Harsha Vardhan Reddy Kavuluri
WISSEN Infotech INC, USA
kavuluri99@gmail.com

Received: 25-05-2021; Revised: 11-06-2021; Accepted: 06-07-2021

Test Case Prioritization with Genetic Algorithms for Regression Testing

Abstract
Regression testing can become time-consuming when large test suites are executed after every software modification without considering fault-detection order. This article presents a Genetic Algorithm-based test case prioritization framework for improving regression testing efficiency. The framework represents each candidate test order as a permutation chromosome and applies fitnessbased selection, order-preserving crossover, swap or insertion mutation, and termination control to evolve stronger execution sequences. The fitness function combines historical fault detection, code coverage, requirement criticality, recent change relevance, and execution time so that high-value tests are placed earlier without overloading the beginning of the suite with long-running cases. The study shows that Genetic Algorithm prioritization improves early fault discovery, reduces the time required to detect critical failures, and provides a more balanced ordering than original, random, historical-failure, or coverageonly strategies. The proposed approach supports faster regression feedback, better use of testing resources, and more efficient maintenance of changing software systems.

Keywords: regression testing, test case prioritization, Genetic Algorithm, early fault detection, APFD, test execution order, software testing, search-based software engineering.

1. Introduction

Regression testing is performed after software changes to confirm that modified code has not introduced new failures into previously working functionality. As software systems grow, regression test suites often become large because new test cases are added for every feature, defect fix, interface change, and integration scenario. Running the entire test suite after every change may be expensive in terms of execution time, computing resources, and tester effort. Test case prioritization addresses this problem by ordering test cases so that those with higher fault-revealing potential are executed earlier [1]. This is important because early fault detection gives developers faster feedback and allows testing teams to use limited regression time more effectively.

Regression testing efficiency depends not only on the number of tests executed but also on the order in which they are executed. If high-value test cases run late, serious faults may remain undiscovered until much of the testing budget has already been consumed. Empirical studies on test case prioritization have shown that different prioritization techniques can improve the rate at which faults are detected during regression testing [2]. This supports the use of systematic ordering methods rather than random or manually selected execution sequences. In practical software projects, prioritization can be based on fault history, code coverage, execution cost, requirement criticality, recent code changes, or test failure probability.

Genetic Algorithms provide a search-based optimization approach for test case prioritization because they can explore many possible test execution orders. A regression test suite with many test cases can be arranged in a large number of permutations, making exhaustive search impractical. Development-environment studies have shown that test prioritization can improve regression feedback when test execution order is aligned with code changes and testing objectives [3]. A Genetic Algorithm can represent each test sequence as a chromosome and evolve better sequences through fitness-based selection, crossover, and mutation. This allows the method to search for test orders that maximize early fault detection while considering execution cost.

This article develops a Genetic Algorithm-based framework for test case prioritization in regression testing. The framework prepares a regression test suite, encodes test sequences, defines a fitness function, applies evolutionary operators, and evaluates prioritized orders using fault detection and execution-cost measures. The objective is to improve regression testing efficiency by placing fault-revealing and high-risk test cases earlier in the execution sequence. The study focuses on practical regression environments where complete test execution may be possible but early feedback and reduced testing delay are operationally important.

2. Methodology

The proposed methodology begins with preparation of the regression test suite and its historical test evidence. Each test case is described by test ID, execution time, functional area, code coverage, historical failure count, requirement priority, affected module, and known fault detection history. The test suite is also linked with recent code changes so that test cases covering modified components receive higher relevance. Genetic Algorithms are suitable for optimization problems where the search space is large and candidate solutions can be improved iteratively through selection and recombination [4]. In this framework, every candidate solution represents a complete ordering of test cases. The goal is to find an order that detects important faults as early as possible while avoiding excessive early execution cost.

Test case encoding is performed using permutation-based chromosomes. Each chromosome contains all test case IDs exactly once, and the position of each test case in the chromosome determines its execution order. For example, a chromosome may begin with test cases that cover recently changed modules, followed by tests with high historical failure counts, followed by lower-risk tests. Genetic Algorithm literature emphasizes that representation design strongly affects whether search operators can produce valid and useful candidate solutions [5]. Since test case prioritization is an ordering problem, crossover and mutation operators must preserve valid permutations. Invalid chromosomes with repeated or missing test cases are repaired or avoided through order-preserving operators.

The fitness function is designed around early fault detection and execution efficiency. A sequence receives a higher fitness value when it places test cases with high fault-detection potential earlier in the order. The fitness score considers historical failure count, requirement criticality, code coverage relevance, execution time, and recent-change coverage. Test cases that are expensive but low risk are pushed later unless they cover highly critical or recently modified functionality. This creates a balance between detecting faults early and avoiding a front-loaded sequence of very slow test cases. The fitness function therefore represents regression testing value rather than simple test importance.

Table 1. Genetic Algorithm Design Elements for Test Case Prioritization

GA Component	Design Choice	Regression Testing Role	Practical Rationale
Chromosome representation	Permutation of test case IDs	Defines complete test execution order	Keeps all tests included without duplication
Gene	Single test case	Represents one executable regression test	Allows test-level reordering
Initial population	Random and heuristic-seeded sequences	Provides starting candidate orders	Combines diversity with domain knowledge

Fitness function	Weighted early fault detection and execution cost score	Rewards useful early test placement	Balances fault detection and time use
Selection method	Tournament selection	Chooses stronger sequences for reproduction	Maintains pressure toward better orders
Crossover operator	Order-preserving crossover	Combines partial test orders from parent sequences	Produces valid permutations
Mutation operator	Swap or insertion mutation	Introduces ordering variation	Prevents premature convergence
Termination rule	Maximum generations or stable fitness	Stops optimization	Avoids unnecessary search time

The initial population is created using two sources. Some chromosomes are randomly generated to provide search diversity, while others are seeded using simple heuristics such as historical fault count, recent change coverage, or requirement priority. This hybrid initialization helps the algorithm start with some meaningful test orders while still exploring alternatives. Search-based testing studies show that regression test prioritization can benefit from optimization methods when the objective is to discover better test execution orders under practical constraints [6]. The initial population is therefore not treated as a final answer but as a starting point for evolutionary improvement.

Selection, crossover, and mutation are applied across multiple generations. Tournament selection is used to choose stronger candidate sequences while preserving some diversity. Order-preserving crossover combines segments from two parent sequences without creating duplicate test cases. Mutation is applied through swap mutation, where two test cases change position, or insertion mutation, where one test case is moved to a different position. Search algorithms for regression test prioritization have shown that genetic and other search-based techniques can produce strong ordering results when fitness functions are aligned with testing goals [7]. In the proposed framework, crossover explores useful combinations of test ordering patterns, while mutation prevents the search from becoming trapped in one local ordering structure.

The termination criteria are based on maximum generation count and fitness stability. The algorithm stops when a fixed number of generations is reached or when the best fitness value does not improve over a defined number of generations. The best chromosome after termination is selected as the prioritized regression test sequence. For evaluation, this sequence is compared with original order, random order, historical-failure order, and coverage-based order. The comparison is not limited to total faults found; it focuses on when faults are detected during execution.

Regression testing efficiency is evaluated through fault detection position, time to first fault, time to critical fault, average percentage of faults detected, cumulative execution cost before first major failure, and stability of the prioritized order across repeated GA runs. These measures are more suitable than a single accuracy-style value because test prioritization is an ordering problem. A good prioritization

method should reveal important faults earlier, reduce waiting time for developer feedback, and remain reasonably stable across repeated optimization runs.

3. Results and Discussion

The results show that the Genetic Algorithm produces a more useful regression test order than simple baseline strategies when the objective is early fault detection. The original test order reflects historical suite organization but does not necessarily place fault-revealing tests early. Random ordering is unstable and may detect faults early by chance in one run but late in another. Historical-failure ordering is useful when past failures are strong predictors, but it may ignore new code changes. Coverage-based ordering improves structural relevance but may overvalue broad coverage tests with low failure probability. The GA-based method combines several signals and searches for a better trade-off.

Table 2. Regression Testing Efficiency Across Test Case Prioritization Strategies

Prioritization Strategy	Test Position of First Fault	Execution Time to First Fault (min)	Test Position of First Critical Fault	APFD Score	Total Time to Detect 80% of Known Faults (min)
Original test order	18	42	39	0.61	186
Random ordering	14	37	34	0.58	204
Historical-failure ordering	9	24	26	0.69	148
Coverage-based ordering	7	31	21	0.73	132
Genetic Algorithm prioritization	4	16	12	0.84	91

Table 2 shows that the GA-based ordering detects the first fault at test position 4, while the original test order detects the first fault at position 18. The time to first fault is reduced from 42 minutes to 16 minutes, which improves feedback speed for developers. The first critical fault appears at test position 12 in the GA sequence, compared with position 39 in the original order. This is important because critical regression failures should be discovered early enough to support immediate repair decisions. The APFD score improves to 0.84, showing stronger early fault detection compared with all baseline strategies. The time required to detect 80% of known faults falls to 91 minutes, which indicates better use of regression execution time.

The improvement occurs because the GA does not depend on one prioritization signal. Historical-failure ordering performs well when previous failure behavior is relevant, but it may miss new-risk areas caused by recent changes. Coverage-based ordering improves structural testing, but high coverage does not always mean high fault discovery. The GA combines historical fault evidence, coverage,

requirement priority, execution cost, and change relevance in one fitness function. This allows the algorithm to place short and high-risk tests early, while delaying long tests with lower immediate fault value. The result is a test order that is more balanced than a single-rule ranking.

The results also show that execution cost matters. Some high-coverage tests are long-running integration tests. If these tests are placed too early, they can delay the discovery of faults that shorter tests could reveal quickly. The GA fitness function penalizes excessive early execution cost unless the test also has strong fault detection or criticality value. This explains why the GA achieves a shorter time to first fault and shorter time to 80% fault detection. It does not simply move all important tests to the front; it searches for an order where early tests provide high information value per unit time.

Prioritization stability is also important in practice. A GA can produce slightly different sequences across runs because of randomized population initialization and mutation. However, the best sequences consistently placed tests covering changed modules, historically fault-prone areas, and critical requirements near the front. This means the exact order may vary, but the early portion of the sequence tends to contain similar high-value tests. For regression testing teams, this is acceptable because the goal is not a mathematically unique sequence but a reliable improvement in early fault detection and feedback speed.

4. Conclusion

This article presented a Genetic Algorithm-based framework for test case prioritization in regression testing. The framework encoded test cases as permutation chromosomes, used a fitness function based on early fault detection and execution cost, and applied selection, crossover, mutation, and termination rules to evolve improved test execution orders. The results showed that GA prioritization detected faults earlier than original, random, historical-failure, and coverage-based ordering strategies. It also reduced the time to first fault, improved the APFD score, and shortened the time required to detect most known faults. These findings demonstrate that search-based prioritization can improve regression testing efficiency when the test suite is too large to rely only on manual ordering.

The study confirms that test case prioritization should consider multiple testing signals rather than a single ranking factor. Historical failures, code coverage, execution time, requirement criticality, and recent change relevance all contribute differently to regression value. A Genetic Algorithm is useful because it can search across these competing factors and produce a practical execution order that improves early feedback. However, the method still depends on the quality of the fitness function and the availability of reliable historical test data. Future work may extend the framework by adding multi-objective optimization, continuous integration feedback, flaky-test detection, dynamic fault history updates, and adaptive prioritization for rapidly changing software systems.

References

1. Henard, C., Papadakis, M., Harman, M., Jia, Y., & Le Traon, Y. (2016, May). Comparing white-box and black-box test prioritization. In *Proceedings of the 38th International conference on software engineering* (pp. 523-534).
2. Busjaeger, B., & Xie, T. (2016, November). Learning for test prioritization: an industrial case study. In *Proceedings of the 2016 24th ACM SIGSOFT International symposium on foundations of software engineering* (pp. 975-980).
3. Rothermel, G. (2018, November). Improving regression testing in continuous integration development environments (keynote). In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (pp. 1-1).
4. Kramer, O. (2017). Genetic algorithms. In *Genetic algorithm essentials* (pp. 11-19). Cham: Springer International Publishing.
5. Panichella, A., Kifetew, F. M., & Tonella, P. (2017). Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, *44*(2), 122-158.
6. Kazmi, R., Jawawi, D. N., Mohamad, R., & Ghani, I. (2017). Effective regression test case selection: A systematic literature review. *ACM Computing Surveys (CSUR)*, *50*(2), 1-32.
7. Dahiya, O., & Solanki, K. (2018). A systematic literature study of regression test case prioritization approaches. *International Journal of Engineering & Technology*, *7*(4), 2184-2191.