Indonesian Journal of Electrical Engineering and Computer Science
Vol. 43, No. 1, July 2026, pp. 233~249
ISSN: 2502-4752, DOI: 10.11591/ijeecs.v43.i1.pp233-249

Auto-generated unit testing using PCA-aDynaMOSA

Made Raja Adi Surya Saputra[1], Maria Seraphina Astriani[2]
[1]Department of Computer Science, Faculty of Computer Science, BINUS Graduate Program – Master of Computer Science, Bina Nusantara University, Jakarta, Indonesia
[2]Department of Computer Science, School of Computing and Creative Arts, Bina Nusantara University, Jakarta, Indonesia

Article Info

Article history:

Received Aug 2, 2025
Revised May 29, 2026
Accepted Jun 27, 2026

Keywords:

aDynaMOSA
Genetic algorithm
Multi-objective optimization
Principal component analysis
Unit test

ABSTRACT

Automated test case generation is essential for improving software quality; however, many-objective search-based testing approaches often experience scalability issues when the number of test objectives increases. This condition leads to slower convergence, higher computational effort, and reduced ability to cover complex program structures. To address this gap, this study proposes an enhanced version of the aDynaMOSA algorithm by incorporating principal component analysis (PCA) to reduce redundant objectives during the search process. The proposed method preserves essential objective information while eliminating dependency noise that typically slows the evolutionary search. Experiments were conducted using the SF110 benchmark dataset through EvoSuite, and the approach was compared with standard many-objective search strategies. The findings demonstrate that PCA-based objective reduction can improve performance, achieving up to 3.87%, 5.50%, and 3.75% for coverage of line, branch, and mutation respectively. These results indicate that dimensionality reduction can significantly enhance scalability and efficiency in automated evolutionary test generation, providing a foundation for future adaptive and hybrid optimization strategies.

This is an open access article under the CC BY-SA license.

Corresponding Author:

Made Raja Adi Surya Saputra
Department of Computer Science, Faculty of Computer Science
Bina Nusantara University
11480 Jakarta, Indonesia
Email: made.saputra@binus.ac.id

1. INTRODUCTION

Conventional approaches to software testing predominantly depend on human involvement [1], [2]. In this setting, testers are responsible for supplying various input scenarios to represent system functionality and subsequently validating the produced outputs, despite having no visibility into the internal implementation logic of the software. Consequently, such testing is highly subjective and innacurate, and stable. This kind of software testing has a long cycle and incurs high costs, accounting for a large proportion of the investment in software reseach and development [3], [4] search-based software testing (SBST) has emerged as an effective approach for automating test case generation by formulating testing tasks as optimization problems and solving them using meta-heuristic techniques such as genetic algorithms (GAs) [5]. Within this paradigm, many-objective algorithms such as the many-objective sorting algorithm (MOSA) and the dynamic version, DynaMOSA, have shown strong effectiveness in generating test suites that optimize multiple coverage criteria simultaneously. To further improve scalability, aDynaMOSA introduces adaptive mechanisms that dynamically prioritize objectives based on control dependencies and search feedback, enabling more efficient exploration of large and complex search spaces [6].

Despite these advances, many-objective test generation still faces scalability challenges when the number of objectives becomes excessively large, as commonly observed in real-world software systems. High-dimensional objective spaces reduce selection pressure and slow convergence, limiting the effectiveness of both static and adaptive many-objective algorithms. To address this issue, dimensionality reduction techniques have been introduced. In particular, PCA-DynaMOSA integrates principal component analysis (PCA) with DynaMOSA to reduce redundancy among correlated objectives, improving search efficiency while maintaining competitive coverage results.

However, existing PCA-based approaches are limited to non-adaptive many-objective settings. PCA-DynaMOSA applies dimensionality reduction independently of adaptive objective management and does not leverage the dynamic prioritization mechanisms that characterize aDynaMOSA. As a result, current solutions either focus on reducing the dimensionality of the objective space without adaptivity or apply adaptive optimization without explicitly addressing objective redundancy. The integration of dimensionality reduction with adaptive many-objective search therefore remains underexplored.

To fill this gap, this study proposes PCA-aDynaMOSA, a novel approach that combines PCA-based dimensionality reduction with the adaptive many-objective optimization framework of aDynaMOSA. By embedding PCA into the adaptive search process, the proposed approach aims to simultaneously mitigate objective explosion and exploit dynamic objective prioritization. This integration is expected to enhance scalability and search efficiency while preserving the effectiveness of test case generation in large-scale, many-objective testing scenarios.

2. RELATED WORKS

Heuristic search techniques have progressively ben adopted in various research domains, including automated test case generation. One notable approach is the whole test suite (WTS) strategy implemented within the EvoSuite framework. In this approach, Fraser *et al.* [7] reformulated the generation of test case task as an optimization with a many-objective-based problem. They applied a weighted-sum strategy in which the overall fitness score is computed by aggregating the fitness values associated with multiple coverage targets.

Zhang and Li [8] introduced a many-objective test generation approach based on the longicorn beetle search algorithm. Their method incorporated a total path similarity metric that evaluates multiple execution paths in a relation to a target path set within the program. Similarly, Sahin *et al.* [9] developed a full test suite generation technique utilizing an archive-based multi-criteria into a unified fitness evaluation scheme, enabling simultaneous optimization of several objectives. The archive mechanism was further employed to enhance search efficiency. In addressing many-objective optimization, they also adopted a weighted-sum mechanism similar to that used in the whole suite strategy to optimize coverage targets concurrently.

Despite the advantages of weighted-sum approaches, including the whole suite method in handling multiple objectives more efficiently than single-objective search, they encounter difficulties when navigating non-convex spaces, often leading to suboptimal convergence in test case generation task [10]. To overcome this limitations, Panichella *et al.* [11] Introduced MOSA, a preference-based GA derived from the whole suite concept. Unlike traditional evolutionary algorithms that emphasize popularity diversity, MOSA prioritizes individuals whose fitness values approach zero, as the ultimate objective in test generation is to satisfy coverage targets completely.

A significant challenge in many-objective search-based testing lies in the explosion of coverage objectives, which can degrade search efficeincy and hinder the generation of high-coverage test cases. To address this issue, Panichella *et al.* [12] Proposed DynaMOSA, an extension of MOSA that enhances performance through dynamic target selection. In practice, coverage targets within a class are often interdependent. For instance, in a single method, certain branches cannot be executed unless precending branches have already been tranversed, folowing the top-down execution order of the code. Furthermore, some objectives are hierarchially dependent on dominant branches. DynaMOSA manages these dependencies by exploring coverage targets within a control dependency graph, where only dominant or currently reachable objectives are considered. Targets are removed from the graph once they have been successfully covered.

By dynamically limiting the set of achieve objectives, DynaMOSA reduced the dimensionality of the optimization problem, thereby improving search effectiveness. Building upon this framework, Panichella, *et al.* [13] later introduced a multi-criteria extension that allows simultaneous optimization of multiple coverage metrics, such as branch coverage and line coverage within a unified search process. More recently, research has extended search-based test case generation to incorporate non-functional properties.

For example, Grano, *et al.* [6] Proposed aDynaMOSA which augments the objective space by including execution time and memory consumption as additional optimization criteria.

3. METHOD

In this study, the aDynaMOSA is adopted as the base algorithm for generating an automatic test case. While aDynaMOSA effectively mitigates the scalability issue of many-objective optimization by dynamically managing target objectives, its performance may still degrade when the number of objectives grows excessively, as commonly observed in complex software systems with dense control flow and numerous coverage targets. To address this limitation, this research introduces a dimensionality-reduction-based objective aggregation technique using PCA that is inspired from previous study by Li *et al.* [14]. PCA has been successfully applied in prior work to reduce redundancy and correlation among coverage objectives while preserving essential optimization information. This research proposes some problem formulations:

3.1. Single-objective optimization

The optimization process begins by generating candidate test suites composed with flexible number of individual test cases. Each test case consists of a sequence of method invocations that may vary in structural composition. The fitness evaluation assesses the extent to which a candidate suites satisfies the predefined coverage objectives. Therefore, identifying an appropriate test suite T requires fulfilling all specified testing targets, which are defined as follows:

$$\min f_u(T) = \sum_{u \in U} d(u,T) \quad (1)$$

$d(u,T)$: The minimum value of the distance measure associated with target u under the defined distance function.

The function is designed to reduce the distance metric d such that $d(u,T) = 0$ indicates complete coverage of target u by the execution of test suite T. Variations in coverage criteria lead to different definitions of the distance function, which measures the degree to which the execution trace approaches fulfillment of the objective u.

When considering branch coverage, each conditional branch in the class under test (CUT) represents a distinct coverage objective. Thus, the problem can be moduled as identifying a test suite capable of coverigng all such branches. The coverage status of each branch is quantified using the distance function d, defined as follows:

$$\min f_b\ (T) = |M| - |M_T\ | + \sum_{b \in B} d(b,T) \quad (2)$$

$|M|$: Total number of methods. $|M_T|$: Number of executed methoused to search for the unexecuted method by T. $d(b,T)$: Normalized branch distance for branch $b \in B$.

Terminology of $|M| - |M_T\ |$ describes the number of method minus by number executed methods that used to find the unexecuted methods by T. The minimum normalized branch distance for each branch $b \in B$ is defined as follows:

$$d(b,t) = \begin{cases} 0 \\ 1 \end{cases} \frac{D_{min}(t \in T,\ b)}{D_{min}(t \in T,\ b)+1} \quad (3)$$

$(t \in T, b)$: Represents the unnormalized branch distance over all test cases $t \in T$. The function represents the raw branch distance computed according to the selected distance evaluation scheme, quantifying how close the predicate controlling branch b is to being satisfied during execution.

In strong mutation coverage, each testing target corrsponds to a modified version of the original class, obtained by introducing an artificial change that simulates a potential fault. The objective is to construct a test suite capable of detecting all such mutants, i.e., killing every injected mutation. A mutant is considered killed by a test case if executing the original and mutated versions under the same input produces different observable behaviors, either in terms of returned values or externally visible object states. In practice, this distinction is verified through automated assertions that compare the observable oputputs of the original implementation. If the assertion holds for the original program, but fails for the mutant, the mutation is regarded as detected.

The effectiveness of strong mutation testing relies on the concepts of infection and propagation. Let S_o and S_m denote the execution states of the original and mutated programs respectively. A mutant is said to cause infection if $S_o \neq S_m$ immediately after the mutated statement is executed, indicating that the injected

fault has altered the internal program state. Propagation occurs when this infected satte influences subsequent computations. Ultimately, such of an event is leading to a discrepancy in observable outputs. Only when both infection and propagation occur can a mutant be successfully detected by the test case.

Problem 2.1. Given a set of mutants $M = \{M_1, \ldots, M_k\}$ contained in a class, determine a test suite $T = \{t_1, \ldots, t_n\}$ that can eliminate the mutants according to the following formulation:

$$\min f_m(T) = f_{B_m}(T) + \sum_{m \in M}(d_i(m,T) + d_p(m,T)) \quad (4)$$

$f_{B_m}(T)$: Overall fitness function for every branch in T that directly controls the dependency of mutant m. d_i: Estimation of distance to the infection state. d_p: Propagation distance.

3.2. Many-objective optimization

In this research, coverage criteria for test case generation are modeled as a many-objective optimization problem, where each objective represents the distance to a particular testing target in the class under examination [7]. The optimization process therefore seeks to minimize these distances collectively. Accordingly, the following formulation is adopted as follows:

Problem 2.2. Assume that $U = \{u_1, \ldots, u_k\}$ denotes the set of coverage objectives. The goal is to construct a set of Pareto-optimal test cases $T = \{t_1, \ldots, t_n\}$ such that the fitness values associated with all targets $u_1, \ldots, u_k$ *i.e.*, are minimized. Thus, the problem can be expressed as a many-objectiive problem with k objectives:

$$\begin{cases} \min f_1(t) = d(u_1, t) \\ \quad . \\ \quad . \\ \quad . \\ \min f_k(t) = d(u_k, t) \end{cases} \quad (5)$$

$d(u_1, t)$: Distance of test case t from covering the test target u_i. $\{f_1, \ldots, f_k\}$: Corresponding fitness vector. Using the generalized reformulation approach, multiple coverage criteria including branch coverage, statement coverage, and strong mutation coverage can be defined according to the following formulation:

Problem 2.3. Let $B = \{b_1, \ldots, b_k\}$ which is set of branches of a class. Find the number of sets of non-dominant test cases $T = \{t_1, \ldots, t_n\}$ that minimize the fitness function for all test case $u_1, \ldots, u_k$ including minimizing objective k as follows:

$$\begin{cases} \min f_1(t) = al(b_1, t) + d(b_1, t) \\ \quad . \\ \quad . \\ \quad . \\ minf_k(t) = al(b_k, t) + d(b_k, t) \end{cases} \quad (6)$$

$d(b_1, t)$: Normalized branch distance for each executed branch closest to t in relation to branch in relation to branch b_j(i.e., those with the minimum control-dependency depth). $al(b_1, t)$: Respective degree of approximation (i.e., the number of control dependencies between the closest executed branch and b_j).

Problem 2.4. There is $S = \{S_1, \ldots, S_k\}$ as a set of statements in the class. Find the number of sets of non-dominant test cases $T = \{t_1, \ldots, t_n\}$ that minimize the fitness function for all test cases $u_1, \ldots, u_k$ including minimizing objective k as follows:

$$\begin{cases} \min f_1(t) = al(s_1, t) + d(b(s_1), t) \\ \quad . \\ \quad . \\ \quad . \\ \min f_k(t) = al(s_k, t) + d(b(s_k), t) \end{cases} \quad (7)$$

$d(b(s_j), t)$: Normalized branch distance between test case t and the branch closest to $b(s_j)$. i.e., the branch controlling the execution of statement s_j. $al(s_j, t)$: Represents the corresponding approximation level.

Problem 2.5. There is $M = \{m_1, \ldots, m_k\}$ as a set of mutant of the class. Find a set of non-dominated test case from $T = \{t_1, \ldots, t_n\}$ which minimize fitness function for all every mutant $m_1, \ldots, m_k$ as follows:

$$\begin{cases} \min f_1(t) = al(m_1,t) + d(b(m_1),t) \\ \qquad + d_i(m_1,t) + d_p(m_1,t) \\ \qquad \vdots \\ \min f_k(t) = al(m_k,t) + d(b(m_k),t) \\ \qquad + d_i(m_k,t) + d_p(m_k,t) \end{cases} \quad (8)$$

$d(b(m_1),t)$: Normalized branch distance and approxination level of test case *t* for mutant m_k. $d_i(m_k,t)$: Infection distance. $d_p(m_k,t)$: Propagation distance.

3.3. PCA algorithm

PCA is a multivariate statistical method designed to analyze tabular data in which observation is characterized by multiple correlated quantitative variables [15]. The primary objective of PCA is to extract the most informative structure from the dataset by transforming the original correlated variables into a new set of mutually orthogonak variables, known as principal components. These components capture dominant variance patterns and reveal similarity structures among observations.

In this study, PCA is applied to examine and reduce the number of coverage objectives involved in the optimization process. By analyzing the relationships among the objective fitness values, redundant or less informative objectives in the high-dimensional objective space can be identified and removed. This dimensionality reduction process aims to simplify the many-objective test case generation problem while preserving the most relevant information [14]. The main advantages of applying PCA in this context includes:

- Eliminating redundant feature characteristics within the objective space;
- Reducing computational complexity and enhancing the efficiency of the seach process; and
- Preserving the most critical coverage objectives.

From a mathematical standpoint, PCA performs dimensionality reduction by projecting the original objective space into a new set of orthogonal coordinates. Each dimension represents an objective, and the transformation is guided by variance maximization. Components associated with larger variance values are considered to contain more significant information. Therefore, the principal components are selected according to the descending order of their variance.

The first principal component is defined as the direction that maximizes data variance. Subsequent components are selected such that they are orthogonal to previously chosen components and maximize the remaining variance. This iterative process continues until a reduced coordinate space is constructed. Dimensions associated with zero variance are discarded, as they do not contribute meaningful information to the transformed space.

3.4. Many-objective search

Over the past decade, considerable attention has been devoted to the problem of automated test data generation [5], [16], particularly with the objective of achieving high levels of code coverage under different coverage metrics such as branch [17], statement [16], and method coverage [18]. Among the proposed techniques, search-based approaches especially GAs [19] have demonstrated strong potential in automating this process [5]. Existing methods generally fall into two main categories: single-target and multi-target formulations. In single-target approaches, evolutionary algorithms attempt to optimize one coverage goal at a time. Each target, such as a specific branch, is translated into a fitness function that evaluates how close a generated test case or test suite is to satisfying that target [16]. This proximity is typically measured using two white-box heuristics, namely the approach level and the normalized branch distance.

To overcome the limitations of optimizing individual target sequentially, Fraser and Arcuri [20] introduced a multi-target strategy that simultaneously optimizes multiple coverage objectives. Their approach, known as WTS generation, evolves entire test suites instead of individual test cases. The search process is guided by a suite-level fitness function that aggregates coverage heuristics across all branches of the CUT. Subsequent improvements led to the archive-based whole suite approach (WSA), which narrows the search to uncovered branches and maintains an archive to store promising test cases that reach those targets [21]. Building upon the concept of simultaneous optimization, Panichella *et al.* [11] later proposed MOSA, a many-objective GA for test case generation. Unlike WTS and WSA, MOSA evolves individual test cases while evaluating them against multiple branch objectives using branch distance and approach level metrics.

The goal of test case generation is to construct a test suite consisting of unit tests that exercise the program under examination. The outputs prodiced by executing these tests are normally evaluated by a test

oracle to determine correctness. However, in automated white-box testing scenarios, such oracles are typically unavailable, making it difficult to automatically verify whether the observed output matches the intended behavior without additional manual intervention by developers [22].

Each test case consists of an ordered sequence of test statements designed to interact with the program. Two important considerations arise in the context. First, test cases should remain concise, as excessively long or complex sequences increase the effort required for manual maintenance and may prolong the testing cycle. Therefore, generating compact test cases contributes to improved practicality and maintainability of the resulting test suite. Second, the generated tests should aim to maximize coverage of the defined testing objectives. Higher coverage indicates that the test suite exercises a broader portion of the program's behavior, thereby improving the effectiveness and robustness of the testing process. Several fundamental concepts related to automated test case generation are discussed in the following section.

3.5. Control flow graph (CFG)

The CFG models the running process behavior of a program by representing the potential paths that may be traversed during runtime. Based on this representation, different coverage objectives can be formally derived and evaluated.

3.6. Test suites and test cases

A test suite represents a collection of test cases designed to assess the behavior of a program. Each test case is composed of a sequence of method calls with variable lengths. In single-objective search techniques, a separate test case is produced for each testing objective. The resulting test cases are subsequently combined to construct a comprehensive test suite for testing the whole program.

3.7. Objectives to be covered

The objectives involved in program evaluation can be assessed using various coverage metrics, including different coverage measures, namely line, branch, mutation, and method coverage metrics. In this study, the evaluation is limited to three specific criteria: line, branch, and mutation coverage. The performance indicators utilized in the aDynaMOSA approach are summarized in Table 1.

Table 1. Performance indicator

ID	Performance indicator	Description
I1	Executed loop count	Indicates the total number of the loop iterations that were performed during the execution of the test cases.
I2	Method coverage count	Represents the number of distinct methods that were involved by the generated test cases.
I3	Total method calls in tests	Reflects the pverall number of method invocations occuring accross all executed test cases.
I4	Object instantiation count	Shows how may objects were created during the execution of the test suite.
I5	Covered statements	Represents how many statements in the code were actually covered.
I6	Test case statements	Total number of executable statements within all generated test cases.
I7	Test length	Average length of test cases in terms of method call sequences.

The proposed performance indicators enable the evaluation of how effectively the PCA-aDynaMOSA algorithm performs compared with the baseline aDynaMOSA method. For example, higher values of covered method (I2) and covered statements (I5) generally reflect stronger coverage capabilities, whereas lower numbers of test case statements (I6) and shorter test lengths (I7) indicate more concise and efficient tests. By applying these indicators during experimental analysis, the study is able to examine the balance between testing thoroughness and efficiency, thereby providing insights into the scalability and practical applicability of the proposed approach.

3.8. Branch coverage

Branch coverage refers to decision points in this profram, such as if, while, or similar conditional constructs. Whitin the CFG, these decision points are represented as nodes that typically contain at least tow outgoing edged. Each outgoing edge corresponds to a possible branch of execution, meaning that single node may lead to multiple alternative paths. The degree to which a test case *t* covers a branch *b* can be quantified using the following fitnesss formulation:

$$f_B(t) = al(b_i, t) + d(b_i, t) \quad (9)$$

$al(b_i, t)$: The structural distance between the branch executed by test case *t* and the target branch that must be covered. The distance $d(b_i, t)$ corresponds to the total count of control dependencies or edged separating decision nodes in the CFG. Furthermore, the branch distance can be determined using the following formulation: It is equal to the number of control dependencies or edges between branch nodes in the CFG. Moreover, $d(b_i, t)$ is the branch distance, which can be calculated by the following formula:

$$d(b,t) = \begin{cases} 0 \\ 1 \end{cases} \frac{D_{min}(t \in T,\ b)}{D_{min}(t \in T,\ b)+1} \tag{10}$$

$D_{min}(t \in T,\ b)$: Represents the minimized non-normalized distance. Since a decision statement normally has two possible outcomes, the predicate may be executed multiple times during testing, and the smallest computed distance value is selected during execution.

3.9. Line coverage

In a CFG representation, each node corresponds to a specific line or block of program code. The line coverage achieved by a test case *t* can therefore be formulated using the following fitness function:

$$f_L(t) = al(s_i, t) + d(b(s_i), t) \tag{11}$$

$al(s_i, t)$: Indicates the structural distance between program statements, and $(b(s_i), t)$: Represents the distance from the branch executed by test case *t* to the nearest branch relative to the target mutation.

3.10. Mutation coverage

Mutation testing is performed by introducing small artificial modifications into the source code through set of mutation operators. These changes simulate potential faults that may occur in real programs. The mutated versions of the program are generated by applying mutation operators to selected locations in the CFG of the original source code.

In this study, several mutation operators are employed, including bariable replacement, unary operator insertion, constant substitution, and arithmetic operator replacement. By executing the same test cases on both original and mutated programs, differences in the resulting outputs can be observed. If the behavior of the mutated program differs from the original program, the mutant is considered detected (or killed). The mutation coverage of test case *t* is then evaluated using the following fitness formulation:

$$F_M(t) = al(m_i, t) + d(b(m_i), t) + d_i(m_i, t) + d_p(m_i, t) \tag{12}$$

$al(m_i, t)$: Denotes the structural distance between the program statement executed by test case *t* and the location of the mutated statement. $d(b(m_i), t)$: Indicates the normalized branch distance associated with the mutated branch. $d_i(m_i, t)$: Represents the infection distance. $d_p(m_i, t)$: Refers to the propagation distance.

3.11. Test case generation under multiple optimization objectives

In automated test case generation, two primary considerations are typically addressed: minimizing the number of statements within each test case while simultaneously maximizing the coverage achieved by the generated tests. Different coverage criteria can be used to evaluate the adequacy of a test suite. Each criterion can be treated as a separate optimization objective, allowing the test generation problem to be modeled as a many-objective optimization task.

Assume that the tested program contains *m* coverage tests, which consists of S_B objectives related to branch coverage, S_L objectives associated with line coverage, and S_M objectives representing mutation coverage. Hence the total number of objectives is given by $m = S_B + S_L + S_M$. For a test suite $T = \{t_1, t_2, \dots, t_n\}$, the goal of the optimization process is to minimize the fitness value associated with all objectives:

$$\begin{cases} \min f_B(t)(i = 1, 2, \dots, s_B) \\ \min f_L(t)(j = 1, 2, \dots, S_L) \\ \min f_M(t)(k = 1, 2, \dots, S_M) \end{cases} \tag{13}$$

3.12. PCA algorithm

To generate effective test cases, PCA is applied to reduce the dimensionality of the objective space by removing redundant information while preserving the most significant components. The resulting reduced

objective set is then utilized during the many-objective sorting process of candidate solutions within the population, guiding the evolutionary search. By applying this dimensionality reduction stragethy, the degradation in search performance often observed in aDynaMOSA when dealing with a large number of optimization objectives can be mitigated.

Figure 1 presents the sequential workflow of the PCA process used in this study to reduce the dimensionality of the objective space prior to guiding the many-objective evolutionary search, The procedure begins by acquiring the initial population along with the complete set of raw objectives generated from the test case optimization process. Once obtained, the algorithm extracts the fitness matrix *f*, which represents the evaluation scores of each iindividual in relation to all defined objectives.

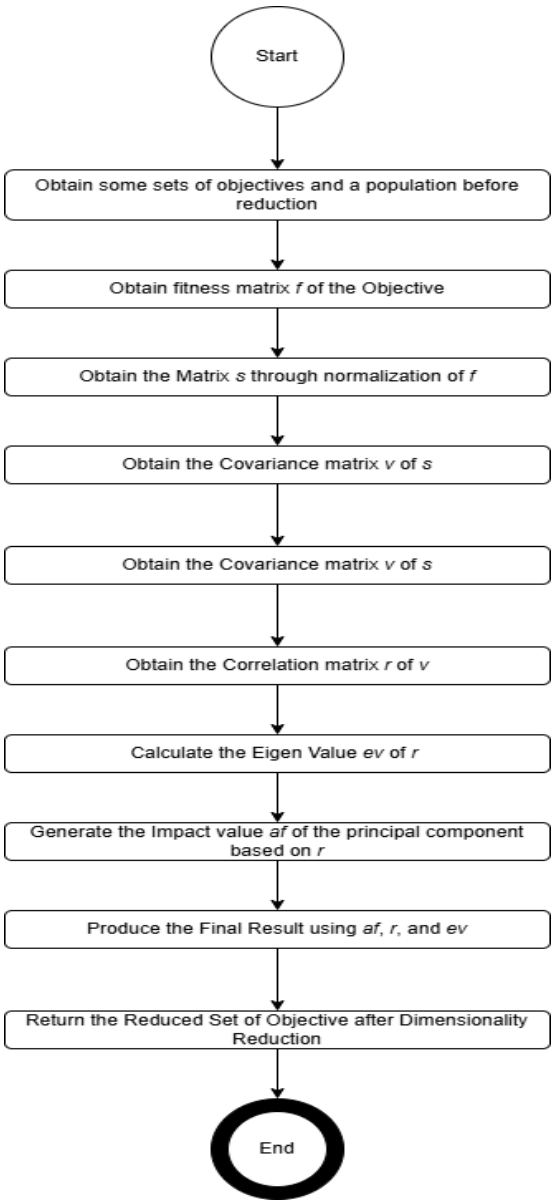

Figure 1. Overview of the PCA procedure

The fitness matrix is then normalized to produce matrix *s*, ensuring that all objectives share a comparable numerical scale and preventing any objective with a disproportionately large value range from dominating the transformation process. Based on normalized values, the covariance matrix *v* is computed to capture the relationships and variability between the objectives. From this covariance matrix, the correlation matrix *r* is derived to analyze the strength of linear relationships across objectives.

Using the correlation matrix, the eigenvalues and eigenvectors are calculated, forming the foundation of the PCA transformation. The eigenvalues indicate the relative contribution of each principal component, whereas the eigenvectors define the new orthogonal axes representing uncorrelated objective dimensions. Subsequently, the algorithm generates a component impact score to determine which principal components retain the highest variance and thus carry the most meaningful information. Only principal components with significant contribution are retained and used to construct the reduced objective set after the dimensionality-reduction step.

Finally, the algorithm returns the reduced set of objectives, which replaces the original full objective set during the evolutionary search phase. By eliminating redundant and highly correlated objectives while preserving the most informative variance, PCA contributes to reducing algorithmic complexitym improving search efficiency, and enhancing scalability when dealing with large many-objective optimization spaces.

Since automated test case generation often involves a large number of objectives, applying dimensionality reduction indiscriminately may introduce issues such as missed coverage targets, increased computational cost, or degraded outcomes from the reduction process. To address this limitation, the proposed method groups objctives into several subsets. The dimensionality of each subset is then adaptively reduced accorfing to the number of objectives within that group. The funal objective set used to guide the optimization process is obtained by aggregating the reduced objectives from all groups. The specific grouping strategy employed in the algorithm was determined through experimental evaluation. The performance of the algorithm was evaluated by series of experiments. The experimental outcomes were subsequently analyzed and compared with the result produced by the aDynaMOSA approach.

3.13. Experimental design

The experimental study was mainly carried out to investigate the following research questions (RQs) concerning multi-critera adaptive many-objective test case generation. RQ1. To what extent does the size of objective groups influence the search performance of the PCA-aDynaMOSA algorithm? Moreover, how can the optimal number of groups for the algorithm be determined?

Figure 2 illustrates the procedural workflow of the proposed PCA-based approach for many-objective test case generation. The process begins by initializing a set of objectives U^* alongside a candidate population P. Once initialized, the algorithm proceeds by grouping the objectives into multiple clusters based on similarity or structural relationships.

Each objective group $U^*[i]$ is then independently processed through a dimensionality-reduction phase using PCA. This step transforms each group of objectives into a reduced objective set $U_{PCA}[i]$, retaining only the most informative features while eliminating redundant correlations. The reduced objective groups are subsequently aggregated to construct a global reduced representation denoted as U_{PCA}.

Following this aggregation, the reduced objective space undergoes preference ordering to prioritize candidate solutions according to their performance across the transformed objective dimensions. The ordered population is then subjected to non-dominated sorting to identify Pareto-optimal individuals and assign dominance ranks. Finally, the sorted population $P^{sorted}$is returned as the output of the process, concluding the workflow. The workflow of the proposed PCA-based many-objective test case generation approach is described in Algorithm 1.

Algorithm 1 PCA-based many-objective test case generation

```
Input: Population P, Population Number S, Target set M
Output: Generated test suite T

1 Initialize population S randomly
2 Initialize archive A
3 Identify uncovered targets U

4 While search budget no exhausted, do
5      Group objectives in U
6      Apply PCA to each group
7      Generate reduced objective set
8      Perform performance-based sorting
9      Update archive with new covered targets
10 end while

11 return A as a final test suite T
```

RQ2. How does PCA-aDynaMOSA compare with aDynaMOSA in achieving line, branch, and mutation coverage for the generated test cases? RQ3. How does PCA-aDynaMOSA compare with aDynaMOSA regarding overall multi-criteria coverage and search efficiency? Moreover, what relationship exists between search efficiency and the multi-criteria obtained from the generated test cases?

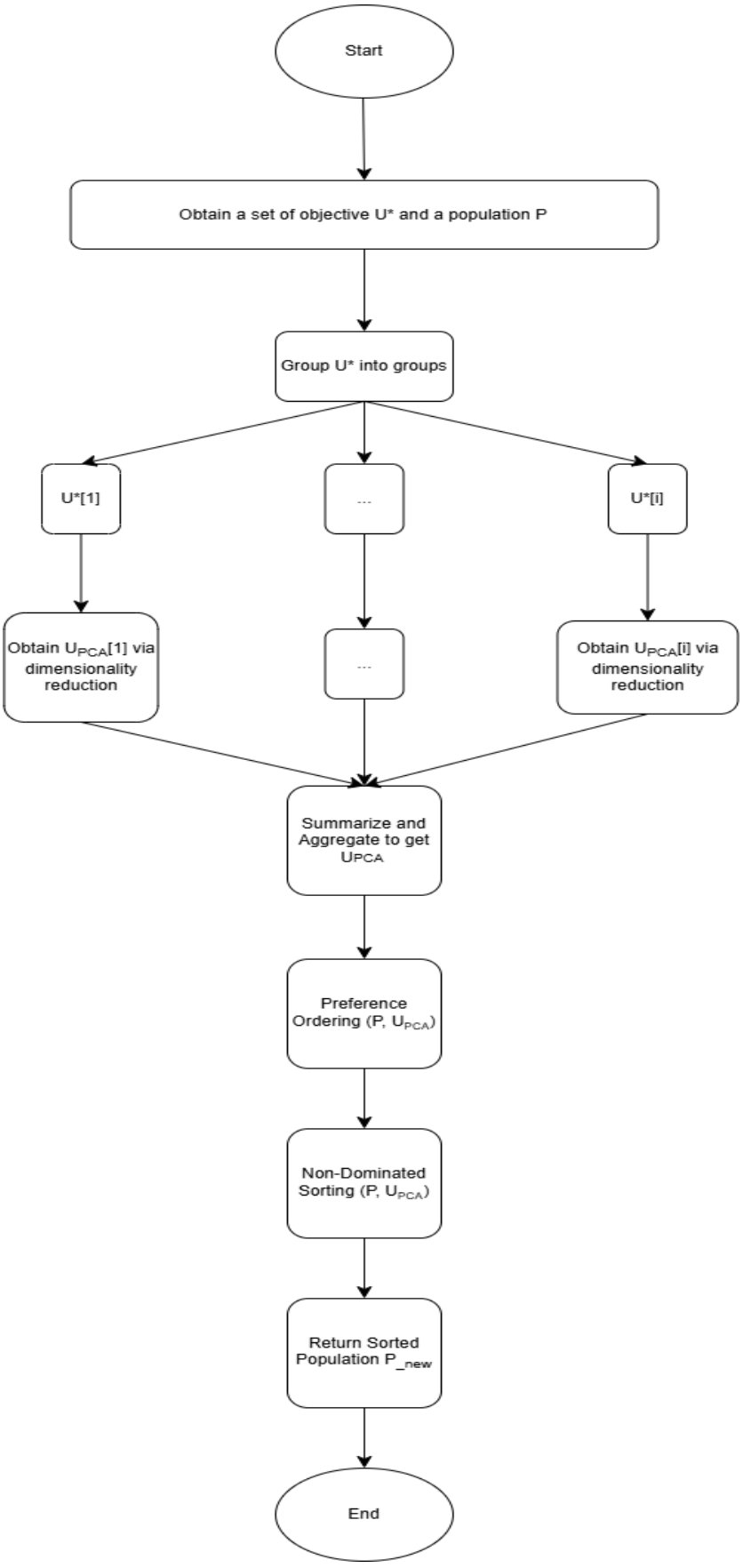

Figure 2. Flowchart of PCA for many-objective test case generation

3.14. Evaluation metrics

The following evaluation metrics were selected to test and compare the test case search capabilities of both the original and the improved algorithm.

3.14.1. Testing coverage

This metric reflects the sufficiency of the final test suite in assessing the program. It is composed of several sub-metrics, namely line coverage, branch coverage, mutation coverage, and multi-criteria coverage. Multi-criteria coverage represents the overall coverage of all testing objectives, obtained by aggregating the coverage of all branches, lines, and mutants involved in the test case generation process. The formulas used to compute these coverage metrics are defined as follows:

$$\begin{cases} statement_{coverage} = \frac{Number\ of\ covered\ statements}{Total\ number\ of\ statements\ to\ be\ covered} \times 100\% \\ branch_{coverage} = \frac{Number\ of\ covered\ branches}{Total\ number\ of\ branches\ to\ be\ covered} \times 100\% \\ mutation_{coverage} = \frac{Number\ of\ killed\ mutants}{Total\ number\ of\ mutants\ to\ be\ killed} \times 100\% \\ multicriteria_{coverage} = \frac{Number\ of\ covered\ objectives}{Total\ number\ of\ objectives\ to\ be\ covered} \times 100\% \end{cases} \quad (14)$$

3.14.2. Search efficiency

By utilizing the coverage percentage recorded at one-second intervals during the test case generation process, the search efficiency of the algorithm can be computed using the following equation:

$$AUC = \sum_{i=0}^{120} \frac{[cov_i + cov_{i+1}] * \Delta Time}{2 * TotalTime} \quad (15)$$

cov_i: Coverage percentage measured at time i. cov_{i+1}: Coverage percentage measured at time $i + 1$. $\Delta Time$: The time interval between two consecutive coverage measurements. $TotalTime$: The total execution time of the algorithm.

The improvement in testing coverage between the original algorithm and the enhanced algorithm can be calculated using the following equation:

$$IncreaseRate = \frac{Value_{after} - Value_{before}}{Value_{before}} \times 100\% \quad (16)$$

3.14.3. Experiment environment

The proposed algorithm was integrated into the EvoSuite framework and experimentally compared with aDynaMOSA. The experiments were carried out on a Windows-based platform. The details of the experimental environment are summarized in Table 2.

Table 2. Experiment environment setup

Hardware	12th-Gen Intel(R) Core (TM) i7-12700H @ 2.3GHz RAM 16GB
Software	Windows 11 Java 1.8

3.14.4. Datasets

First, the CKJM tool was used to analyze the complexity of all projects in the SF110 dataset [23]. Subsequently, projects with relatively high complexity were selected to evaluate the performance of the proposed algorithm. The SF110 data consists of 110 statistically representative open-source Java projects collected from SourceForge. SourceForge is a well-known open-source repository that hosts more than 300,000 projects and over two million registered users, covering a wide variety of software domains. Therefore, the SF110 dataset is considered suitable for evaluating the performance of test case generation algorithms and frameworks.

However, due to the large number of projects and classes contained in the dataset, conducting experiments on all classes would require a considerable amount of time and computational resources. In addition, more than 50% of the classes in the SF110 dataset exhibit low program complexity, meaning that relatively simple methods can already achieve high coverage for most testing objectives.

To address this issue, CKJM [24], [25] a tool that calculates various object-oriented metrics by analyzing the bytecode of compiled Java programs was used to assess the complexity of all classes in the dataset. The metrics include weighted methods per class (WMC), depth of inheritance tree (DIT), coupling between object classes (CBO), lack of cohesion in methods (LCOM), response for a class (RFC), and number of children (NOC).

The experimentail dataset was constructed using the following procedure. Initially, the complexity of all classes in the SF110 dataset was analyzed using the CKJM tool, and the number of testing objectives for each class was identified using the EvoSuite framework. The classes under test were then ordered based on their complexity and the number of objectives. Projects containing more complex classes with a larger number of testing objectives were subsequently selected and incorporated into the experimental dataset.

Considering that a small number of projects or classes exhibited abnormal behavior during the experiments, those projects or classes were excluded from the study. The selected projects and the number of classes chosen from each project are presented in Table 3. As shown in Table 3, 49 out of the 110 projects in the SF110 dataset were selected. In total, 110 classes were used as the primary test data for the experiments. To compare the performance of PCA-aDynaMOSA with aDynaMOSA, each experiment was repeated 10 times for every test class in the selected dataset. The results obtained from these runs were then averaged to support comparison and analysis.

Table 3. List of software projects used as experimental dataset

Project	Number of classes	Project	Number of classes
gaj	1	biff	1
sfmis	1	lavalamp	2
imsmart	2	jhandballmoves	3
jdbacl	1	hft-bomberman	2
omjstate	1	dom4j	5
beanbin	1	openjms	2
inspirento	3	gae-app-manager	1
jsecurity	5	biblestudy	1
nekomud	1	lhamacaw	5
geo-google	3	ext4j	1
jni-inchi	1	fim1	2
gangup	4	fixsuite	1
apbsmem	7	twfbplayer	2
bpmail	1	wheelwebtool	7
xisemele	1	javathena	1
corina	5	xbus	2
schemaspy	3	ifx-framework	1
petsoar	1	classviewer	2
diffi	2	quickserver	1
glengineer	6	feudalismgame	3
follow	2	liferay	1
lilith	3	pdfsam	2
lotus	1	firebird	3
resources4j	1	heal	3
diebierse	1	-	

3.14.5. Parameter settings

The experimental configuration follows the default parameter settings provided by EvoSuite for the aDynaMOSA algorithm, which has previously demonstrated stable performance. To maintain experimental consistency and ensure fair comparison, the proposed method employs the same parameter configuration summarized in Table 4. Note: The search duration corresponds to approximately two minutes of execution time. The coverage include line, brahc, and mutation coverage metrics.

Table 4. Configuration of parameters used in the many-objective optimization algorithm

Algorithm settings	Assigned value
Population size	100 individuals
Maximum search duration	120000 (≈2 minutes)
Covering objectives	Line, branch, and mutation
Crossover probability	75%

4. RESULTS AND DISCUSSION

To identify the optimal grouping strategy for PCA-aDynaMOSA, a set of group settings was evaluated, as listed in Table 5. The experiments were conducted with seven different numbers of groups, ranging from 5 to 60, to investigate the effect of grouping granularity on the algorithm's performance. This configuration enables a comprehensive comparison to identify the group size that provides the best balance between search efficiency and optimization effectiveness.

Table 5. Experimental group configurations for PCA-aDynaMOSA

Group ID	Number of groups
PCA_5	5
PCA_10	10
PCA_20	20
PCA_30	30
PCA_40	40
PCA_50	50
PCA_60	60

4.1. Performance of PCA-aDynaMOSA under different group settings

To identify the PCA-aDynaMOSA group configuration that yields the best performance in terms of multi-criteria coverage. 18 classes were randomly selected from the test dataset. The average coverage of these classes was then calculated and analyzed to evaluate the effectiveness of the different group settings.

To identify the most effective grouping for PCA-aDynaMOSA, several group size settings were evaluated based on their ability to improve multi-coverage performance. Table 6 illustrates that total objectives assigned to PCA groups significantly affected the effectiveness of the test case generation process. Both excessively large and excessively small group sizes can negatively impact the effectiveness of the algorithm.

For instance, the PCA configurations for group 5 and 60 do not demonstrate notable improvements compared to the baseline aDynaMOSA algorithm. When total objectives within each group are too small, the resulting objective set used to guide the search process remains nearly unchanged, while additional computational effort is still required to perform dimensionality reduction. Conversely, when total objectives in a group become too large, the dimensionality reduction process becomes more difficult to converge, which can considerably increase the computational time of the algorithm. Nevertheless, with appropriate parameter configurations, the application of PCA can enhance the performance of aDynaMOSA by improving search efficiency and increasing overall test coverage.

Table 6. Statistics of different numbers of PCA groups

Group ID	Number of groups	Average of multi-criteria coverage	Average of AUC
PCA_5	5	82.37%	79.81%
PCA_10	10	83.87%	80.47%
PCA_20	20	83.61%	80.86%
PCA_30	30	82.89%	80.24%
PCA_40	40	82.34%	77.96%
PCA_50	50	83.71%	80.15%
PCA_60	60	81.38%	78.71%
aDynaMOSA	-	82.07%	79.44%

4.2. Project-level coverage analysis (RQs 2 And 3)

Tables 7–9 omits projects for which the compared algorithms achieved identical coverage values. Performance differences of less than or equal to 0.5% are marked with the symbol "-". Additionally, values highlighted in blue denote statistically significant improvements, while values shown in red indicate statistically significant reductions.

4.2.1. Line coverage for each project

Table 7 presents the average line (statement) coverage achieved by PCA-aDynaMOSA and the baseline aDynaMOSA across the selected benchmark projects. The comparison highlights project-level differences in coverage performance, allowing the effectiveness of the proposed dimensionality reduction strategy to be evaluated under various software characteristics.

4.2.2. Branch coverage for each project

Table 8 summarizes the average branch coverage obtained by both PCA-aDynaMOSA and aDynaMOSA for each benchmark project. The project-level comparison provides insights into the capability of the proposed approach to improve decision-path exploration and branch coverage consistency across software system with different levels of complexity.

Table 7. Average statement coverage across benchmark projects

Benchmark Project	PCA-aDynaMOSA Statement Coverage (%)	aDynaMOSA Statement Coverage (%)	PCA-aDynaMOSA vs aDynaMOSA (%)
imsmart	90.67	84.14	7.75
inspirento	92.61	91.18	1.57
jsecurity	92.58	92.12	0.50
apbsmem	90.18	89.60	0.65
corina	77.82	78.04	-
diffi	92.98	92.07	0.99
follow	94.01	81.11	15.91
resources4j	77.94	75.76	2.18
diebierse	97.06	100.00	-2.94
dom4j	85.62	85.04	0.69
openjms	87.98	85.78	2.56
lhamacaw	64.77	63.02	2.78
twfbplayer	98.14	97.33	0.83
wheelwebtool	59.30	57.46	3.19
xbus	85.21	83.04	2.62
quickserver	96.53	95.83	0.72
heal	72.68	72.49	-
liferay	75.79	74.91	1.18
firebird	94.05%	91.52%	2.76%
Mean over all projects	86.42%	83.71%	3.87%

Table 8. Average branch coverage across benchmark projects

Benchmark projects	PCA-aDynaMOSA branch coverage (%)	aDynaMOSA branch coverage (%)	PCA-aDynaMOSA vs aDynaMOSA (%)
imsmart	86.11	76.11	13.14
inspirento	96.45	94.86	1.67
jsecurity	90.64	89.72	1.02
apbsmem	90.89	90.62	-
xisemele	80.00	77.14	3.70
corina	70.83	69.79	1.48
diffi	92.86	88.80	4.57
follow	91.98	66.98	37.32
resource4j	91.89	70.27	30.77
diebierse	80.00	90.00	-11.11
hft-bomberman	98.53	97.06	1.52
dom4j	82.59	82.20	-
openjms	77.42	76.04	1.81
lhamacaw	79.04	77.32	2.22
twfbplayer	91.41	88.65	3.11
wheelwebtool	69.25	67.75	2.22
xbus	83.89	81.32	3.15
classviewer	86.37	87.14	-0.89
quickserver	95.00	93.33	1.79
heal	79.51	79.32	-
liferay	70.26	68.84	2.06
firebird	85.88	81.80	4.98
Mean over all projects	85.04	81.59	5.50

4.2.3. Mutation coverage for each project

Table 9 reports the average mutation coverage achieved by PCA-aDynaMOSA and the baseline aDynaMOSA for each benchmark project. Mutation coverage provides a stronger assessment of test effectiveness because it evaluates the ability of generated test cases to detect injected faults. The project-level comparison therefore complements the previous line and branch coverage analyses by demonstrating the fault-detection capability of the proposed approach across different software projects.

Table 9. Average mutation coverage across benchmark projects

Name	PCA-aDynaMOSA branch coverage (%)	aDynaMOSA branch coverage (%)	PCA-aDynaMOSA vs aDynaMOSA (%)
imsmart	86.60	73.33	18.09
inspirento	93.42	91.16	2.48
jsecurity	89.34	88.93	-
apbsmem	95.12	95.04	-
xisemele	83.82	82.35	1.79
corina	77.71	76.78	1.21
schemaspy	69.74	68.62	1.64
diffi	89.65	86.48	3.67
glengineer	89.42	89.29	-
follow	87.17	71.14	22.54
lilith	76.77	75.98	1.05
diebierse	98.42	100.00	-1.58
jhandballmoves	99.10	98.75	-
dom4j	80.48	79.73	0.94
openjms	83.21	82.48	0.88
biblestudy	96.15	96.15	-
lhamacaw	63.08	60.20	4.78
ext4j	100.00	97.87	2.17
fim1	94.69	95.58	-0.93
twfbplayer	92.48	91.61	0.95
wheelwebtool	63.98	61.49	4.05
xbus	82.61	79.25	4.24
quickserver	88.30	87.47	0.96
heal	81.44	80.11	1.65
liferay	72.97	69.93	4.35
Mean over all projects	85.43	83.19	3.75

5. CONCLUSION

This research proposed an enhanced test case generation method by integrating PCA into the aDynaMOSA evolutionary algorithm to address the challenges of many-objective optimization in automated unit testing. The PCA technique was applied to reduce the dimensionality of the objective space, thereby eliminating redundant objectives while preserving critical coverage goals. The results of the experiments conducted of 110 selected classes from 49 projects in the SF110 dataset demonstrated that PCA-aDynaMOSA consistently outperformed the baseline aDynaMOSA from the perspective of line, branch, and mutation coverage. Specifically, PCA-aDynaMOSA achieved mean improvements of 3.87%, 5.50%, and 3.75% for line, branch, and mutation coverage, respectively. Furthermore, it exhibited improved search efficiency and more concise test cases without sacrificing thoroughness. These findings confirm that dimensionality reduction via PCA can enhance the scalability and effectiveness of evolutionary algorithms in software testing and show that dimensionality reduction can significantly enhance many-objective search in automated test case generation. Future work may explore alternative dimensionality reduction techniques, such as t-SNE and UMAP, to evaluate their comparative impact on automated evolutionary test generation. Additionally, integrating the proposed PCA-aDynaMOSA approach into continuous integration/continuous deployment (CI/CD) environments would enable assessment of its practical usefulness in modern development workflows. To improve external validity, future studies should also expand experimentation to software projects beyond the SF110 benchmark to examine cross-domain generalization.

Further investigation into runtime performance, particularly execution time and memory consumption in real-world settings, is necessary to evaluate scalability and industrial applicability. Finally, the development of an interactive visualization or user interface for test suite outputs and PCA results is recommended to enhance interpretability and usability for software engineering practitioners.

FUNDING INFORMATION

Authors state there is no funding involved.

AUTHOR CONTRIBUTIONS STATEMENT

This journal uses the Contributor Roles Taxonomy (CRediT) to recognize individual author contributions, reduce authorship disputes, and facilitate collaboration.

Name of Author	C	M	So	Va	Fo	I	R	D	O	E	Vi	Su	P	Fu
Made Raja Adi Surya Saputra	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓	
Maria Seraphina Astriani				✓		✓				✓		✓		

C : **C**onceptualization
M : **M**ethodology
So : **So**ftware
Va : **Va**lidation
Fo : **Fo**rmal analysis
I : **I**nvestigation
R : **R**esources
D : **D**ata Curation
O : Writing - **O**riginal Draft
E : Writing - Review & **E**diting
Vi : **Vi**sualization
Su : **Su**pervision
P : **P**roject administration
Fu : **Fu**nding acquisition

CONFLICT OF INTEREST STATEMENT

Authors state no conflict of interest.

DATA AVAILABILITY

The data that supports the findings of this study are openly available on the EvoSuite official website at SF110 Corpus of Classes | EvoSuite. The code implementation of this study available at: Dema09/PCA-aDynaMOSA-Algorithm: This repo is used for aDynaMOSA implementation (Adaptive Dynamic Many-Objective Sorting Algorithm) in Python. Please kindly request to pull the source code through this email: made.saputra@binus.ac.id.

REFERENCES

[1] S. Anand *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, Aug. 2013, doi: 10.1016/j.jss.2013.02.061.

[2] P. M. S. Bueno, M. Jino, and W. E. Wong, "Diversity oriented test data generation using metaheuristic search techniques," *Information Sciences*, vol. 259, pp. 490–509, Feb. 2014, doi: 10.1016/j.ins.2011.01.025.

[3] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2014, doi: 10.1109/TSE.2014.2372785.

[4] H. Y. Chien, C. Y. Huang, and C. C. Fang, "Applying slicing-based testability transformation to improve test data generation with symbolic execution," *International Journal of Performability Engineering*, vol. 17, no. 7, pp. 589–599, 2021, doi: 10.23940/ijpe.21.07.p3.589599.

[5] P. McMinn, "Search-based software testing: Past, present and future," in *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011*, IEEE, Mar. 2011, pp. 153–163. doi: 10.1109/ICSTW.2011.100.

[6] G. Grano, C. Laaber, A. Panichella, and S. Panichella, "Testing with fewer resources: An adaptive approach to performance-aware test case generation," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2332–2347, Nov. 2019, doi: 10.1109/TSE.2019.2946773.

[7] G. Fraser, A. Arcuri, and P. McMinn, "A memetic algorithm for whole test suite generation," *Journal of Systems and Software*, vol. 103, pp. 311–327, May 2015, doi: 10.1016/j.jss.2014.05.032.

[8] M. Zhang and H. Li, "Research on automatic generation of test data for multi-path coverage," in *Proceedings - 2021 4th International Conference on Advanced Electronic Materials, Computers and Software Engineering, AEMCSE 2021*, IEEE, Mar. 2021, pp. 1185–1188. doi: 10.1109/AEMCSE51986.2021.00242.

[9] O. Sahin, B. Akay, and D. Karaboga, "Archive-based multi-criteria artificial bee colony algorithm for whole test suite generation," *Engineering Science and Technology, an International Journal*, vol. 24, no. 3, pp. 806–817, Jun. 2021, doi: 10.1016/j.jestch.2020.12.011.

[10] Y. Liu, Y. Sun, B. Xue, M. Zhang, G. G. Yen, and K. C. Tan, "A survey on evolutionary neural architecture search," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 2, pp. 550–570, Feb. 2023, doi: 10.1109/TNNLS.2021.3100554.

[11] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation, ICST 2015 - Proceedings*, IEEE, Apr. 2015, pp. 1–10. doi: 10.1109/ICST.2015.7102604.

[12] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, Feb. 2017, doi: 10.1109/TSE.2017.2663435.

[13] A. Panichella, F. M. Kifetew, and P. Tonella, "Incremental control dependency frontier exploration for many-criteria test case generation," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11036 LNCS, 2018, pp. 309–324. doi: 10.1007/978-3-319-99241-9_17.

[14] D. Li, W. E. Wong, S. Pan, L. S. Koh, S. Li, and M. Chau, "Automatic test case generation using many-objective search and principal component analysis," *IEEE Access*, vol. 10, pp. 85518–85529, 2022, doi: 10.1109/ACCESS.2022.3198694.

[15] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 4, pp. 433–459, Jul. 2010, doi: 10.1002/wics.101.

[16] P. McMinn, "Search-based software test data generation: A survey," *Software Testing Verification and Reliability*, vol. 14, no. 2, pp. 105–156, Jun. 2004, doi: 10.1002/stvr.294.

[17] P. Tonella, "Evolutionary testing of classes," *ISSTA 2004 - Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, vol. 29, no. 4, pp. 119–128, Jul. 2004, doi: 10.1145/1013886.1007528.
[18] J. Campos, Y. Ge, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for test suite generation," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10452 LNCS, 2017, pp. 33–48. doi: 10.1007/978-3-319-66299-2_3.
[19] D. E. Goldberg, "Genetic algorithms in search, optimization, and machine learning," 1989, doi: 10.5860/choice.27-0936.
[20] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, Feb. 2012, doi: 10.1109/TSE.2012.14.
[21] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A detailed investigation of the effectiveness of whole test suite generation," *Empirical Software Engineering*, vol. 22, no. 2, pp. 852–893, Apr. 2017, doi: 10.1007/s10664-015-9424-2.
[22] S. abdullahi *et al.*, "Software testing: Review on tools, techniques and challenges," *International Journal of Advanced Research in Technology and Innovation*, vol. 2, no. 2, pp. 11–18, 2020.
[23] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using EvoSuite," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, pp. 1–42, Dec. 2014, doi: 10.1145/2685612.
[24] M. Jureczko and D. Spinellis, "CKJM extended-an extended version of tool for calculating chidamber and kemerer java metrics (and many other metrics)," *http://gromit. iiar. pwr. wroc. pl/p {_} inf/ckjm* (2011).
[25] B. Suri and S. Singhal, "Investigating the OO characteristics of software using CKJM metrics," in *2015 4th International Conference on Reliability, Infocom Technologies and Optimization: Trends and Future Directions, ICRITO 2015*, IEEE, Sep. 2015, pp. 1–6. doi: 10.1109/ICRITO.2015.7359254.

BIOGRAPHIES OF AUTHORS



Made Raja Adi Surya Saputra is a graduate student at BINUS University, Indonesia, specializing in Software Engineering. His research focuses on search-based software testing, test case optimization, and evolutionary algorithms. He has contributed to projects involving the EvoSuite framework, particularly in enhancing test generation through many-objective optimization strategies. His current work explores the integration of dimensionality reduction texhniques such as PCA into advanced search-based algorithms like aDynaMOSA. He can be contacted at email: made.saputra@binus.ac.id.



Maria Seraphina Astriani is a faculty member and senior lecturer in the Computer Science Department at BINUS University, Indonesia. She holds a Doctorate in Computer Science and a master's in information technology. Her academic background includes expertise in software engineering, object-oriented programming, human-computer interaction, and intelligent systems. She has published numerous papers in international journals and conferences, focusing on intelligent systems, software quality assurance, and adaptive technologies in education. She has been actively involved in curriculum development, research supervision, and academic management within the School of Computer Science. Her research contributions in this study involve the methodological supervision, validation, and final review of the proposed evolutionary algorithm approach for automated test case generation. She can be contacted at email: seraphina@binus.ac.id.