DOI: https://doi.org/10.5281/zenodo.16944404

Exploring the effectiveness of differential evolution for maximizing coverage in automatic testing

Explorando la efectividad de la evolución diferencial para maximizar la cobertura en pruebas automáticas

Saúl Domínguez-Isidro[1]

Resumen: Las pruebas de software basadas en búsqueda (SBST) pretenden optimizar la generación de casos de prueba mediante metaheurísticas, maximizando métricas de calidad como la cobertura de código. Este estudio examina la eficacia de la evolución diferencial (ED) en la generación automática de conjuntos de pruebas que maximizan la cobertura. Se diseñó un estudio empírico exploratorio, utilizando funciones basadas en Python para evaluar el rendimiento de la ED. Los resultados obtenidos destacan el potencial de la ED en comparación con el recocido simulado (SA), demostrando mejoras tanto en la cobertura alcanzada como en el número de evaluaciones de funciones objetivo necesarias.

Palabras clave: pruebas de software basadas en búsqueda, evolución diferencial, cobertura

Abstract: Search-Based Software Testing (SBST) aims to optimize test case generation through metaheuristics, maximizing quality metrics such as code coverage. This study examines the effectiveness of Differential Evolution (DE) in generating test suites that maximize coverage automatically. An exploratory empirical study was designed, utilizing Python-based functions to evaluate the performance of DE. The obtained results highlight DE's potential compared to Simulated Annealing (SA), demonstrating improvements in both achieved coverage and the number of objective function evaluations required.

Keywords: search-based software testing, differential evolution, coverage

List of Acronyms

- SBST: Search-based software testing
- DE: Differential evolution
- SA: Simulated annealing
- SUT: System under test
- GA: Genetic algorithm

1 Facultad de Estadística e Informática, Universidad Veracruzana, Xalapa, México: sauldominguez@uv.mx; ORCID: 0000-0002-9546-8233

- PSO: Particle swarm optimization
- ACO: Ant colony optimization
- FEs: Fitness function evaluations
- NP: Population size (DE parameter)
- CR: Crossover probability
- F: Scaling factor
- UML: Unified modeling language

Introduction

Software testing plays a fundamental role in software development, aiming to detect faults, evaluate performance, and assure the reliability and quality of the final product. During this phase of the development cycle, emphasis is placed on formulating test case suites to verify software compliance with predefined requirements and correct functionality across various scenarios. Despite its crucial role, software testing typically requires significant time and resources, posing challenges particularly in extensive projects or complex systems, where managing the balance between constraints, essential requirements, and the inherent imprecision of these requirements is vital (Harman, Mansouri, & Zhang, 2001).

Over time, test design has evolved significantly, driven by emerging tools and techniques that have radically transformed the testing landscape. Prominent among these are automation and optimization methods. Tools such as Selenium (2023), widely used for test design and management, and optimization approaches employed in Search-Based Software Engineering (SBSE) have become essential (Harman & Jones, 2001). SBSE does not necessarily seek definitive solutions but instead employs metaheuristic algorithms to generate optimal or near-optimal test suites, effectively automating and optimizing test processes (Harman, Jia, & Zhang, 2015).

Metaheuristics employed within SBST can be categorized as traditional single-point methods, such as Simulated Annealing (SA), population-based evolutionary algorithms, such as Genetic Algorithms (GA) and Differential Evolution (DE), and population-based swarm intelligence algorithms (Kennedy and Eberhart, 2001), such as Particle Swarm Optimization (PSO) and Ant Colony Optimization (ACO). These techniques explore the solution space efficiently, optimizing objectives like code coverage, test case reduction, and fault detection (Hernández-Suárez, 2024). DE (Storn and Price, 1997) has gained significant attention due to its straightforward structure, robustness, and efficacy in handling continuous optimization problems. These characteristics make DE suitable for automatically generating test suites that maximize code coverage. In order to evaluate DE's potential, specifically in test suite generation, an exploratory empirical study was designed using Python-based functions. Results indicate DE's superior performance

compared to SA, reinforcing its capability to enhance code coverage significantly and thus emphasizing its suitability for SBST.

The structure of this document is as follows: Section 2 describes the problem statement; Section 3 presents related works; Section 4 explains the proposed approach; Section 5 presents the experimental design and results, while Section 6, the discussion; finally, conclusions and future work are outlined in Section 7.

Problem statement

Code coverage is a metric used in software testing to measure the extent to which the source code of a program is executed when a test suite runs (Myers, Sandler, & Badgett, 2011). It provides quantitative insight into how thoroughly the software has been tested, identifying areas of code that have not been exercised. Code coverage can be calculated by dividing the number of code elements (e.g., statements, branches, conditions) executed by the tests by the total number of elements in the software (Ammann & Offutt, 2016). Higher coverage typically correlates with a lower likelihood of undiscovered defects, thereby enhancing software quality and reliability (Zhu, Hall, & May, 1997).

The coverage problem in software testing involves selecting test cases that collectively execute the largest possible proportion of a software's code elements (branches, statements, conditions). Formally, the coverage problem can be expressed as an optimization problem.

Let T be the set of potential test cases, *C(t)* represent the set of code elements covered by the test case *t*, and E be the set of all code elements in the software under test. The problem consists of selecting a subset $T' \subseteq T$ such as the proportion of covered code elements, defined as Eq 1.

$$maximize: \quad \frac{\left|\bigcup_{t \in T'} C(t)\right|}{|E|} \tag{1}$$

where *U* denotes the union of all unique code elements covered by the selected test cases *T'*.

Related works

DE has demonstrated effectiveness in generating high-quality test cases due to its simplicity, robustness, and capability to handle continuous search spaces.

Varshney and Mehrotra (2016) proposed a DE-based approach for automated test data generation to maximize data-flow coverage. Their study designed a fitness function leveraging dominance relations and branch distance metrics to guide the search process. Experimental results showed that their DE-based approach outperformed Random Search, GA, and PSO regarding average coverage and number of generations, particularly for benchmark programs where data-flow coverage is critical (Varshney & Mehrotra, 2016). Panda et al. (2020) developed a hybrid metaheuristic framework combining the Firefly Algorithm (FA) with DE for test suite generation in object-oriented programs. The approach utilized UML behavioral state chart models to derive feasible test sequences and then applied the hybrid FA-DE algorithm for optimization. Their results demonstrated improved performance over individual FA and DE implementations, achieving better exploration, exploitation, and coverage of transition paths in model-based testing scenarios (Panda et al., 2020).

Pietrantuono and Russo (2018) analyzed search-based optimization techniques applied to the Testing Resource Allocation Problem (TRAP). Their study provided an overview of metaheuristic methods, including DE, employed to optimize resource distribution among software components under constraints such as cost and reliability. While not directly focused on test case generation, their work highlighted the versatility and applicability of search-based methods, including DE, in broader software testing contexts (Pietrantuono & Russo, 2018). Nguyen et al. (2016) investigated exploration-focused techniques to enhance SBST. They experimentally evaluated strategies designed to maximize behavioral diversity during test generation. Their findings emphasized the importance of balancing exploration and exploitation in SBST, a challenge where DE's adaptive mechanisms can offer advantages (Nguyen et al., 2016).

Despite these contributions, most prior studies have focused on either hybrid metaheuristic frameworks, specific test adequacy criteria such as data-flow coverage, or resource allocation problems in testing. Much prior work has also emphasized comparisons between DE and population-based evolutionary algorithms like GA or swarm-based approaches like PSO and ACO. This study differs from previous works in that it conducted an exploratory empirical evaluation focused on differential evolution (DE) as a test suite generation technique to maximize code coverage. Unlike prior studies emphasizing data flow or path coverage, this work addresses statement and branch coverage as primary metrics, which are fundamental and widely adopted in white-box testing.

Furthermore, this study compares DE against SA, a single-solution metaheuristic known for its conceptual simplicity and low computational complexity regarding operators and memory requirements. The choice of SA as a baseline is grounded in two key reasons. First, SA was one of the earliest metaheuristic algorithms applied to SBST, and it has historically served as a foundational method for automated test generation. Second, comparing DE to SA highlights the advantages of population-based strategies over

more straightforward, trajectory-based methods regarding exploration capabilities and solution quality. By contrasting DE with a more basic metaheuristic, this work clarifies the trade-offs between algorithmic complexity and performance, particularly regarding coverage achieved and the number of fitness function evaluations required.

Proposed approach

This section presents the approach employed to generate test suites using Differential Evolution (DE) for maximizing code coverage in the system under test (SUT). The methodology consists of four components: the problem definition, solution representation, fitness function, and the DE algorithm.

System Under Test (SUT)

The goal of this study is to generate test suites that maximize code coverage in a given SUT. The SUT, modeled as a decision logic function, determines which code branches are exercised based on input variables. The problem is formalized as an optimization task:

$$\text{decision\_logic(x ,y, z)} = \begin{cases} \text{Branch A} & \text{if } x > 10 \land y < 5 \land z = 0 \\ \text{Branch B} & \text{if } x > 10 \land y < 5 \land z \neq 0 \\ Branch\ C & if\ x > 10 \land y \geq 5 \\ Branch\ D & if\ x > 10 \land y > 20 \\ Branch\ E & if\ x > 10 \land y \leq 5 \end{cases} \quad (2)$$

Optimization problem

Minimize:

$$f(\vec{x}) = -\left| \bigcup_{i=1}^{N} C(t_i) \right| \quad (3)$$

where: $\vec{x} = [t_1, t_2, \ldots, t_N]$ represents a test suite composed of *N* test cases. $C(t_i)$ is the set of code branches covered by test case t_i. The objective is to maximize the number of unique branches covered by the test suite.

subject to: $x_i \in [0,30]; y_i \in [0,30]; z_i \in [0,5]$ for $i = 1,\ldots,N$

Solution representation

Each candidate solution (individual) represents a test suite. A test suite consists of *N*= 5 test cases. Each test case is composed of three input variables: *x,y,* and *z*. Therefore, each individual is represented as a real-valued vector of dimension *D* = 15.

Example representation

$$\text{Individual } \overrightarrow{X} = [x_1, y_1, z_1, x_2, y_2, z_2, \dots, x_5, y_5, z_5]$$

Each x_i,y_i , and z_i are constrained within the ranges defined in the problem domain

$x_i \in [0,30]; y_i \in [0,30]; z_i \in [0,5]$

Fitness function

The fitness function evaluates the quality of a test suite based on the number of unique code branches covered when executing the SUT with the test cases in the suite.

Objective function

Maximize:

$$coverage(\vec{X}) = \left| \bigcup_{i=1}^{N} C(t_i) \right| \quad (3)$$

Differential evolution algorithm

The complete DE algorithm follows the standard DE/rand/1/bin strategy. The pseudocode for the implementation is shown in algorithm 1.

Algorithm 1: Differential evolution (DE/rand/1/bin)

1. Randomly generate an initial population of NP vectors $P_0 = \{X_{0,1}, \dots, X_{0,NP}\}$
2. Evaluate the fitness of each individual in the initial population.
3. Repeat until the stop condition is met (maximum iterations or full coverage):
   - For each individual $i = 1, \dots, NP$:
     i. Randomly select $r1, r2, r4 \in \{1, \dots, NP\}$ with $r1 \neq r2 \neq r4 \neq i$
     ii. Generate a mutant vector $V_i = X_{r1} + F(X_{r2} - X_{r3})$
     iii. Apply binomial crossover to produce a trial vector U_i

          For each dimension $j = 1, \dots, D$:

          $$U_{i,j} = \begin{cases} V_{i,j} & if\ rand_j \leq CR\ or\ j = j_{rand} \\ X_{i,j} & otherwise \end{cases}$$

     iv. Evaluate U_i
     v. Apply selection

     $$X_i = \begin{cases} U_i & if\ coverage(U_i) \geq coverage(U_i) \\ X_i & otherwise \end{cases}$$

   - Update the best-so-far solution and coverage.
4. Terminate when the maximum number of iterations is reached or when full coverage is obtained.

Implementation details

The algorithms and fitness functions were implemented in Python. The implementation included:

- A fitness function to measure the number of unique branches covered by the test suite.
- A System Under Test (SUT) modeled as a deterministic decision logic function with 8 branches.
- Custom implementations of DE and SA, utilizing vector operations and stochastic components.
-

The development relied on the following Python libraries:

- NumPy for vector and numerical operations.
- random for stochastic behaviors and seed control.
- Matplotlib for generating sensitivity analysis plots.
- time for internal execution measurements.

All experiments were conducted using reproducible random seeds and can be reproduced upon request.

Experimental design and results

This section presents the exploratory empirical study conducted to evaluate the performance of Differential Evolution (DE) as a technique for automated test suite generation aimed at maximizing code coverage. The experimental design is structured in two phases: i) Sensitivity Analysis of DE Parameters and ii) Comparative Analysis with Simulated Annealing (SA), which focuses on convergence speed and coverage effectiveness.

The cause for this experimental design is twofold. First, sensitivity analysis allows the identification of appropriate parameter settings for DE in the context of SBST. Understanding parameter sensitivity is essential for determining how parameter variations influence the algorithm's performance in this specific problem domain. Second, by comparing DE to Simulated Annealing (SA), a simpler single-solution metaheuristic and one of the first algorithms applied in SBST, it is possible to assess the relative advantages of DE in terms of convergence speed (evaluated by fitness function evaluations or FEs) and final coverage achieved.

Sensitivity analysis

The Differential Evolution (DE) algorithm parameters were selected based on systematic experimentation. Various configurations were tested by varying the parameters individually, and the combination that provided the best trade-off between exploration,

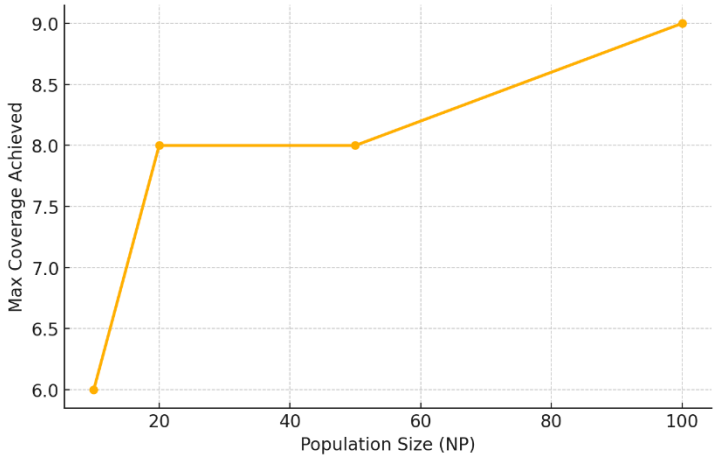
exploitation, and convergence speed was chosen. The selected parameters and their corresponding justifications are presented in table 1.

Table 1. Parameter settings for DE

Parameter	Value	Justification / impact
Population Size (NP)	20	A moderate population size balances exploration and computational cost. Larger populations improve diversity but increase evaluations.
Scaling Factor (F)	0.2	A low scaling factor limits the magnitude of mutation steps, promoting fine-tuned search and avoiding disruptive changes.
Crossover Probability (CR)	0.9	A high crossover probability encourages diversity by incorporating components from mutant vectors more frequently.
Maximum Iterations	100	Provides sufficient opportunity for convergence while maintaining reasonable computational time.
Dimensionality (D)	15	Corresponds to 5 test cases × 3 input variables. This defines the search space dimensionality for the test suite generation.

The systematic experimentation phase varied each parameter within typical ranges suggested in the literature (Storn & Price, 1997). To identify the most effective configuration for test suite generation and code coverage maximization, sensitivity analyses were conducted for key parameters, as shown in Figures 1 to 3.

Figure 1. Sensitivity analysis of population size (NP)



Increasing NP improved coverage up to a point, with decreasing returns observed beyond NP = 50. A value of NP = 20 was selected to balance diversity and computational cost (see Figure 1). Optimal performance was achieved with F=0.2, enabling fine-tuned adjustments without excessive randomness. Lower or higher values resulted in suboptimal coverage (see Figure 2).

A high crossover rate (CR=0.9) facilitated exploration, leading to better coverage compared to lower rates, while avoiding instability observed at CR=1.0 (see Figure 3).

Figure 2. Sensitivity analysis of scaling factor (F)

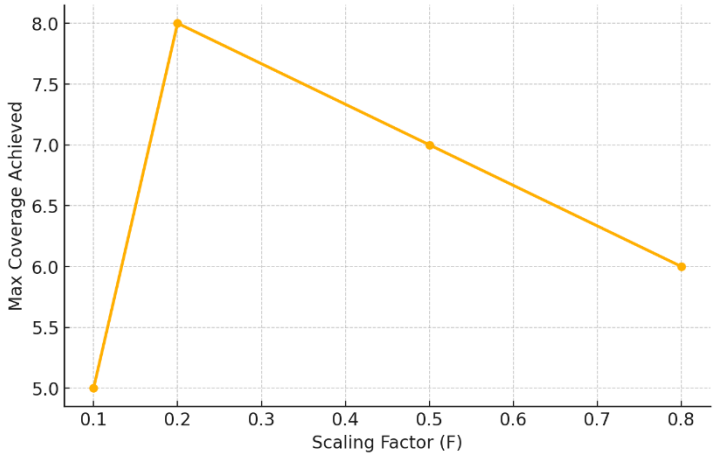
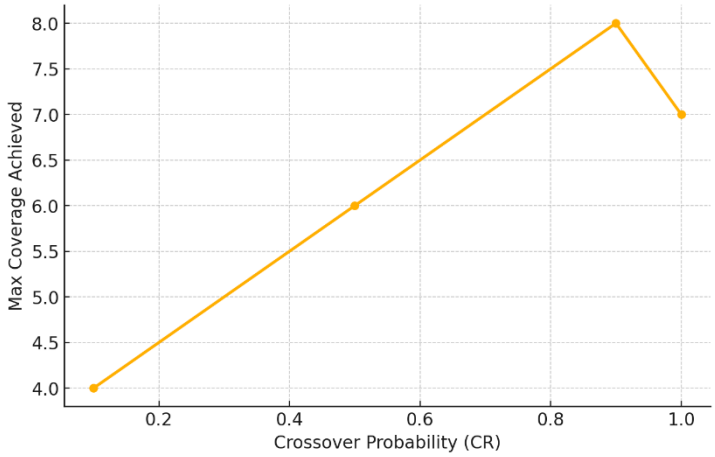


Figure 3. Sensitivity analysis of crossover probability (CR)



Convergence speed analysis

The second experiment assesses the convergence speed of DE in comparison to SA, focusing on the number of fitness function evaluations (FEs) required to achieve competitive or optimal solutions. FEs are a standard metric for evaluating algorithmic efficiency in optimization tasks, as they directly represent the computational cost associated with invoking the fitness function during the search process. One FE corresponds to a single invocation of the fitness function within the optimization loop. This metric is particularly relevant when comparing population-based and single-point metaheuristics, as population-based algorithms typically perform NP times more FEs per iteration than single-point techniques like SA.

This study executed DE and SA under equivalent stopping criteria, with a maximum of 100 iterations and the same evaluation limits. The objective was to analyze the

speed at which each algorithm converges toward optimal or near-optimal test suites, measured in terms of the number of FEs required.

To ensure the robustness and statistical validity of the results, 30 independent runs were performed for each algorithm. Each run was initialized with a distinct random seed to account for the algorithms' stochastic nature and reduce potential biases in the evaluation.

Table 2. Descriptive statistics for fitness function evaluations (FEs)

Criteria	DE	SA
Maximum	2020	1801
Minimum	80	324
Mean	664.83	1750.07
Median	300	1801
Mode	2020	1801
Standard Deviation	786.85	274.27

Table 2 demonstrates apparent differences in the convergence behavior of both algorithms. In 30 runs, DE always required fewer fitness function evaluations (FEs) to achieve competitive solutions than SA. Specifically, DE exhibited a mean of 664.83 FEs and a median of 300 FEs, indicating that, in most runs, it was able to converge toward optimal or near-optimal solutions well before reaching the maximum iteration limit. This is further supported by the minimum of 80 FEs, which reflects DE's ability to rapidly find high-quality solutions in favorable scenarios.

On the other hand, SA required more evaluations, with a mean of 1750.07 FEs and a median of 1801, corresponding to its maximum limit. The fact that SA's mode is also 1801 suggests that in most runs, the algorithm exhausted its evaluation budget without converging early, highlighting its slower convergence rate. The lower standard deviation (274.27) in SA FEs indicates more consistent but slower convergence behavior across runs.

On the other hand, DE showed more significant variability in the number of evaluations (standard deviation of 786.85) due to its adaptive search dynamics. It reflects a range of convergence speeds and demonstrates DE's potential to converge early, reducing computational cost when successful. This analysis shows that DE converges faster on average and demonstrates more flexibility in balancing exploration and exploitation, making it a more efficient option for test suite generation in this experimental context.

Coverage performance comparison

The final experiment evaluates the effectiveness of DE and SA in maximizing code coverage. The target problem considered in this study has a theoretical maximum

coverage of 8, which corresponds to the total number of distinct branches defined within the System Under Test (SUT). This upper bound was determined through structural analysis of the SUT's decision logic, where 8 unique decision branches were identified. Full coverage is achieved when all these branches are exercised by the generated test suite.

The descriptive statistics for code coverage obtained from 30 independent runs of each algorithm are summarized in Table 3.

Table 3. Descriptive statistics for code coverage

Criteria	DE	SA
Maximum	8	8
Minimum	7	2
Mean	7.76	6.07
Median	8	6
Mode	8	7
Standard Deviation	0.44	1.28
Optimal Coverage Achieved (8)	22 / 30 runs	1 / 30 runs

The results show that DE significantly outperforms SA in terms of coverage quality. DE achieved the optimal coverage in 22 out of 30 runs, representing 73.3% of the executions. On the other hand, SA reached the optimal coverage only once across all runs. This substantial difference highlights DE's superior capability to consistently generate test suites that exercise all available branches in the SUT.

Moreover, DE exhibits a higher average coverage (7.76) and a median of 8, which indicates that most of its runs achieved full coverage or came very close to it. The mode of 8 further confirms that full coverage was the most frequently observed outcome for DE. Its low standard deviation (0.44) suggests a high level of consistency in its performance across runs, with minimal variation in coverage.

Conversely, SA shows a lower mean coverage of 6.07, with a median of 6, implying that in at least half of its runs, it failed to cover two or more branches. Although SA managed to achieve a maximum coverage of 8 in one instance, its minimum coverage dropped to 2, underscoring its inability to consistently generate effective test suites. The standard deviation of 1.28 indicates greater variability in SA's performance, reflecting inconsistency in its capacity to explore the search space and maximize coverage.

The coverage analysis demonstrates that DE is more effective in achieving higher coverage, and more reliable and consistent across multiple independent runs. These findings reinforce the suitability of DE as a robust approach for maximizing code coverage in automated test suite generation.

Experimental setup

All experiments were executed on the following hardware:
- Processor: AMD Ryzen 7 5800H (8 cores, 16 threads)
- RAM: 8 GB DDR4
- Operating System: Windows 11
- Python Version: 3.12.4

Each independent run of DE and SA took approximately 5 to 15 seconds, depending on convergence speed. A whole batch of 30 runs per algorithm was completed in under 10 minutes, demonstrating the efficiency of DE for test suite generation in SBST contexts.

Discussion

The results obtained from the experiments provide clear evidence of DE advantages over SA in the context of SBST. The convergence speed analysis demonstrated that DE requires significantly fewer fitness function evaluations (FEs) to achieve competitive or optimal solutions. This suggests that DE can reduce computational costs by converging faster, even in scenarios involving complex search spaces and multiple test case parameters.

In addition to convergence speed, DE outperformed SA regarding code coverage. DE consistently achieved or approached the theoretical maximum coverage of the SUT, with minimal variation across multiple independent runs. This reliability is crucial in the automated generation of test suites, where consistency and high coverage are essential to ensure effective fault detection and verification of software behavior.

The sensitivity analysis provided valuable insights into the impact of DE control parameters on its performance. The results confirmed that parameter tuning is critical for balancing exploration and exploitation within the search process. Specifically, the selected combination of a low scaling factor (F=0.2) and a high crossover probability (CR=0.9) enabled DE to maintain a fine-grained search capability while ensuring sufficient diversity in the population.

By contrast, SA showed limited capacity to adapt its search trajectory, frequently exhausting its evaluation budget without achieving high coverage. While SA convergence behavior was more consistent regarding FEs, its overall effectiveness in maximizing code coverage was significantly lower than DE.

In summary, the empirical evidence supports the conclusion that DE is better suited than SA for automated test suite generation tasks to maximize code coverage. DE population-based search strategy, adaptability, and robustness make it a compelling choice for SBST applications.

Conclusions and future works

This study explored Differential Evolution (DE) as an automated test suite generation optimization within Search-Based Software Testing (SBST). The primary objective was to maximize code coverage in the System Under Test (SUT), and the experimental results provide solid evidence supporting DE's effectiveness in achieving this goal.

The experiments began with a sensitivity analysis to identify the optimal configuration of DE parameters. Through systematic experimentation, it was determined that a population size of 20, a scaling factor of 0.2, and a crossover probability of 0.9 provided a good balance between exploration and exploitation. These settings enabled DE to navigate the search space effectively and generate high-quality test suites.

Once configured, DE was compared to Simulated Annealing (SA), a classic single-solution metaheuristic historically applied in SBST tasks. The comparison focused on two critical aspects: convergence speed, measured by the number of fitness function evaluations (FEs), and the quality of the solutions, measured by the achieved code coverage.

The results demonstrated that DE consistently outperformed SA. Regarding convergence speed, DE required significantly fewer FEs to reach competitive or optimal solutions. In several cases, DE converged early, reducing the computational cost associated with the test suite generation process. On the other hand, SA frequently exhausted its evaluation budget without achieving high coverage, reflecting its slower convergence rate.

Regarding coverage quality, DE proved to be more effective and reliable. It reached the theoretical maximum coverage of the SUT in 73.3% of the runs and consistently produced test suites that covered nearly all available branches. In contrast, SA exhibited lower coverage on average, higher variability in its results, and a limited ability to explore the search space effectively.

Overall, this study's findings confirm that Differential Evolution is a robust and efficient approach for automated test suite generation aimed at maximizing code coverage. Its adaptability, convergence behavior, and consistent performance make it a suitable candidate for addressing optimization problems in SBST.

Future work will extend this approach to other coverage criteria, integrate DE with hybrid metaheuristics, explore dynamic parameter adaptation techniques, and conduct experiments on larger and more complex systems. Additionally, incorporating fault detection metrics as optimization objectives will be considered to enhance the evaluation of the generated test suites' effectiveness.

References

Ammann, P., & Offutt, J. (2016). Introduction to software testing. Cambridge University Press.

Feldt, R., & Poulding, S. (2015). Broadening the Search in Search-Based Software Testing: It Need Not Be Evolutionary. IEEE/ACM 8th International Workshop on Search-Based Software Testing (SBST). https://doi.org/10.1109/SBST.2015.8

Harman, M., Jia, Y., & Zhang, Y. (2015). Achievements, open problems, and challenges for search-based software testing. IEEE Transactions on Software Engineering, 41(6), 544-566. https://doi.org/10.1109/TSE.2014.2360629

Harman, M., & Jones, B. F. (2001). Search-based software engineering. Information and Software Technology, 43(14), 833-839.

Harman, M., Mansouri, S. A., & Zhang, Y. (2001). Search-based software engineering: Trends, techniques and applications. ACM Computing Surveys, 45(1), 1-61.

Hernández-Suárez, J. E., Jiménez-Jiménez, M. A., Domínguez-Isidro, S., & Ocharán-Hernández, J. O. (2024). Search-based software testing driven by bioinspired algorithms: A systematic literature mapping. In 2024 12th International Conference in Software Engineering Research and Innovation (CONISOFT) (pp. 101–108). IEEE. https://doi.org/10.1109/CONISOFT63288.2024.00023

Kennedy, J., & Eberhart, R. C. (2001). Swarm intelligence. San Francisco, CA: Morgan Kaufmann Publishers.

Myers, G. J., Sandler, C., & Badgett, T. (2011). The art of software testing. John Wiley & Sons.

Nguyen, T. H., et al. (2016). Using Exploration Focused Techniques to Augment Search-Based Software Testing: An Experimental Evaluation. IEEE International Conference on Software Testing, Verification and Validation (ICST). https://doi.org/10.1109/ICST.2016.26

Panda, M., Dash, S., Nayyar, A., Bilal, M., & Mehmood, R. M. (2020). Test Suit Generation for Object Oriented Programs: A Hybrid Firefly and Differential Evolution Approach. IEEE Access, 8, 179167-179181. https://doi.org/10.1109/ACCESS.2020.3026911

Pietrantuono, R., & Russo, S. (2018). Search-Based Optimization for the Testing Resource Allocation Problem: Research Trends and Opportunities. IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST). https://doi.org/10.1145/3194718.3194721

Selenium. (2023). Selenium WebDriver documentation. Retrieved from https://www.selenium.dev/documentation/

Storn, R., & Price, K. (1997). Differential evolution – A simple and efficient heuristic for global optimization over continuous spaces. Journal of Global Optimization, 11(4), 341–359. https://doi.org/10.1023/A:1008202821328

Varshney, S., & Mehrotra, M. (2016). A Differential Evolution based Approach to Generate Test Data for Data-Flow Coverage. International Conference on Computing, Communication and Automation (ICCCA). IEEE. https://doi.

org/10.1109/ICCCA.2016.7813770

Zhu, H., Hall, P. A. V., & May, J. H. R. (1997). Software unit test coverage and adequacy. ACM Computing Surveys, 29(4), 366-427.