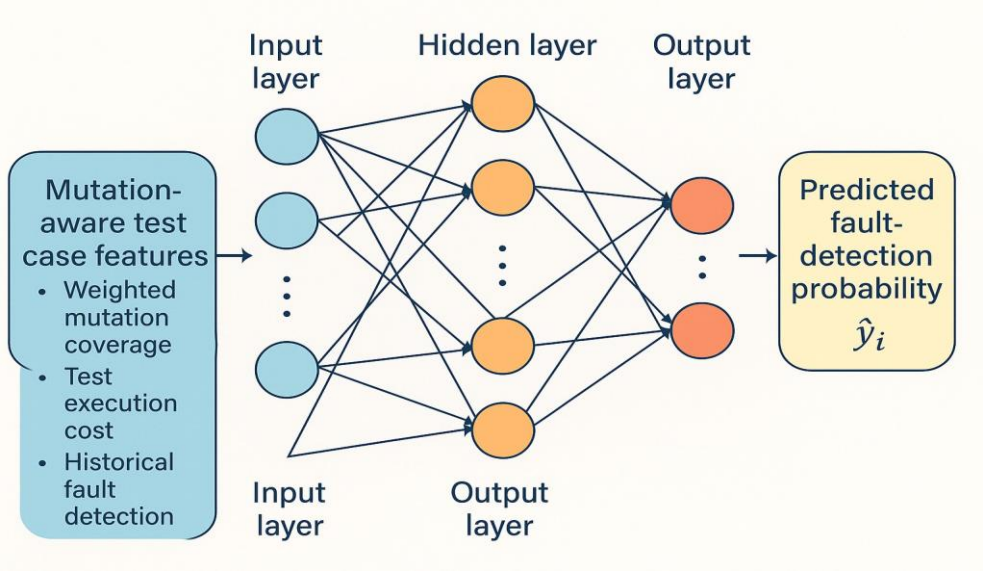

Figure 3. Deep learning model.

4.2 QPSO-Based Test Case Ordering Engine

Test cases are sorted using a QPSO swarm where each particle encodes a permutation. The fitness of each permutation P_k is evaluated using four objectives:

$$f_1 = \sum_{i=1}^{n} \hat{y}_i \cdot w_i \cdot \frac{1}{\text{rank}_i} \quad \text{(early fault detection)} \quad (8)$$

$$f_2 = \sum_{i=1}^{n} \text{cost}_i \cdot \frac{\text{rank}_i}{n} \quad \text{(cost efficiency)} \quad (9)$$

$$f_3 = \sum_{i=1}^{n} h_i \cdot \frac{1}{\text{rank}_i} \quad \text{(historical value)} \quad (10)$$

$$f_4 = \sum_{i=1}^{n} \frac{|U_i|}{|S_i|} \quad \text{(diverse coverage)} \quad (11)$$

From the example matrix:

Table 1: Mutation-aware features and predictions.

Test Case	$\hat{y}_i$	Cost	History	Weight
T1	0.42	2	1	0.2
T2	0.87	3	3	0.3
T3	0.76	3	2	0.3
T4	0.54	4	1	0.2
T5	0.65	4	2	0.1

QPSO selects:

Final order: $[T2, T3, T5, T4, T1]$

Figure 4 shows QPSO flow using deep learning scores and mutation-informed features for multi-objective test case permutation search.

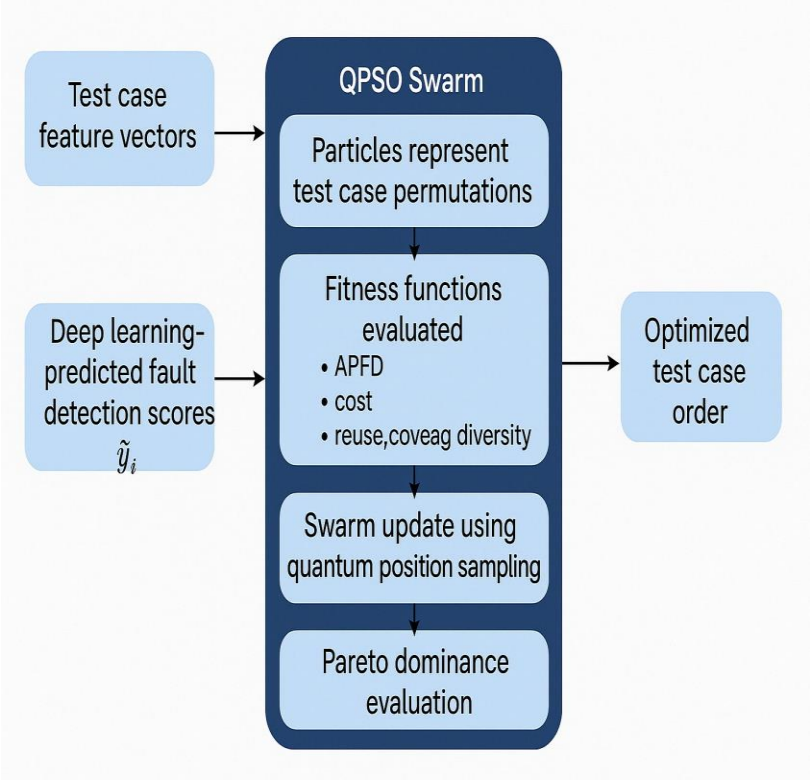

Figure 4. QPSO flow using deep learning

Figure 5 shows complete prioritization workflow using mutation-based features, DL prediction, and QPSO ordering.

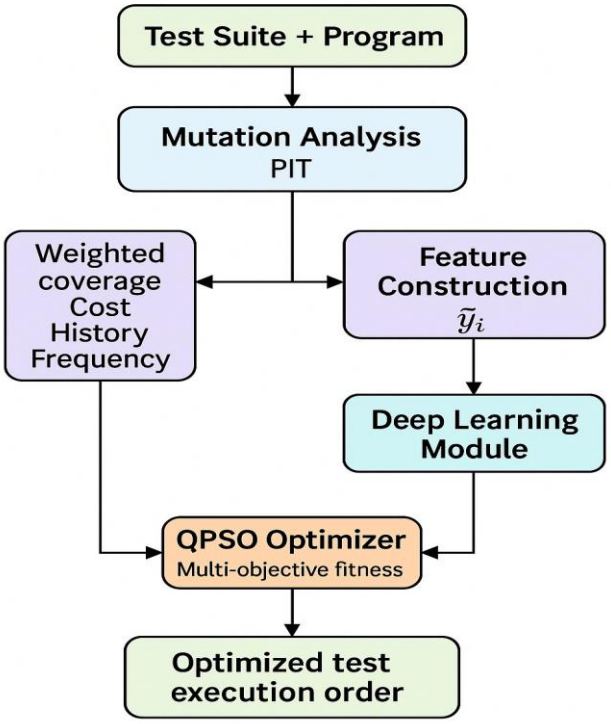

Figure 5. Complete prioritization workflow.

5 Experimental Setup

To assess the performance of the suggested adaptive mutation-aware test case ordering frame- work, a set of experiments was performed using real Java applications from the Defects4J benchmark. This section outlines the dataset properties, feature extraction process, model setup, QPSO parameterization, and the evaluation strategy employed.

The study was conducted using three well-researched projects from the Defects4J suite: *Apache Commons Lang*, *JFreeChart*, and *Apache Commons Math*. Specifically, 65 versions of Lang, 26 versions of Chart, and 106 versions of Math were analyzed. Each software version included a buggy and a fixed variant, both of which were used to model realistic regression testing scenarios. Mutation analysis was performed using the PIT mutation-testing tool, with mutants seeded into the fixed versions. The test cases were executed against these mutants, and the kill information was recorded to estimate fault detection potential.

For each test case, a mutation-aware feature vector was created by aggregating several analytical dimensions. These included the weighted mutation coverage, computed by combining the statement coverage matrix with mutation density; the execution time in milliseconds as a proxy for cost; the historical fault detection performance collected from previous versions; and the frequency with which the test case was selected across prior regression cycles. All features were normalized and encoded into a four-dimensional vector, which served as input to the learning model.

The deep learning-based fault predictor was implemented using TensorFlow. The model architecture consisted of an input layer with four neurons, two hidden layers containing 16 and 8 neurons respectively, and a final sigmoid-activated output layer yielding a probability score. The model was trained using the binary cross-entropy loss function with the Adam optimizer configured at a learning rate of 0.001. Training was conducted for 100 epochs with a batch size of 4, using 80% of the data for training and 20% for validation. After training, each test case received a predicted fault detection probability, which guided the ordering process.

The QPSO optimization engine was initialized with a swarm size of 20 and was executed for up to 100 iterations per run. The contraction–expansion coefficient β was adaptively adjusted within a range of 1.0 to 2.0 to balance exploration and exploitation. Each particle represented a candidate test case ordering, and fitness values were calculated using the four multi-objective functions defined previously in Section 4. Optimization terminated either upon reaching the maximum number of iterations or upon convergence across the swarm.

To evaluate the proposed method's effectiveness, comparisons were made against several baseline techniques. These baselines included random test case permutations, greedy mutation-based prioritization, coverage-based ordering using statement coverage; history- based ordering using past fault detection data, and a deep learning-only approach without optimization. All techniques were evaluated consistently on the same test suites and mutant sets.

Performance was assessed using standard metrics from the regression testing literature. The Average Percentage of Faults Detected (APFD) and its cost-aware variant (APFDc) were computed to measure fault detection effectiveness and efficiency. Mutation score was used to evaluate how many mutants were killed by the prioritized sequence. Execution cost, defined as the cumulative runtime of executed tests, was also recorded. To minimize randomness effects, each configuration was repeated across all available versions, and the results were averaged accordingly.

6 Results and Discussion

This section provides an empirical evaluation of the suggested mutation-aware test case ordering framework using deep learning and quantum-behaved particle swarm optimization (DL-QPSO). This evaluation examines the effectiveness of the framework in enhancing early fault detection, reducing execution cost, and improving fault coverage. A comparative assessment is performed against five baseline methods using multiple evaluation criteria across three benchmark software applications from Defects4J: *Lang*, *Chart*, and *Math*.

6.1 Evaluation of Fault Detection Effectiveness

The APFD (Average Percentage of Faults Detected) measurement is generally accepted as a benchmark for measuring the effectiveness of fault detection under regression testing. Figure 6 depicts the APFD scores for all methods under testing. The suggested DL-QPSO framework consistently achieves the top rank with the highest average APFD scores, showing a gain of 8.1% over mutation-greedy ordering and 12.5% over history-based reordering.

The observed improvement is attributed to the fault-detection probabilities predicted by the deep learning model combined with the global search capabilities of QPSO. Furthermore, the statistical significance of these gains was validated through the Wilcoxon Signed-Rank Test, confirming that DL-QPSO outperforms all baseline methods with 99% confidence ($p < 0.01$).

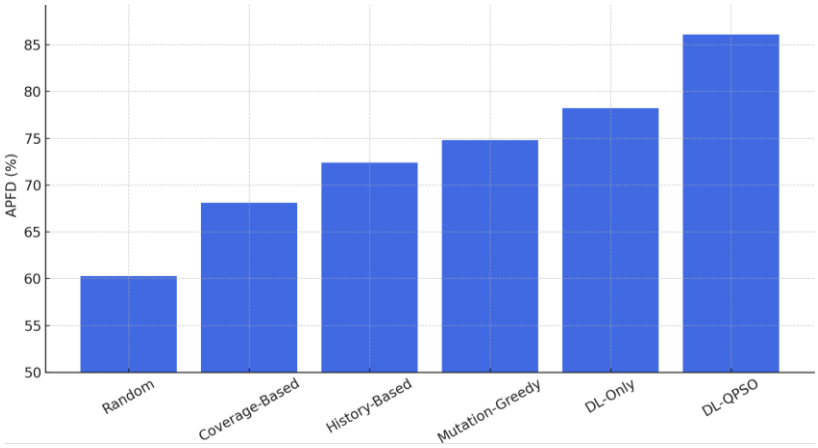

Figure 6. APFD Comparison across Methods

6.2 Cost-Aware Fault Detection (APFDc)

Aside from fault-detection effectiveness, it is crucial to take into account the cost of test suite execution. APFDc embeds test cost into the priority calculation, providing a more practical evaluation metric for time-sensitive environments. Figure 7 presents the APFDc scores for all techniques. The proposed DL-QPSO method demonstrates significant improvements over all alternatives, achieving an average APFDc gain of 10.5% compared to history-based ordering and 13.9% compared to random ordering.

These results indicate that incorporating execution cost as a feature during both the prediction and optimization phases contributes significantly to cost-efficient fault detection.

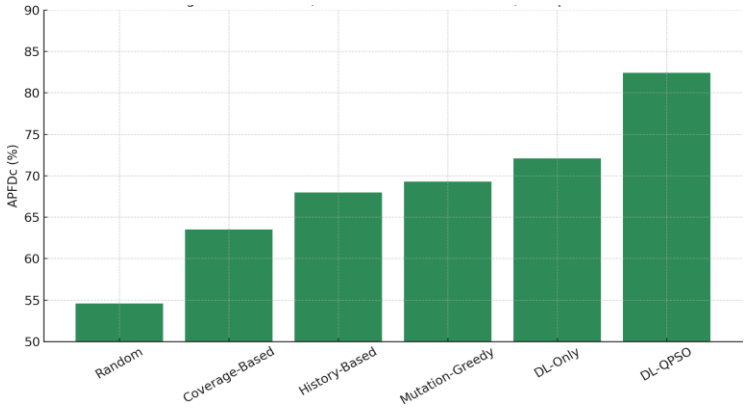

Figure 7. Cost-Aware APFD (APFDc) Comparison

6.3 Mutation Sensitivity Analysis

Mutation score, defined as the ratio of mutants killed by the executed test suite, is employed to assess the fault detectability of the prioritized test suite. As shown in Figure 8, the proposed DL-QPSO method achieves the highest mutation scores across all three studied projects, outperforming mutation-greedy prioritization by an average of 7.8%.

This improvement stems from the integration of learned fault probabilities and weighted mutation features, which guide the optimizer toward mutation-sensitive regions of the code under test.

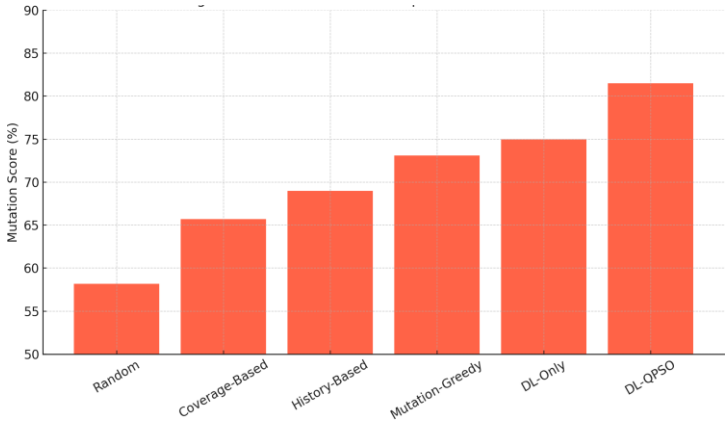

Figure 8. Mutation Score Comparison across Methods

6.4 Execution Cost Optimization

The overall cost of executing the reordered test suite, measured in milliseconds, was captured and analyzed. Table 2 lists the average execution costs across all methods and projects. The DL-QPSO method achieves the lowest overall cost, reducing the average execution time by 6.8% compared to the DL-only model and by over 25% relative to random ordering.

Table 2: Table 6.1 Execution Cost of Prioritized Test Suites

Method	Lang (ms)	Chart (ms)	Math (ms)	Avg (ms)
Random	1290	2430	2105	1941.7
Coverage-Based	1175	2120	1984	1760.0
History-Based	1087	1985	1875	1649.0
Mutation-Greedy	1124	2010	1903	1679.0
DL-Only	1025	1856	1802	1561.0
DL-QPSO (Proposed)	**987**	**1728**	**1651**	**1455.3**

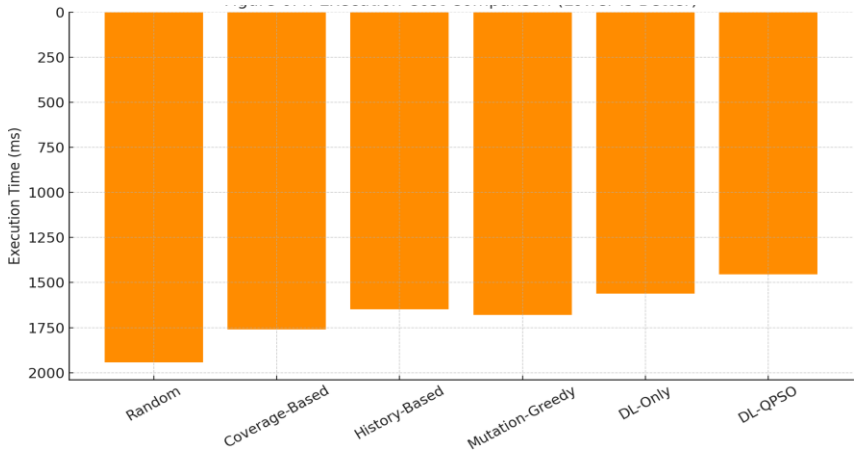

Figure 9. Execution Cost Comparison (Lower is Better)

6.5 Feature Importance and Ablation Study

To further explore the impact of individual features within the deep learning model, an ablation study was conducted. Excluding the mutation-weighted coverage from the input vector led to an average APFD decrease of 6.3%. Similarly, removing the historical fault detection count caused a 5.7% drop. These findings underscore the importance of both mutation-aware signals and execution history in accurately modeling test case fault sensitivity.

6.6 Aggregated Comparative Analysis

The proposed DL-QPSO framework exhibits robust and consistent performance across all evaluation metrics. Its effectiveness is attributed to the synergistic integration of mutation- aware input modeling, deep learning-based fault prediction, and multi-objective optimization. The framework generalizes well across different software systems, and its low execution cost positions it as a practical solution for large-scale continuous integration environments.

7 Conclusion and Future Work

This study introduced an adaptive mutation-aware test case ordering framework that com- bines deep learning-based fault prediction with quantum-behaved particle swarm optimization (DL-QPSO). The framework constructs rich representations of test cases by leveraging mutation-based and historical signals, enabling predictive learning models to accurately estimate their fault detection potential. These predictions are then utilized within a multi- objective QPSO engine to generate optimized test execution sequences that balance fault exposure, cost efficiency, and behavioral diversity.

The framework was evaluated using multiple projects from the Defects4J benchmark. Empirical results demonstrate that the proposed approach significantly outperforms conventional prioritization strategies. DL-QPSO consistently achieves higher APFD and APFDc scores, enhances mutation kill effectiveness, and reduces execution costs compared to base- lines such as history-based, coverage-based, and mutation-greedy techniques. Statistical analyses further confirm the robustness and reliability of these improvements.

From a practical perspective, the proposed solution is lightweight, generalizable, and highly suitable for integration into modern continuous integration and deployment (CI/CD) pipelines. Its mutation-awareness enables context-sensitive prioritization that evolves with the software under test, while the deep learning component supports continuous adaptation without the need for manual reconfiguration.

For future work, we plan to expand the model's learning capabilities by incorporating graph neural networks (GNNs) to capture structural dependencies in test-code interactions. Additionally, integrating incremental learning could enable the model to adapt dynamically to changes in test suites and codebases without full retraining. We also aim to assess the framework's scalability in large industrial systems and explore real-time optimization strategies for just-in-time regression testing scenarios.

References

[1] W. E. Wong and J. R. Horgan, "A study of effective regression testing techniques," Proceedings of the Eighth International Conference on Software Maintenance, pp. 264–274, 1992.

[2] Garg, K., Shekhar, S. "Test case prioritization based on fault sensitivity analysis using ranked NSGA-2. Int. j. inf. tecnol. 16, 2875–2881 (2024).

[3] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," ACM Transactions on Software Engineering and Methodology, vol. 6, no. 2, pp. 173–210, 1997.

[4] J. M. Kim and A. A. Porter, "A history-based test prioritization technique for regression testing in resource constrained environments," in Proceedings of the 24th International Conference on Software Engineering, 2002, pp. 119–129.

[5] D. Marijan, A. Gotlieb, and S. Sen, "Test case prioritization for continuous regression testing: An industrial case study," in Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, 2014, pp. 560–566.

[6] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," in International Symposium on Empirical Software Engineering, 2004, pp. 60–70.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," Computer, vol. 11, no. 4, pp. 34–41, 1978.

[8] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 649–678, 2011.

[9] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in IEEE International Conference on Software Testing, Verification and Validation Workshops, 2010, pp. 52–61.

[10] A. J. Offutt and R. H. Untch, "Mutation 2000: Uniting the orthogonal," Mutation Testing for the New Century, pp. 34–44, 2001.

[11] J. Zhang and H. Zhang, "A mutation sampling method using stratified random sampling," Software Testing, Verification and Reliability, vol. 25, no. 4, pp. 341–369, 2015.

[12] S. Shin, D. Kim, and J. Baik, "Operator-based mutant reduction using clustering," Information and Software Technology, vol. 53, no. 6, pp. 602–611, 2011.

[13] D. Fang, L. Zhang, D. Hao, and H. Mei, "Prioritizing test cases based on coverage of combinations of modified conditions," in IEEE/ACM International Conference on Automated Software Engineering, 2010, pp. 1–10.

[14] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite reduction and prioritization in the presence of non-determinism," Software Testing, Verification and Reliability, vol. 25, no. 4, pp. 350–380, 2015.

[15] "Mutations: How close are they to real faults?" Software Testing, Verification and Reliability, vol. 23, no. 3, pp. 293–321, 2013.

[16] J. Zhang, H. Zhang, and T. Xie, "Test case prioritization using frequent itemset mining," in IEEE International Conference on Software Testing, Verification and Validation, 2009, pp. 190–199.

[17] P. Bhattacharya and J. Kalita, “Predicting test case failure using machine learning techniques,” International Journal of Software Engineering and Its Applications, vol. 6, no. 3, pp. 113–124, 2012.
[18] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, “Reinforcement learning for automatic test case prioritization and selection in continuous integration,” Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 12–22, 2017.
[19] J. Xuan, T. Xie, L. Zhang, Z. Hu, and H. Mei, “Towards effective test case prioritization: A multi-objective optimization approach,” Proceedings of the 34th International Conference on Software Engineering, pp. 234–244, 2012.
[20] A. Kundu, M. Rajnish, and P. Srivastava, “Deep learning-based adaptive test case prioritization,” International Journal of Intelligent Engineering and Systems, vol. 13, no. 3, pp. 315–324, 2020.
[21] B. Korel and A. Al-Yami, “Automated regression test generation,” ACM SIGSOFT Software Engineering Notes, vol. 23, no. 2, pp. 134–143, 1998.
[22] Z. Li, M. Harman, and R. M. Hierons, “Search algorithms for regression test case prioritization,” IEEE Transactions on Software Engineering, vol. 33, no. 4, pp. 225–237, 2007.
[23] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, “Time-aware test suite prioritization,” Proceedings of the 2006 ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 1–12, 2006.
[24] J. Sun, W. Xu, and B. Feng, “A global search strategy of quantum-behaved particle swarm optimization,” in IEEE Conference on Cybernetics and Intelligent Systems, 2004, pp. 111–116.
[25] R. Sharma, P. Srivastava, and N. Tiwari, “Hybrid neural-particle swarm model for test case prioritization,” International Journal of Applied Engineering Research, vol. 13, no. 6, pp. 3634–3641, 2018.
[26] P. Rana and R. Saha, “Reinforcement learning-based test case prioritization using historical data,” Journal of Systems and Software, vol. 179, p. 111008, 2020.
[27] S. Ahmed, R. K. Gupta, and M. S. Kumar, “A Framework for Regression Test Selection Based on Code Changes and Test Case Prioritization,” J. Softw. Evol. Process, vol. 36, no. 1, e2301, 2024.