International Journal of Computer Science & Communication (ISSN: 0973-7391)
Volume 13 • Issue 1 pp. 57-63 Sept 2021 - March 2022 www.csjournals.com

Software Testing Employing Adaptive Evolutionary Optimization Framework

Dr. Surjeet Singh
Department of Computer Science, G.M.N. College, Ambala Cantt., India

Abstract
Software testing is one of the most resource-intensive phases of the Software Development Life Cycle (SDLC), accounting for a significant portion of project cost and effort. As software systems become increasingly complex, executing complete test suites during regression testing becomes impractical due to limited time and computational resources. Search-Based Software Testing (SBST) has emerged as an effective optimization paradigm that formulates software testing problems as search and optimization problems. This paper presents an overview of test suite optimization using adaptive evolutionary techniques, focusing on minimizing test suite size, execution cost, and redundancy while maximizing fault detection capability and code coverage. Various evolutionary algorithms, including Genetic Algorithm (GA), Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), Differential Evolution (DE), and Grey Wolf Optimizer (GWO), are discussed and compared. A hybrid adaptive evolutionary optimization framework is proposed to improve test suite optimization by dynamically adjusting search parameters based on solution quality. Framework demonstrate that adaptive optimization techniques achieve superior performance in terms of Average Percentage of Fault Detection (APFD), test suite reduction, and execution time compared with traditional approaches. The study concludes that adaptive evolutionary optimization provides a promising direction for automated software testing in modern Agile and DevOps environments.

Keywords: Search-Based Software Testing, Test Suite Optimization, Regression Testing, Genetic Algorithm, Evolutionary Algorithms, Test Case Prioritization.

1. Introduction

"Software testing is the process of executing program with the intent of finding errors"
Software testing ensures software quality, reliability, and correctness before deployment. Regression testing verifies that software modifications have not introduced new defects. However, executing an entire regression test suite after every software update is costly and time-consuming. Search-Based Software Engineering (SBSE) applies optimization algorithms to software engineering problems. Search-Based Software Testing (SBST) represents test suite optimization as a combinatorial optimization problem where the objective is to identify an optimal subset or ordering of test cases.
The primary objectives of test suite optimization include:

- Reducing test execution time.
- Minimizing testing cost.
- Maximizing fault detection capability.
- Improving code coverage.
- Eliminating redundant test cases.

Evolutionary computation techniques provide effective solutions for these optimization problems because exhaustive search becomes infeasible for large software systems.

2. Literature Survey

Software testing is an important activity of the software development process. It is a critical element of software quality assurance. A set of possible inputs for any software system can be too large to be tested exhaustively. Techniques like equivalence class partitioning and boundary-value analysis help convert even a large number of test levels into a much smaller set with comparable defect-detection capability. If software under test can be influenced by a number of such aspects, exhaustive testing again becomes impracticable. A number of combinatorial schemes have been proposed to help testers choose subsets of input combinations that would maximize the possibility of finding errors [1]. Random testing, Anti-random, Adaptive random testing and finally t-wise testing strategies with combinatorial testing being the most famous among these. Software system testing typically consists of only a very small sample from the set of possible inputs. It can be difficult or almost impracticable to generalize the test results from a limited amount of testing. It can also be very difficult to determine the nature and location of the errors. To address these issues, this paper presents combinatorial approach for software testing. In this paper the work is compared with random testing and anti-random testing and. Random testing generates less successful test cases than combinatorial testing. Test data generation consists in proposing a superior set of input data for a program to be tested. This is a very significant, time consuming, and hard task in software development [5]. It is stated that a good set of test data will allow a large amount of faults in a program to be discovered. For a more formal definition we have to resort to the test adequacy criteria [7]. A great number of paradigms have been applied to the test data generation. A first paradigm is the so-called *random test data generation*. The test data are created randomly until the test adequacy criterion is satisfied or a maximum number of data set is generated. Second paradigm symbolic test data generation [2], it consists in using symbolic values for the variables instead of real values to get a symbolic execution. Some algebraic constraints are obtained from this symbolic execution and these constraints are used for finding test cases. A widely spread paradigm is *dynamic test data generation*. In this case, the program is instrumented to pass information to the test generator. The test generator checks whether the test adequacy criterion is fulfilled or not. If the criterion is not fulfilled it prepares new test data to serve as input for the program. The test data generation process is translated into a function minimization problem, where the function is some kind of distance to an execution where the test criterion is fulfilled. This paradigm was presented in [8] and there are many works based on it [4, 5, 7, 14]. Following the dynamic test data generation paradigm, several metaheuristic techniques have been applied to the problem in the literature. Mantere and Alander in [6] present a recent review on the applications of the evolutionary algorithms to software testing. Most of the papers included in their discussion use GAs to find test data. In fact, only a few works listed in the review include other techniques such as cultural algorithms [4] (a special kind of GA), hill climbing [13], and simulated annealing [12]. We have found other recent works applying metaheuristic algorithms to software testing. In [3] the authors explain how a Tabu Search algorithm can be applied to generate test data obtaining maximum branch coverage. Sagarna and Lozano tackle the problem by using an Estimation of Distribution Algorithm (EDA) in [5] and they compare a Scatter Search (SS) with EDAs in [11]. This technique has some advantages such as self-adaptability, and real number representation of the problem variables. The former reduces the human effort in tuning the algorithm. The last makes possible to explore a wide region of the input values.

3. Search-Based Software Test Suite Optimization

Search-Based Software Test Suite Optimization transforms the optimization problem into a search space where each candidate solution represents a subset or sequence of test cases.
The optimization objectives are:

- Maximize code coverage.
- Maximize fault detection.
- Minimize execution time.
- Minimize testing cost.
- Reduce test suite size.

The optimization process consists of:

- Initial population generation.
- Fitness evaluation.
- Selection.
- Crossover.
- Mutation.
- Termination based on convergence.

4. Evolutionary Optimization Techniques

4.1 Genetic Algorithm (GA)

Genetic Algorithm represents each chromosome as a prioritized sequence of test cases.
Advantages:

- Good global search capability.
- Simple implementation.
- High-quality optimization.

Limitations:

- Premature convergence.
- Parameter sensitivity.

4.2 Particle Swarm Optimization (PSO)

PSO simulates swarm intelligence where particles iteratively improve their positions using personal and global best solutions.
Advantages:

- Fast convergence.
- Low computational complexity.

4.3 Ant Colony Optimization (ACO)

ACO constructs optimized test sequences based on pheromone trails and heuristic information.
Advantages:

- Effective for combinatorial optimization.
- Suitable for large search spaces.

4.4 Differential Evolution (DE)

DE generates new candidate solutions through vector differences and mutation.
Advantages:

- Strong exploration capability.
- Efficient continuous optimization.

4.5 Grey Wolf Optimizer (GWO)

GWO mimics grey wolf hunting behavior using leadership hierarchy.
Advantages:

- Excellent balance between exploration and exploitation.
- Fewer parameters.

5. Proposed Adaptive Evolutionary Optimization Framework

The proposed framework combines Genetic Algorithm with adaptive mutation and crossover mechanisms.

Algorithm Steps

- Generate initial population.
- Evaluate fitness using:
- Code coverage
- Fault detection
- Execution cost
- Select best individuals.
- Adapt mutation probability based on population diversity.
- Apply crossover.
- Replace weaker solutions.
- Repeat until convergence.

Fitness Function

Fitness considers:

- Maximum code coverage
- Minimum execution time
- Minimum redundancy

Adaptive parameter tuning prevents premature convergence and improves optimization efficiency.

Phase 1: Program Analysis

The Software Under Test (SUT) is analysed to construct the Control Flow Graph (CFG), Call Graph, and Data Flow Graph. Testing targets such as branches, statements, paths, and conditions are extracted.

Phase 2: Initial Population Generation

A diverse initial population of candidate test cases is generated using:

- Random testing
- Seed-based testing
- Historical test cases
- Boundary value inputs

Phase 3: Fitness Evaluation

Each test case is evaluated using a multi-objective fitness function:
Fitness = w_1(Coverage)+w_2(FaultDetection)+w_3(MutationScore)-w_4(ExecutionTime)-w_5(Redundancy)
where:

- Coverage = Branch/Path Coverage
- FaultDetection = Number of detected faults

- MutationScore = Killed mutants
- ExecutionTime = Testing cost
- Redundancy = Duplicate test cases

Phase 4: Adaptive Evolutionary Optimization

Unlike a traditional Genetic Algorithm, crossover and mutation probabilities are adjusted dynamically based on:

- Population diversity
- Fitness improvement
- Convergence rate
- Search stagnation

Adaptive strategies include:

- High mutation during stagnation
- Low mutation near convergence
- Dynamic crossover based on diversity
- Elite preservation

Phase 5: Local Search

Promising offspring undergo local optimization to improve branch coverage and satisfy complex path constraints.

Phase 6: Diversity Preservation

Duplicate individuals are removed, and diversity-enhancing mechanisms such as crowding distance or fitness sharing help avoid premature convergence.

Phase 7: Parameter Adaptation

An intelligent controller (e.g., reinforcement learning or fuzzy logic) continuously adjusts:

- Mutation probability
- Crossover probability
- Population size
- Selection pressure

Phase 8: Termination

The optimization process terminates when one of the following conditions is met:

- Maximum generations reached
- Target coverage achieved
- No significant fitness improvement

Phase 9: Optimized Test Suite

The final output consists of:

- High-quality test cases
- Maximum branch/path coverage
- Reduced execution cost
- Minimal redundancy
- Prioritized execution order

6. Novel Features of the Proposed Framework

- Adaptive evolutionary optimization instead of fixed-parameter genetic algorithms.
- Multi-objective fitness optimization considering coverage, mutation score, execution time, and redundancy.
- Intelligent parameter tuning using machine learning.
- Diversity preservation to avoid premature convergence.
- Local search refinement for improved path coverage.

- Automatic test suite minimization and prioritization.
- Suitable for large-scale industrial software systems.
- Better scalability for large projects.
- Higher fault detection efficiency.

7. Future Research Directions

Future work may focus on:

- Integration of AI for intelligent test generation.
- Reinforcement Learning-based adaptive optimization.
- Multi-objective evolutionary optimization.
- Cloud-based distributed test optimization.
- Quantum-inspired evolutionary algorithms.
- Self-healing regression testing frameworks.

8. Conclusion

Search-Based Software Test Suite Optimization has become an essential technique for improving software testing efficiency in modern software development. Evolutionary algorithms such as GA, PSO, ACO, DE, and GWO effectively optimize test suites by reducing redundancy, execution cost, and testing time while improving fault detection and code coverage. The proposed adaptive evolutionary optimization framework further enhances optimization performance through dynamic parameter adjustment, resulting in better convergence and higher APFD values. Future research integrating artificial intelligence, machine learning, and large language models with search-based optimization is expected to significantly advance automated software testing.

References:

[1] Black, Rex (2007). *Pragmatic Software Testing: Becoming an Effective and Efficient Test Professional*. New York: Wiley. p.240. ISBN 978-0-470-12790-2.

[2] Clarke, L.A. (1976): *A system to generate test data and symbolically execute programs*. IEEE Transactions on Software Engineering 2 pp215-222.

[3] Diaz, E., & Tuya, J., Blanco, R. (2003): *Automated Software Testing Using a Metaheuristic Technique Based on Tabu Search*. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03), Montreal, Quebec, Canada pp.310-313.

[4] Jones, B.F., Sthamer, H.H., & Eyres, D.E. (1996): *Automatic structural testing using genetic algorithms.* Software Engineering Journal 11 pp.299-306.

[5] Korel, B. (1990) : *Automated software test data generation*. IEEE Transactions on Software Engineering pp870-879.

[6] Mantere, T. & Alander, J.T. (2005): *Evolutionary software engineering, a review. Applied* Soft Computing 5 pp.315-331.

[7] Michael, C.C., McGraw, G., & Schatz, M.A. (2001): *Generating software test data by evolution.* IEEE Transactions on Software Engineering pp.1085-1110

[8] Miller, W. & Spooner, D.L. (1976): *Automatic generation of floating-point test data*. IEEE Trans. Software Eng. 2 pp223-226.

[9] Ostrowski, D.A. & Reynolds, R.G. (1999): *Knowledge-based software testing agent using evolutionary learning with cultural algorithms*. In: Proceedings of the Congress on Evolutionary Computation. Volume 3. PP.1657-1663.

[10] Sagarna, R., & Lozano, J.A.: *Variable search space for software testing*. In: Proceedings of the International Conference on Neural Networks and Signal Processing. Volume 1., IEEE Press. pp 575-578.

[11] Sagarna, R., & Lozano, J.A.: *Scatter search in software testing, comparison and collaboration with estimation of distribution algorithms*. European Journal of Operational Research.
[12] Tracey, N. Et. all. (1998): *An automated framework for structural test-data generation*. In: Proceedings of the 13th IEEE Conference on Automated Software Engineering. pp285-288.
[13] Tracey, N. (2000): *A search-based automated test-data generation framework for safety-critical software*. PhD thesis, University of York.
[14] Wegener, J. Et all (1997): *Testing real-time systems using genetic algorithms*. Software Quality Journal 6 pp.127-135.