# Adaptive Budget-Constrained Execution Framework for Prioritized Regression Test Suites

S. Sowmyadevi[1*] and Anna Alphy[2]
{ss2860@srmist.edu.in[1*], annaa@srmist.edu.in[2]}

Department of CSE, SRMIST, Delhi-NCR Campus, Ghaziabad ,201204, Uttar Pradesh, India[1, 2]

**Abstract.** In regression testing, executing all test cases is often impractical due to strict time and resource limitations. This paper presents an adaptive test execution framework designed to maximize fault detection within constrained resources. The approach formulates scheduling as a multi-objective optimization problem, aiming to increase cumulative fault detection while minimizing execution costs. A Multi-Objective Particle Swarm Optimization (MOPSO) algorithm dynamically generates optimized execution sequences, with penalties applied to discourage infeasible solutions. An illustrative example demonstrates how the framework adapts execution plans based on available resources. Experimental evaluations on benchmark test suites show that the proposed approach consistently achieves higher fault detection efficiency compared to greedy and random scheduling strategies. This work contributes a scalable and resource-aware solution for enhancing quality assurance in real-world regression testing environments.

**Keywords:** Regression Testing, Test Case Execution, Time Budgeting, Fault Detection, Optimization.

## 1 Introduction

Regression testing plays a critical role in ensuring software reliability by verifying that recent code changes do not introduce new faults into previously validated functionality. As software systems grow in size and complexity, their associated test suites also expand, making it impractical to execute all test cases in every development cycle due to time and resource limitations.

This challenge is particularly evident in agile and continuous integration (CI) environments, where rapid feedback is essential, and testing resources are often tightly constrained. Traditional test case prioritization (TCP) techniques attempt to execute the most critical tests first. However, many methods assume unlimited resources or rely on static priorities. This often leads to inefficient fault detection and, in practice, may cause testing operations to exceed budgets or miss critical faults.

Recent advances in optimization algorithms, particularly swarm intelligence techniques, offer promising solutions. Multi-Objective Particle Swarm Optimization (MOPSO) has demonstrated strong potential for balancing competing objectives and efficiently navigating complex search spaces. However, its application to budget-aware regression test scheduling remains underexplored. To address this gap, we propose an adaptive execution optimization framework that formulates prioritized test case scheduling as a bi-objective optimization problem. The framework aims to schedule the prioritized tests within a strict budget constraint to maximize

ICITSM-Part II 2025, April 28-29, Tiruchengode, India

DOI 10.4108/eai.28-4-2025.2358034

fault detection effectiveness.

The key contributions of this study are as follows:

- A formal representation of budget-constrained regression test execution as a bi-objective optimization problem balancing fault detection and execution cost.
- A novel application of MOPSO to dynamically construct optimized test execution plans under strict resource constraints.
- A comprehensive evaluation using realistic datasets demonstrating superior performance over random and greedy approaches.

The remainder of this paper is organized as follows: Section 2 reviews related work, Section 3 presents the proposed MOPSO-based framework, Section 4 describes the experimental setup, Section 5 discusses results and analysis, and Section 6 concludes with future research directions.

## 2 Related Work

### 2.1 Test Case Prioritization Techniques

Test case prioritization (TCP) is a major focus of regression testing research, aiming to increase fault detection efficiency by reordering test execution. Foundational studies introduced empirical approaches to test case prioritization, demonstrating their ability to detect faults earlier in the testing process [1], [11]. Early strategies relied heavily on structural coverage metrics such as statement and branch coverage [3], [4]. Other work leveraged historical fault data to guide prioritization, improving effectiveness in systems with extensive version histories [5], [6].

### 2.2 Cost-Aware and Budget-Constrained Regression Testing

Traditional TCP methods often assume virtually unlimited testing resources, which is unrealistic in modern continuous integration and delivery environments. To address this, cost-aware TCP approaches were introduced to optimize fault detection within strict execution budgets [7]. Further studies have proposed Pareto-optimal solutions to balance fault detection with execution costs, enabling more practical trade-offs in resource-constrained testing cycles [2].

### 2.3 Optimization-Based Execution Scheduling

Search-based software engineering approaches have been widely applied to TCP, using optimization algorithms such as genetic algorithms, simulated annealing, and multi-objective particle swarm optimization (MOPSO) to efficiently schedule test execution [6]. Comparative studies demonstrate that multi-objective optimization strategies outperform single-objective methods, achieving superior fault detection and cost-effectiveness [3], [10].

### 2.4 Adaptive and Dynamic Scheduling Techniques

Recent research emphasizes adaptive scheduling strategies that incorporate real-time execution feedback. Techniques based on reinforcement learning have been explored to dynamically refine prioritization policies over time [2]. Hybrid approaches combining historical data with adaptive execution models show potential for improving responsiveness in evolving software systems.

### 2.5 Datasets and Benchmarking

Defects4J has become a standard dataset for evaluating TCP, offering real-world faults and supporting reproducible regression testing experiments [8]. Mutation analysis tools like PIT are widely used to assess test suite effectiveness through simulated faults [9]. In addition, studies have underscored the importance of rigorous benchmarking practices, including dependency-aware test selection and mutation-based evaluation, to ensure reproducibility and practical applicability [10].

## 3 Proposed Methodology

This section describes the MOPSO-based framework for balancing fault detection and execution cost under resource constraints).

### 3.1 Problem Formulation

Let the set of available test cases be given by $T = \{t_1, t_2, \ldots, t_n\}$. Each test case $t_i$ is associated with a probability of fault detection $f_i$ and an execution cost $c_i$. The testing environment restricts the total execution cost to a defined budget constraint $B$. Thus, the scheduling task can be formulated as a multi-objective optimization problem, aiming to maximize overall fault detection while minimizing execution cost.

Define a binary decision variable $x_i$:

$$x_i = \begin{cases} 1, if\ test\ case\ t\_i\ is\ selected\ for\ execution \\ 0, otherwise \end{cases} \quad (1)$$

The objectives are formally stated as:

**Maximize Fault Detection:**

$$\text{Maximize} \quad F = \sum_{i=1}^{n} f_i x_i \quad (2)$$

**Minimize Execution Cost:**

$$\text{Minimize} \quad C = \sum_{i=1}^{n} c_i x_i \quad (3)$$

subject to the budget constraint:

$$\sum_{i=1}^{n} c_i x_i \leq B \quad (4)$$

To illustrate the setup, consider a small example where $T = \{t_1, t_2, t_3, t_4, t_5\}$, with their respective attributes shown in Table 1.

**Table 1.** Attributes of Test Cases for Illustrative Example.

| Test Case | Fault Detection Likelihood ($f_i$) | Execution Cost ($c_i$) |
|---|---|---|
| $t_1$ | 0.9 | 5 |
| $t_2$ | 0.7 | 3 |
| $t_3$ | 0.6 | 2 |
| $t_4$ | 0.5 | 4 |
| $t_5$ | 0.8 | 6 |

Assume the available budget is given by $B = 10$ units. The objective is to choose a sub- set of the given test cases such that their aggregate execution cost does not exceed 10, while maximizing the total likelihood of fault detection.

### 3.2 MOPSO-Based Execution Optimization

To address this bi-objective problem, we employ a Multi-Objective Particle Swarm Optimization (MOPSO) strategy. Each particle in the swarm represents a potential execution plan, encoded as a binary vector $X = (x_1, x_2, \ldots, x_n)$, where $x_i = 1$ if and only if the test case $t_i$ is selected.

Particles are assessed using a fitness function that simultaneously considers both objectives, incorporating penalties for any violations of the budget constraint:

$$\text{Fitness}(X) = \alpha\left(1 - \frac{F}{F_{\max}}\right) + \beta\left(\frac{C}{B}\right) + \gamma \times \text{Penalty} \tag{5}$$

where $\alpha$ and $\beta$ are weighting factors, $F_{\max}$ denotes the maximum possible cumulative fault detection, and the penalty term is defined as:

$$\text{Penalty} = \max\left(0, \frac{C - B}{B}\right) \tag{6}$$

During initialization, the particles are created randomly, producing different combinations of test cases. For example, one particle might represent $\{t_1, t_3\}$ with a total cost of 7 and a fault detection sum of 1.5, while another could represent $\{t_2, t_3, t_4\}$ with a total cost of 9 and a fault detection sum of 1.8. Particles that violate the budget constraint are penalized and are gradually steered toward feasible solutions with higher fault detection effectiveness.

During swarm evolution, the velocities and positions of the particles are updated according to MOPSO rules, while a Pareto-optimal archive is maintained to store the best non-dominated solutions. This process supports dynamic adaptation of the execution schedule based on the available resources.

The process begins with prioritized test cases and budget information, proceeds through swarm initialization and evolution, evaluates fitness under budget constraints, and finally out- puts an optimized test execution plan to maximize fault detection within the available resources.

## 4 Experimental Setup

The proposed framework for testing with a given limited budget was assessed for its effectiveness using actual programs within the Defects4J dataset [23]. The *Chart*, *Lang*, and *Math* projects were selected, representing a range of different sizes and complexities. Each project provides multiple faulty versions along with developer-written test suites, enabling a realistic evaluation of regression testing strategies. To estimate the fault detection likelihood of individual test cases, mutation analysis was performed using the PIT tool [24].

Experiments were conducted on a system equipped with an Intel Core i7 processor running at 3.6 GHz, 32 GB of RAM, and Ubuntu 22.04 LTS. The implementation was developed using Python 3.11 and scientific computing libraries such as NumPy and SciPy. These system specifications ensured that the MOPSO optimization and test execution simulations could proceed efficiently without artificial bottlenecks. Fig 1 presents the overall flow of the proposed optimization framework.

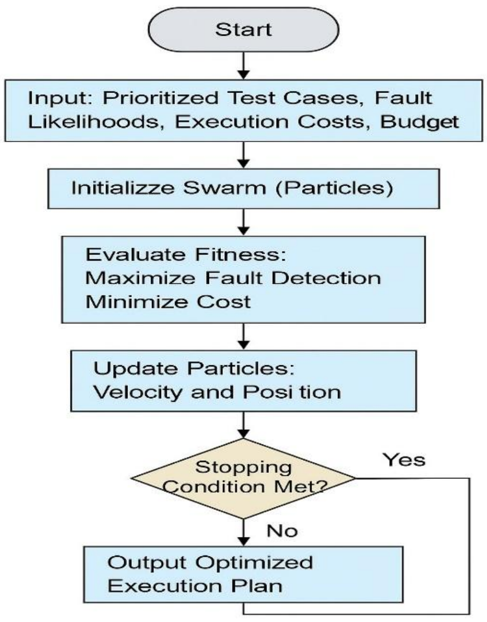
Start
Input: Prioritized Test Cases, Fault Likelihoods, Execution Costs, Budget
Initializze Swarm (Particles)
Evaluate Fitness: Maximize Fault Detection Minimize Cost
Update Particles: Velocity and Posi tion
Stopping Condition Met?
Yes
No
Output Optimized Execution Plan

**Fig. 1.** Workflow of Test Case Optimization using Particle Swarm Algorithm.

Flow Diagram of the Proposed MOPSO-Based Execution Optimization Performance evaluation primarily relied on the Average Percentage of Faults Detected under budget constraints (APFDc) [25], which captures the rate of fault detection relative to re- source consumption. Higher APFDc values reflect more effective early fault detection under tight budget conditions. Additionally, the total number of faults detected within varying budget thresholds (ranging from 30% to 70% of the total execution cost) was recorded to assess the practical effectiveness of different scheduling strategies.

The parameters of the MOPSO framework were determined through preliminary tuning. The swarm size was set to 30, with a maximum of 100 iterations. The inertia weight was fixed at 0.7, and both the cognitive and social learning factors were assigned values of 1.5. A mutation probability of 0.1 was introduced to maintain diversity among particles. Budget constraints were enforced using penalty functions incorporated into the fitness evaluations. Each experimental configuration was independently executed 30 times to accommodate stochastic variations, and the results presented represent averaged values across these repetitions.

## 5 Results and Discussion

This section reports the experimental results obtained by applying the proposed MOPSO-based adaptive test case execution framework. Comparative analysis was conducted against two baseline approaches: random scheduling and greedy fault-to-cost ratio scheduling. The evaluation examines both the effectiveness of fault detection and the efficiency of resource usage under varying budget constraints.

Table 2 summarizes the number of faults detected and the APFDc scores achieved by each method across different budget levels. The budget was adjusted between 30% and 70% of the total available execution cost to represent varying degrees of resource availability.

**Table 2.** Comparison of Faults Detected and APFDc Across Methods at Different Budget Levels.

| 2*Budget Level | Random | | Greedy | | Proposed (MOPSO) | |
|---|---|---|---|---|---|---|
| | Faults | APFDc | Faults | APFDc | Faults | APFDc |
| 30% | 9 | 0.54 | 11 | 0.63 | 13 | 0.71 |
| 50% | 15 | 0.61 | 18 | 0.72 | 21 | 0.80 |
| 70% | 22 | 0.68 | 26 | 0.79 | 29 | 0.86 |

The outcomes clearly show that the proposed MOPSO-based optimization consistently outperforms both random and greedy scheduling methods across all budget levels. Under a restricted budget constraint of 30%, the MOPSO strategy detects 13 faults with an APFDc of 0.71, whereas random scheduling detects only 9 faults with an APFDc of 0.54. As the available budget increases, this performance gap remains, demonstrating the robustness and adaptability of the proposed approach in maximizing fault detection under resource-constrained conditions. The trends for fault detection across the three methods at different budget levels are illustrated in Fig 2. It can be observed that the number of faults detected by the proposed MOPSO-based method consistently exceeds those detected by both random and greedy strategies. This advantage becomes even more significant as the available budget grows, highlighting the effectiveness of swarm-based adaptive scheduling in budget-constrained environments.

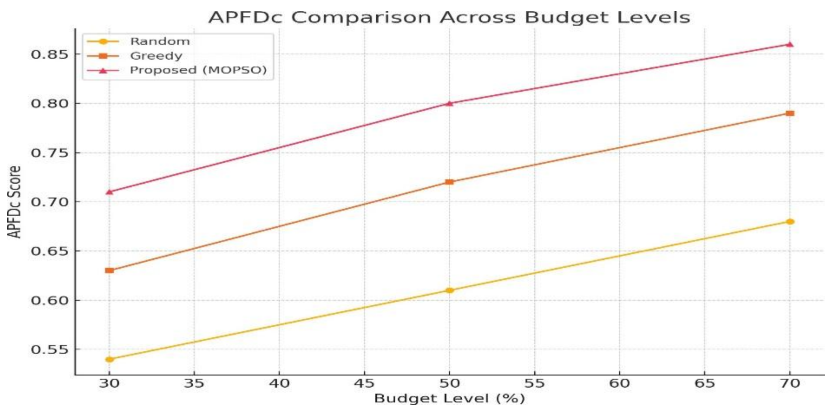

**Fig. 2.** Faults Detected vs Budget Level.

A closer examination of early fault detection effectiveness, as measured by APFDc, is presented in Fig 3. The APFDc scores achieved by the MOPSO-based approach are consistently higher across all budget levels, reflecting its capability to prioritize fault-prone test cases early in the execution process even under tight resource constraints. The steepness of the curve for the proposed method indicates a faster rate of fault detection, which is particularly advantageous for regression testing cycles operating under stringent time or resource limitations.

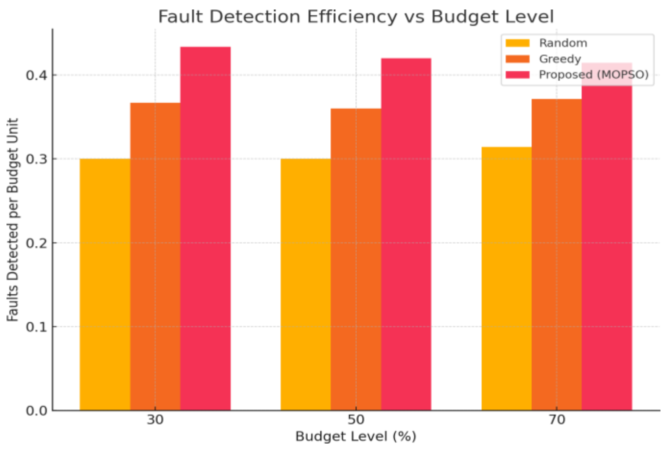

**Fig. 3.** APFDc Comparison Across Budget Levels.

In addition to examining the number of detected faults and the early fault-detection rates, we also measured the resource efficiency by the number of faults detected per budget spent. This is shown in Fig 4. As can be seen, the MOPSO-based approach always enjoys better fault detection efficiency compared to the competing ones at all budget levels. This also corroborates that the proposed method can not only raise the absolute number of detected failures, but it can also make fault detection as an economically attractive testing process, which makes it applicable to real-world CI environments requiring resource awareness.

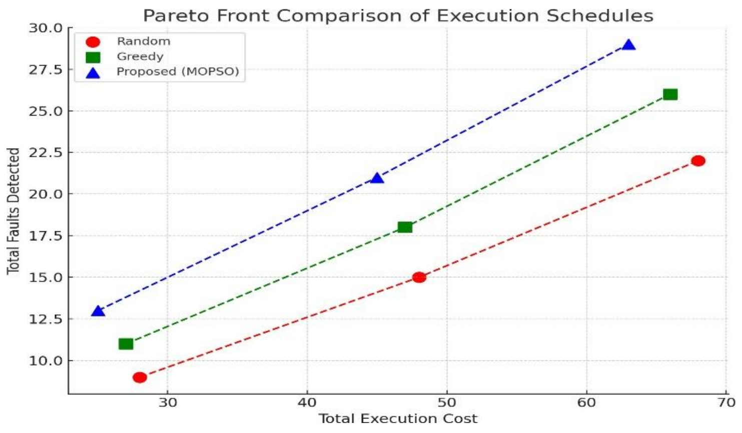

**Fig. 4.** Fault Detection Efficiency vs Budget Level.

In general, the experimental results verify the benefits of multi-objective optimization in adaptive test execution scheduling. The proposed policy has consistently shown good ability to maximize fault detection at a fixed budget better than traditional policies for a diverse set of projects and test suites. These results validate the framework as a practically feasible mechanism for inclusion in contemporary agile and DevOps test pipelines, at which balancing testing efficacy and resource efficiency is essential.

## 6 Conclusion and Future Work

It is suggested to test the design using an up to date budget constraint procedure (BC: the special version of a multi-objective swarm particle initialization technique MOSPT). Through casting the test scheduling as a bi-objective optimization problem, the framework manages to well-tune inspection efficacy with execution efficiency under tight resource limitations. It constructs optimized sequences of execution dynamically, using available resources to maximize the fault detection early.

Experimental results from real datasets showed that the MOPSO-based scheme outperforms the classical random and greedy scheduling in various cases. Under different budget levels, the proposed technique obtained a larger number of faults detected, higher APFDc scores, and better cost effectiveness, as evidenced by in-depth comparative analyses as well as visualizations. The results confirm that the implementation of swarm intelligence in budget-aware regression testing workflows is viable and applicable, notably in the context of agile and DevOps industries.

While the results are very encouraging, a number of avenues for future work are left open. First, the existing approach uses a static budget, while introducing dynamic modification of the budget based on feedback obtained from the real execution environment can potentially improve adaptability. Second, the incorporation of more advanced learning mechanisms (e.g., reinforcement learning) for scheduling would be beneficial to guide particle evolution. Third, the evaluation on larger industrial-scale projects and integration of more diverse cost models such as energy cost and cloud resource utilizations could make the approach more realistic.

On the whole, the key contribution of this work lies in advancing intelligent and resource-efficient RT, through which software developers and industry can maintain rigorous quality assurance (QA) practices  even when faced with tight operational tolerances.

## References

[1] G. Rothermel, R. H. Untch, Chengyun Chu and M. J. Harrold, "Test case prioritization: an empirical study," *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, Oxford, UK, 1999, pp. 179-188, doi: 10.1109/ICSM.1999.792604.

[2] Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification & Reliability, 22*(2), 67–120. https://dl.acm.org/doi/abs/10.1002/stv.430

[3] Elbaum, S., Rothermel, G., & Penix, J. (2014). Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)* (pp. 235–245). Association for Computing Machinery. https://doi.org/10.1145/2635868.2635910

[4] Chen, T. Y., Kuo, F.-C., Merkel, R. G., & Tse, T. H. (2010). Adaptive random testing: The ART of test case diversity. *Journal of Systems and Software, 83*(1), 60–66. https://doi.org/10.1016/j.jss.2009.02.022

[5] Graves, T. L., Harrold, M. J., Kim, J., Porters, A., & Rothermel, G. (1998). An empirical study of regression test selection techniques. In *Proceedings of the 20th International Conference on Software Engineering* (pp. 188–197). IEEE. https://doi.org/10.1109/ICSE.1998.671115

[6] Li, Z., Harman, M., & Hierons, R. M. (2007). Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering, 33*(4), 225–237. https://doi.org/10.1109/TSE.2007.38

[7] Zhang, L., Hou, S.-S., Guo, C., Xie, T., & Mei, H. (2009). Time-aware test-case prioritization using integer linear programming. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis* (pp. 213–224). Association for Computing Machinery. https://doi.org/10.1145/1572272.1572297

[8] Just, R., Jalali, D., & Ernst, M. D. (2014). Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (pp. 437–440). Association for Computing Machinery. https://doi.org/10.1145/2610384.2628055

[9] Coles, H., Laurent, T., Henard, C., Papadakis, M., & Ventresque, A. (2016). PIT: A practical mutation testing tool for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (pp. 449–452). Association for Computing Machinery. https://doi.org/10.1145/2931037.2948707

[10] Gligoric, M., Eloussi, L., & Marinov, D. (2015). Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (pp. 211–222). Association for Computing Machinery. https://doi.org/10.1145/2771783.2771784

[11] Rothermel, G., Untch, R. H., Chu, C., & Harrold, M. J. (2001). Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering, 27*(10), 929–948. https://doi.org/10.1109/32.962562