Improving Search-based Test Case Generation with Local Search using Adaptive Simulated Annealing and Dynamic Symbolic Execution

Dongcheng Li[1], W. Eric. Wong[1,*], Shenglong Li[2], and Matthew Chau[1]
[1]Department of Computer Science, University of Texas at Dallas, Richardson, Texas, USA
[2]School of Computer Science, China University of Geosciences, Wuhan, China
*corresponding author

2022 9th International Conference on Dependable Systems and Their Applications (DSA) | 978-1-6654-8877-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/DSA56465.2022.00047

Abstract—**DynaMOSA is an effective search-based test case generation algorithm. However, it uses an alternating variable method for local search. This method follows a greedy strategy that considers each input variable of an optimization function independently and attempts to optimize it. Some problems with this kind of search are that it can easily become stuck in the local optimal solution and its search capability becomes inadequate in the late stage of the search. Such constraints may lead to a dramatic drop in search performance. To solve these problems, this study proposed a local search algorithm based on adaptive simulated annealing and symbolic path constraints to generate test cases with high coverage for multiple testing criteria within a limited time budget. On the one hand, the simulated annealing algorithm was selected to explore the neighborhood of candidate solutions during the search. On the other hand, various simulated annealing operators were designed for the search of each statement to enhance the applicability of the algorithm in various programs. Additionally, symbolic execution was introduced as a supplement to the simulated annealing algorithm for local search to generate test cases for inputs with complex structures. Furthermore, the proposed algorithm was implemented in EvoSuite framework. From an SF110 open-source benchmarking dataset, 49 projects or 110 classes were selected according to the complexity and number of objectives of each class under test to conduct the experiments. The proposed algorithm outperformed the original algorithm in generating high coverage test cases on most projects in terms of line, mutation, and multicriteria coverage as well as search efficiency.**

Keywords-automated test case generation; local search; adaptive simulated annealing algorithm; dynamic symbolic execution; EvoSuite

I. INTRODUCTION

Software testing is a critical means to ensure the quality and reliability of a software system [1], especially in safety-critical systems. Test case generation is the foundation of software testing and test cases are used to comprehensively cover software functions. A complete test case should cover situations in a software program and code as many as possible [2, 3]. Thus, software functions can be sufficiently validated.

However, a global search may struggle to generate the specific values necessary for covering challenging parts of the program, while a local search that uses the alternating variable method (AVM) [4] with a greedy strategy can easily cause the algorithm to become stuck in a local optimal solution. Therefore, this study proposed a local search algorithm that combines adaptive simulated annealing and symbolic path constraints to perform the neighborhood search of optimal solutions. In an automatic test case generation problem, an optimal solution contains test cases that achieve high code coverage, and a test case consists of many test statements. Therefore, it may be beneficial to search for individual test statements. To avoid falling into local optimal solutions, an adaptive simulated annealing algorithm was proposed to explore the parameters of each statement effectively. In addition, a local search based on symbolic path constraints worked through the constraints encountered during the execution of test cases, thus achieving high code coverage. In this study, the solution space of test case generation was effectively searched through a trade-off of resource consumption between global and local search. Moreover, the proposed algorithm was compared with the state-of-the-art test case generation algorithm and tested on SF110 [5] open-source benchmarking datasets. Thus, the effectiveness of the proposed algorithm was verified.

The remainder of the paper is organized as follows. Section 2 presents related studies; Section 3 describes the proposed local search for the automatic generation of test cases; and Section 4 describes the experimental setup and analyzes the results. Finally, Section 5 presents the conclusions.

II. RELATED STUDIES

A. Test Case Generation based on Local Search

Local search is a supplement to global search. A heuristic search algorithm conducts a constant search in global space; however, it is difficult for a global search algorithm with a large step size in search and a random mutation method to step out of the current local optimal solution. The key idea behind local search is not to focus the search in the entire space of all candidate solutions, but rather on a set of feasible solutions obtained by the global search algorithm. It emphasizes the process of performing a local search in the neighborhood of current solutions to produce new feasible solutions [6]. Popular local search algorithms include hill climbing, simulated annealing, tabu search, and variable neighborhood search [7].

2767-6684/22/$31.00 ©2022 IEEE
DOI 10.1109/DSA56465.2022.00047

As a local search algorithm is only concerned with the neighborhood of feasible solutions, it is incapable of searching the global space. By contrast, the global search algorithm's capability to search the neighborhood of feasible solutions is insufficient as it needs to search the entire global space. Therefore, a memetic algorithm is created by combining global and local search algorithms [8-10].

In automatic test case generation, the memetic algorithm was used to generate test data [11]. Sharifipour et al. [11] introduce a memetic ant colony optimization algorithm for generating structural test data. Wang [12] combined a genetic algorithm with hill climbing and applied in test case generation. Through experiments, the effects of the memetic algorithm were proven to be superior to those of the hill climbing or genetic algorithm alone. Liaskos and Roper [13] proposed another memetic algorithm that combined the genetic algorithm and artificial-immune-system algorithm with clonal selection for test case generation. The results of the experiments indicated that the proposed algorithm significantly improved the coverage of the generated test cases for the projects under test.

Fraser et al. [14] combined the genetic algorithm with the AVM for test case generation. The AVM was selected as the local search algorithm to perform a local search for different types of data in a class under test, including strings, arrays, etc. Through combining the global and local search algorithms, the coverage of the test cases was significantly improved. In a genetic algorithm, the reproduction operation in the search process is based on chromosomes. Although such a coarse-grained search contributes to the search in the global space, it is difficult to effectively search the neighborhood of optimal solutions. In this respect, the AVM alleviates the deficiencies of genetic algorithms.

Reinforcement learning has also been introduced to the local search for automatic test case generation. Esnaashari and Damia [15] proposed a memetic algorithm that uses reinforcement learning to perform a local search. The reinforcement learning module receives all optimal test cases, which are generated through the genetic algorithm, and attempts to generate new and improved test cases. The experimental results demonstrated that the proposed method can generate more quality test cases faster than many existing heuristic algorithms.

Moreover, studies have indicated that local search is still a direction worth studying. The most popular search-based automatic test case generation framework is EvoSuite [16], the local search strategy of which is the AVM. Although such a memetic algorithm has a stronger search capability compared with the global search algorithm alone, it is less likely to eliminate the limitations of current solutions and avoid stuck into a local optimal solution. In this case, local search may be ineffective.

B. Test Case Generation based on Symbolic Execution

Ramamoorthy et al. [17] was the first to propose an automated test case generation technique based on symbolic execution (SE). According to SE, each branch is seen as a path condition for passing that branch. The results returned are a set of equality and inequality constraints over input variables of the program, which define a subset in the search space and execute the selected path. In this case, a test data generator attempts to identify a test input from the subset space that meets all constraints [18]. If no such input can be confirmed, the corresponding path is infeasible. Then, a set of inputs that satisfy various path constraints is obtained to generate test cases. Moreover, solving the elements in a path constraint set is a complicated mathematical constraint satisfaction problem. At present, common constraint solvers include Z3, SMT, and CVC3 [19]. These SE methods would not execute the program under test; instead, the program undergoes simulated execution with symbolic values obtained after resolving. Therefore, this type of SE is also referred to as static SE. As the program size becomes larger, path constraints sent back by SE may become gradually longer. In this case, the solving time of the constraint solver may increase, making it difficult to determine solutions that satisfy such constraints from a subset space, further leading to a decline in test case generation efficiency [20, 21].

To improve the performance of SE-based automated test case generation, dynamic SE (DSE) was proposed [22]. In contrast to static SE, DSE selects specific values as the input and uses a program executor to execute the tested program. From the current running status, DSE also collects decision statements that cannot pass the corresponding branch, namely path constraints. Ultimately, DSE aims to solve such path constraints with the help of a path constraint solver and generate new concrete input. This operation is repeated to continuously execute programs [23].

Furthermore, compared with static SE, DSE does not use symbolic values as input, but rather executes specific input values in the program. Thus, it can reduce both time costs and computing resources during each execution, leading to a rather high performance. However, the set of paths eventually generated is smaller than that of all paths because DSE continually searches program paths thoroughly [24]. Popular SE-based automatic test case generation frameworks include Java Bytecode Symbolic Executor (JBSE) [25], SUSHI [26], and TARDIS [27], which have all been developed based on SE.

More specifically, JBSE is the first symbolic executor especially designed for processing programs executed on complex heap inputs. This is an advanced method that implements novel heap exploration logic and handles data structure constraints. Through this method, incremental checks of constraints over complex heap inputs has been proven effective [25].

SUSHI, applicable to programs with complicated structured inputs, is a test case generator for generating high-coverage test cases. It uses SE to generate path conditions that accurately represent the relationship between the program path and the input data structure. Moreover, the path condition is transformed into the fitness function for the search. Through combining it with an appropriate search algorithm, the test case generation performance can be improved.

TARDIS combines DSE, search-based testing, and machine learning and is capable of efficiently generating a complete test suite at a class level. The main idea behind this method is to explore the path space of the objective program

by DSE, and to search and generate complete test cases through a genetic search algorithm guided by fitness functions that satisfy symbolic path conditions. Additionally, classification algorithms are also adopted to prioritize symbol equations that are more likely to correspond to feasible program paths. Through a path selection strategy based on the classification algorithm, the number of feasible paths of analytical procedures is increased, thus further increasing the number of test cases generated. Therefore, TARDIS has been proven to be a valid approach.

As JBSE, SUSHI, and TARDIS indicate, automatic test case generation technology based on SE is advancing rapidly. However, SE still faces certain challenges, which are described as follows [28]: (1) path selection: when the program complexity increases, the number of execution paths that SE explores exponentially increases as the number of program branches increases, which may lead to the problem of exponential path explosion; (2) constraint solving: this restricts the efficiency of SE to some extent; and (3) memory modeling: how program statements are precisely translated into symbolic constraints has a large impact on the coverage of test cases generated through SE.

III. Local Search Algorithm for Test Case Generation

As search algorithms are continuously optimized, a population may consist of many similar individuals in later phases. In this case, all individuals in the population will begin to evolve in the direction of optimization, and the search performance of such a genetic operation will significantly decline. To specifically address this problem, the present study designed a local search algorithm for test case generation which plays a major role in the neighborhood search of feasible solutions, preventing the algorithm from falling into a local optimum, improving the convergence of optimal solutions, and eventually enhancing search efficiency. Through local search, new solutions available for search can be generated randomly in the neighborhood of feasible solutions. If new solutions outperform preceding solutions, the algorithm continues to search in the neighborhood of new solutions; otherwise, it returns to the previous solution. Such a search strategy is capable of rapidly approaching the optimal solution.

In the EvoSuite framework, the AVM was applied in the local search of test cases, producing rather strong results. A key advantage of DynaMOSA [29] is that it is a novel many-objective search method based on preference ordering and the dynamic selection of optimization objectives. Presently, DynaMOSA is one of the most advanced search-based algorithms for test case generation, achieving excellent results in international search-based software testing tool competitions each year [30]. However, it still has an inadequate local search capability, performs reproduction operation on chromosomes, and does not search individual genes. Furthermore, the difficulty in effectively and extensively exploring neighborhoods of superior individuals makes the search falling into a local optimum much more likely.

A. Local Search based on the Alternating Variable Method

The AVM [31], proposed by Korel, is a local search algorithm similar to hill climbing and adopts a greedy strategy. If a local search is conducted for a given integer, the AVM will select the current integer as the starting point to make exploratory movements, in which case the step size will increase by 1. If the corresponding fitness value increases, the step size will increase by 2, and if the fitness continues to rise, the step size will increase by 4. Moreover, the doubling of the step size will not stop until the fitness function stops increasing. If a "hill" is found, which signifies the existence of a peak value in the neighborhood, the growth of the previous step size (e.g., +1, +2, or +4) is adopted by the AVM as its new starting point to make exploratory movements again. If explorations that start from a starting point under a condition of a step size growth of 1 do not cause the fitness function to increase in the process of such a search, the same operation will be conducted in the negative direction of the current search at this point. In this case, the step size will increase by −1, −2, or −4 and so forth. When the fitness function that corresponds to a step size of −1 of the starting point no longer increases, the operation is performed against the abovementioned negative direction. This process is referred to as the alternating variable. During exploratory movements, the search by the algorithm stops once fitness functions corresponding to step size increases of +1 or −1 of a starting point are not elevated, as illustrated in Figure 1.

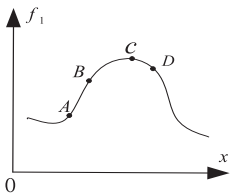

Figure 1. Schematic diagram of the alternating variable method

Because of the greedy strategy, the traditional AVM can easily fall into a local optimum. Once the first peak value is searched, it will be reported back and the search will thus end. However, this peak value may not be the global maximum. If the first starting point is point C in Figure 1, the algorithm will directly converge and not perform a local search.

B. Adaptive Simulated Annealing Based Local Search for Test Case Generation

The simulated annealing algorithm [7] inspired by the following physical law: The motion state of molecules changes along with temperature variations. Under an ultra-high temperature, matter molecules move fiercely. As the temperature drops, the molecular movement slows and the structure of the corresponding matter is inclined to be stable. Accordingly, molecular kinetic energy in the matter tends to decline. However, an excessively sharp temperature drop may sometimes incur condensation of matter molecules. Molecules in fierce motion may suddenly condense, which is deemed a pseudo stable state. In this context, the inter-molecular distance can be found. To sufficiently reduce such distance, the temperature of the matter should be raised so that the matter cannot condense. Subsequently, temperature reduction is conducted to generate a stabler matter state. The

simulated annealing algorithm deals with the condensation state of matter. As the stable state cannot be directly identified as true or false, the Metropolis criterion was proposed by physicists, which uses a probability calculation formula to determine whether the stable state can be accepted.

Objective optimization frequently involves feasible solutions that may fall into a local optimum. For example, the AVM may quit the search once a peak value appears. Such a problem can be avoided by the simulated annealing algorithm, which defines both the initial temperature and the rate at which the temperature falls. The algorithm selects poor solutions surrounding the peak value at a certain probability. Thus, feasible solutions can be prevented from falling into a local optimum. Moreover, the probability may decline as the temperature drops, which ensures the convergence of the algorithm.

1) Simulated Annealing Algorithm Applied in the Local Search of Statements

The simulated annealing algorithm was adopted to perform a neighborhood search. Specific procedures for applying the algorithm in the local search of statements are presented in Figure 2.

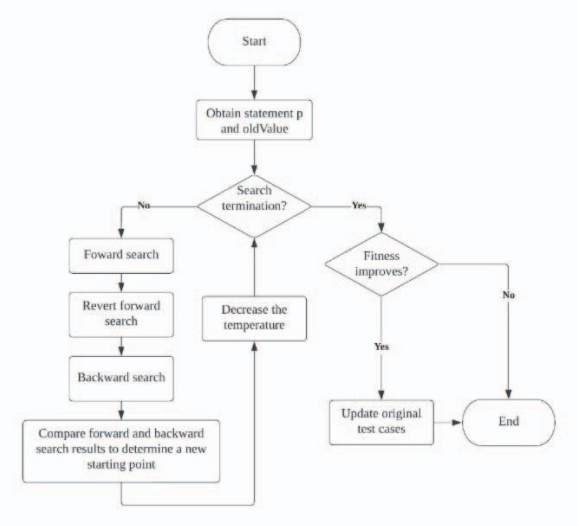

Figure 2. Simulated annealing -based search of statement values

In this algorithm, poor search results are received at a probability based on the current temperature. The forward search and backward search refer to a search at incremental and descending step sizes, respectively. Moreover, the temperature of the current iteration serves as the initial temperature of the forward and backward search to perform a simulated annealing-based search.

Through forward and backward search, optimal values in forward positive and backward negative neighborhoods were acquired under the circumstance of the current value serving as the starting point. Then, the optimal point with a greater value between the two was selected as the new starting point to search again. A cooling operation also existed in an iteration of the same level, which ensured the final convergence of the search.

2) Adaptive Annealing Operator

A test suite usually contains statements of various datatypes. Because of the influence of internal codes, different programs under test exhibit different sensitivities to a local search for such statements. For example, a local search of floating point data is invalid for some programs but effective for integer data. Moreover, as the objectives are covered, previous local searches of strings may produce good results. However, the local search of strings in this case is ineffective. For the traditional simulated annealing algorithm, all local searches are treated equally and relevant search resources are allocated equally. Apparently, this is unsuitable for test case generation. Therefore, an adaptive parameter should be defined for statements of each datatype. Then, the adaptive parameters are used to adaptively adjust resource allocation for a local search of such statements. In the simulated annealing algorithm, the annealing parameter is the most effective parameter for controlling the search resource. For this reason, adaptive cooling operators are designed for statements in most datatypes to control resource allocation for a local search of test case generation; thus, the search performance of the algorithm can be improved. The specific implementation process is presented in Figure 3.

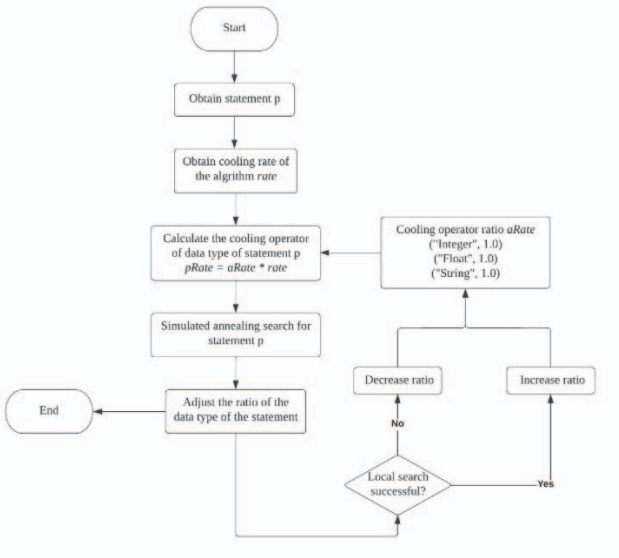

Figure 3. Implementation of the adaptive annealing operator

C. Local Search of Basic Datatypes

EvoSuite, a test case generation framework, uses many-objective search to simultaneously optimize the entire test suite. Each individual in a population is a test suite and a gene is a line of code statement. In this study, local search was conducted for statements of basic datatypes, including string, integer, short, long, byte, character, float, double, boolean, and enum. Simulated annealing was selected as the local search algorithm to prevent it from falling into a local optimum. Given that different programs under test exhibit different sensitivities to local search as a whole or the local search of diverse statements, adaptive annealing parameters were designed for various statement types to control

applications of local search and properly allocate various computing resources.

In this study, basic datatypes were divided into four categories, namely string, integer (e.g., integer, short, long, byte, and character), floating point (e.g., float and double), and iterating (e.g., Boolean and Enum). Different local search methods were designed for each category.

1) Local Search for Strings

Data structures of strings are not as complicated as those of other datatypes. A string is composed of a set of unconstrained characters, but it has the most diversified components and is also the most unordered. To improve the string search efficiency of the algorithm, we needed to clarify whether changes in string values affect the fitness of the current individual before local search. If an impact exists, a local search of strings would be conducted, followed by character removal, character addition, and character deletion.

Character removal: After characters traversed (or iterated) and removed from the original string, the fitness function of the current individual is calculated. If the fitness is improved, the local search of the string should be ended and the modified string returned.

Character addition: To ensure the search performance of the algorithm, characters should be added to both the head and tail of the string. Starting from valid characters 9 to 127 of the ASCII table, the characters are constantly added to the string. After each addition operation, the fitness of the current individual is calculated and, if it is improved, the local search of the string should be ended and the modified string value returned.

Character substitution: In the event that all characters of an entire string are traversed, the simulated annealing algorithm is adopted to perform a local search of each character, and since the local search has been performed for all characters, the algorithm ends.

2) Local Search of the Integer

The easiest implementation of local search is fulfilled for data in the integer category. The simulated annealing algorithm directly performs a local search for integer.

3) Local Search of the Floating Point Data Category

Floating point data should be divided into two parts for optimization, namely integers and fractions. The search for integers is conducted by following the abovementioned procedure. As for fractions, the simulated annealing algorithm is selected to search fractions of the corresponding precision (i.e., single or double precision).

3) Local Search of Iterating Datatypes

Data in the iterating category are a set of enumerable data with a limited number of classes. Therefore, the fitness of the current individuals is calculated after substitution of the data in a statement through traversing or iterating. If the fitness is improved, the current local search is ended and current values are reserved. For example, “true” is replaced with “false” in a Boolean statement, or “male” is substituted by “female” in an enumeration type.

D. Local Search for Test Case Generation based on Symbolic Path Constraints

SE lays an essential foundation for automated test case generation. However, it is generally not used in test case search alone as it requires massive computing resources. In the TARDIS framework, EvoSuite is adopted to execute test cases, thus obtaining path constraints of the corresponding program to depict a conditional tree of paths. Then, test cases are generated with the help of such a conditional tree. In EvoSuite, DSE was also used to improve the mutation operation of the genetic algorithm.

In this study, JBSE was used as the medium to acquire and traverse or iterate a constraint path set of current test cases. Then, the Z3 constraint solver was adopted and attempted to solve constraint paths of the current test cases and generate new ones. Only if the fitness of newly generated test cases is superior to the preceding fitness, can novel test cases be accepted; otherwise, they will be abandoned. For the specific implementation process, please refer to Figure 4.

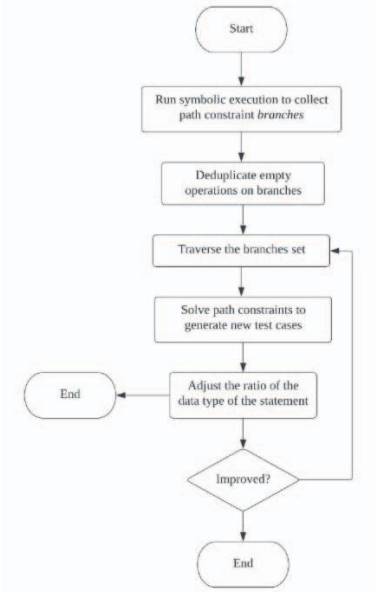

Figure 4. Implementation of symbolic execution -based local search

SE is capable of effectively generating test cases with a complex structure, although its constraint-solving efficiency is rather low and may lead to the problem of path explosion. DSE, selected for local search in the proposed method, is a supplement to the simulated annealing algorithm and only conducts a local search targeted at a few individuals each time. Thus, the problem of resource consumption can be eased.

1) Test Case Generation by Local Search Combining Adaptive Simulated Annealing and Symbolic Execution

Local search is a supplement to global search. Only through the reasonable allocation of search resources and balancing of the trade-off between local and global search can the search performance of the algorithms be effectively improved and test case generation efficiency increased. Otherwise, adverse effects may be produced.

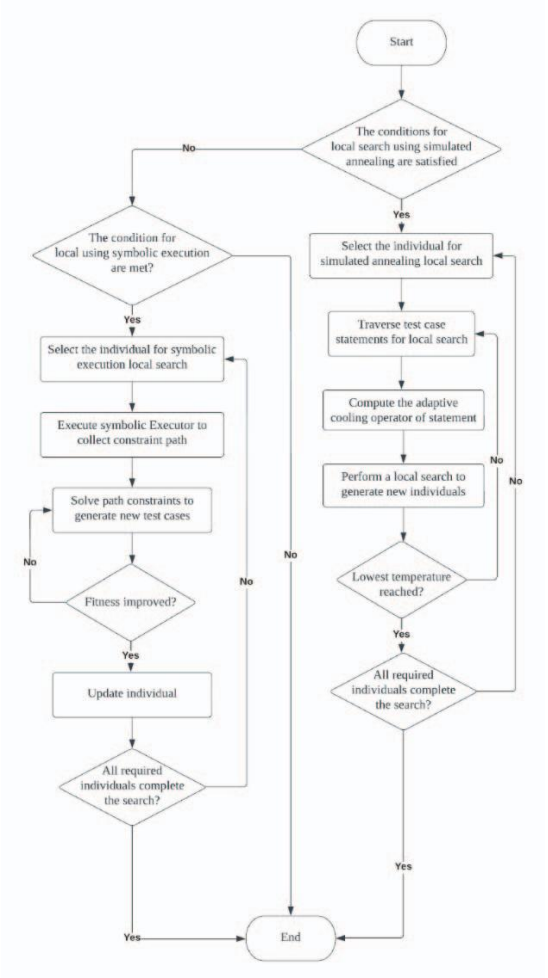

Figure 5. Flow diagram of proposed local search

To measure global and local search for test cases, critical parameters were defined in this study (as shown in Table 5). Where, the running time of the local search ensures that the corresponding resource consumption is within the established constraint. The interval algebra of simulated annealing or SE-based local search is designed to guarantee differences in individuals of each local search and reduce the possibility of repeated local searches of the same individual. The number of individuals in simulated annealing and SE-based local search is established to ensure that the algorithm only performs a local search of individuals that exhibit the optimal performance after ordering in the current population. The local search probability of the simulated annealing algorithm represents the probability of the adaptive simulated annealing algorithm being used for local search. Each time the simulated annealing algorithm is applied, it is adjusted according to its variation rate. Once values of those parameters are set, whether the local search algorithm is executed in each iteration is determined. The specific flow of this local search algorithm is presented in Figure 5.

IV. Experimental Design and Result Analysis

To evaluate the actual performance of the improved algorithm, various experiments were designed and conducted on SF110 dataset. The experimental results of the proposed method were analyzed in comparison with those produced by the AVM in EvoSuite. In this study, DynaMOSA served as the global search algorithm and, based on this, the proposed improved local search algorithm based on adaptive simulated annealing was named SA-DynaMOSA. The original local search algorithm based on the AVM was therefore called AVM-DynaMOSA.

The experiments designed and conducted in this study were primarily aimed at answering the following research questions (RQs) for multi-criteria many-objective test case generation with local search:

- RQ 1: How does SA-DynaMOSA perform compared with AVM-DynaMOSA from the perspectives of line, branch, and mutation coverage of the generated test cases?
- RQ 2: How does SA-DYNAMOSA perform compared with AVM-DynaMOSA from the perspectives of the multicriteria total coverage of the generated test cases and the search efficiency of the algorithm?

A. Experimental Design

1) Evaluation Metrics

To compare the test case search capabilities of relevant test case generation algorithms, the following evaluation metrics were selected to test and compare both the original and improved algorithms.

Testing coverage: This metric represents the adequacy of the final test suite in testing the program, including line, branch, mutation, and multi-criteria coverage [29]. Multi-criteria coverage refers to the coverage of all objectives (the aggregation of all branches, lines, and mutants) for test case generation. The metrics for calculating those types of coverage are as follows:

$$\begin{cases} statement\_coverage = \dfrac{\text{Number of covered statements}}{\text{Total number of statements to be covered}} \times 100\% \\ branch\_coverage = \dfrac{\text{Number of covered branches}}{\text{Total number of branches to be covered}} \times 100\% \\ mutation\_coverage = \dfrac{\text{Number of killed mutants}}{\text{Total number of mutants to be killed}} \times 100\% \\ multicriteria\_coverage = \dfrac{\text{Number of covered objectives}}{\text{Total number of objectives to be covered}} \times 100\% \end{cases} \quad (1)$$

Search efficiency: By using the coverage percentage collected every second throughout the entire duration of test case generation, the search efficiency of an algorithm can be calculated using the following equation:

$$AUC = \frac{\sum_{i=0}^{120} [\text{cov}_i + \text{cov}_{i+1}] * \Delta Time}{2 * TotalTime} \quad (2)$$

where cov_i refers to the percentage of coverage at time i, and cov_{i+1} refers to the percentage of coverage at time i+1; $\Delta Time$ represents the time interval, which was one second in this study; and $TotalTime$ is the total running time of an algorithm, which was 120 seconds.

The rate of increase for different testing coverages of the improved algorithm from the original algorithm was calculated based on the following equation:

$$Increase\_rate = \frac{\text{Value}_{\text{after}} - \text{Value}_{brfore}}{\text{Value}_{brfore}} \times 100\% \quad (3)$$

In this study, a proportional change of no more than 0.5% was considered a nonsignificant change or no change. Therefore, projects or classes with an increase rate above or below 0.5% were removed when comparing the coverage and search efficiency of the algorithms.

2) Experimental Setup

The proposed algorithm was implemented in the EvoSuite framework and compared with DynaMOSA. Experiments were conducted on the Windows platform. The setup for the experimental environment is presented in Table 1.

Table 1. Experimental environment

Hardware	Intel(R) Core(TM) i5-6500 CPU@ 3.20GHz
	RAM 12GB
Software	Windows10
	Java 1.8

1) Datasets

In this study, CKJM was first used to analyze the complexity of all projects in the SF110 dataset. Then, projects with high complexity were selected to verify the performance of the proposed test case generation algorithm.

Table 2. List of projects and classes under test

Project name	Classes	Project name	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	heal	3
follow	2	feudalismgame	1
lilith	3	liferay	2
lotus	1	pdfsam	1
resources4j	1	firebird	3
diebierse	1	-	-

The SF110 datasets contain 110 statistically representative open-source Java projects collected from SourceForge. SourceForge is a popular open-source repository with over 300,000 projects and more than 2,000,000 registered users. It includes all types of projects with diversified functions. Thus, SF110 was appropriate for testing the performance of test case generation algorithms and frameworks. However, because of the massive number of projects and classes in the dataset, experiments using all classes in the dataset may take much time and incur high costs. More than 50% of classes in the SF110 dataset had low program complexity, which means that a simple method call is capable of covering most of the objectives in such classes. Therefore, CKJM, a program that calculates several object-oriented metrics (e.g., weighted methods per class, depth of Inheritance Tree, coupling between object classes, lack of cohesion in methods, response for a class, and number of children) by processing the bytecode of compiled Java files, was adopted to analyze the complexity of all classes in the SF110 dataset.

Specifically, the data sets used in the experiments was selected using the following procedure. First, the code complexity of all classes in the SF110 dataset was calculated using CKJM and the total number of objectives covered in each class was obtained using EvoSuite framework. Next, classes under test were sorted according to their complexity and number of objectives covered; projects containing more complex classes with a greater number of objectives were selected and included in the final test data sets. Considering that very few projects or classes behaved abnormally during the experiment, such projects or classes were excluded from the experiment. The projects and the number of classes selected from each one are presented in Table 2.

As Table 2 indicates, 49 of the 110 projects were selected from the SF110 dataset. In total, 110 classes served as the fundamental test data for the experiments conducted in this study.

To compare the performance of SA-DynaMOSA with AVM-DynaMOSA, experiments were conducted and repeated 10 times for each test class in the selected data sets. The running results were averaged for comparison and analysis.

4) Experiment Setup

a) Parameter Settings for the Proposed Algorithm

The default parameters established for the DynaMOSA algorithm in EvoSuite have rather good performance, as empirically validated by other scholars. Therefore, to preserve the high quality of DynaMOSA and ensure that the experiments were conducted in a controlled setting, the same parameter settings (presented in Table 3) for global search were adopted in the proposed algorithm.

Table 3. Parameter settings for the global optimization algorithm

Parameter name	Value
Population size	100
Search time	120000 seconds (2 minutes)
Covering objectives	Total number of lines, branches, and mutants to be covered
Crossover rate	0.75

Table 4. Parameter settings for AVM-DynaMOSA

Parameter name	Value
Initial local search probability	1
Local search probability change rate	2

Table 5. Parameter settings for SA-DynaMOSA

Parameter name	Value
Simulated annealing local search interval	3
Simulated annealing local search individual number	5
Simulated annealing local search initial probability	1
Simulated annealing local search probability change rate	2
Simulated annealing local search initial temperature	100
Simulated annealing local search lowest temperature	1
Initial cooling rate of each type of statement	0.9
Change rate of cooling rate of various types of statements	0.8
Symbolic execution local search interval	5
Symbolic execution local search individual number	5

b) Parameter Settings for Local Search Algorithms

The specific parameter settings of AVM-DynaMOSA and SA-DynaMOSA are listed in Tables 4 and 5, respectively.

B. Experimental Results and Analyses for Test Case Generation based on Local Search

1) Different Coverage Achieved for Each Project (RQ1)

a) Line Coverage Achieved for Each Project

The line coverage of AVM-DynaMOSA and SA-DynaMOSA is presented in Table 6.

b) Branch Coverage Achieved for Each Project

The branch coverage of AVM-DynaMOSA and SA-DynaMOSA is presented in Table 7.

c) Mutation Coverage Achieved for Each Project

The mutation coverage of AVM-DynaMOSA and SA-DynaMOSA is presented in Table 8:

Figure 6 presents the number of projects with superior and inferior results according to the line, branch, and mutation coverage of SA-DynaMOSA and AVM-DynaMOSA.

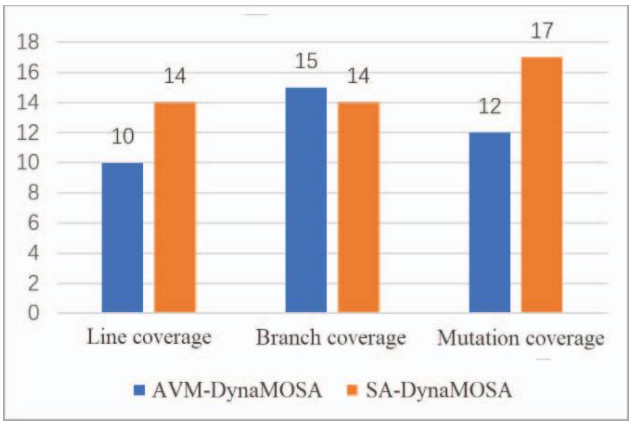

Figure 6. Number of projects with superior and inferior results in multiple coverage criteria

Table 6. Mean line coverage achieved for each project

Project Name	Classes	Statement Coverage		Increase-rate	Statistics	
		AVM-DYNAMOSA	SA-DYNAMOSA		SA-DYNAMOSA versus AVM-DYNAMOSA	
					%>0.50	%<-0.50
gfarcegestionfa	2	73.59%	78.79%	7.07%	100.00%	
imsmart	2	84.14%	90.67%	7.75%	50.00%	
beanbin	1	85.25%	90.16%	5.77%	100.00%	
inspirento	3	90.71%	92.61%	2.10%	33.33%	
byuic	2	47.14%	47.56%	0.89%	50.00%	
gangup	4	95.17%	95.82%	0.69%	25.00%	
apbsmem	7	87.84%	91.56%	4.23%	28.57%	
diffi	2	92.07%	92.98%	0.99%	50.00%	
lilith	3	82.02%	81.49%	-0.65%		33.33%
resources4j	1	75.76%	86.36%	14.00%	100.00%	
hft-bomberman	2	92.92%	95.01%	2.24%	50.00%	
dom4j	5	86.57%	85.01%	-1.79%		80.00%
lhamacaw	5	63.77%	65.26%	2.34%	40.00%	
battlecry	2	91.55%	89.59%	-2.14%		50.00%
wheelwebtool	7	57.61%	56.13%	-2.56%		57.14%
xbus	2	83.04%	80.86%	-2.62%		50.00%
at-robots2-j	2	83.90%	84.50%	0.71%	50.00%	
jiggler	5	63.90%	94.11%	47.28%	80.00%	
dcparseargs	1	100.00%	90.36%	-9.64%		100.00%
jcvi-javacommon	6	65.28%	65.86%	0.89%	50.00%	
quickserver	1	96.53%	95.14%	1.44%		100.00%
weka	3	66.38%	55.89%	-15.81%		100.00%
liferay	2	74.52%	73.14%	-1.86%		50.00%
firebird	3	94.06%	92.73%	-1.41%		66.67%
Mean over all projects		84.29%	84.84%			
No. cases SA-DYNAMOSA significantly better than AVM-DYNAMOSA						14 (20.59%)
No. cases SA-DYNAMOSA significantly worse than AVM-DYNAMOSA						10 (14.71%)

From Tables 6–8 and Figure 6, compared with AVM-DynaMOSA, the mean coverage of SA-DynaMOSA increased by 0.55%, 0.69%, and 0.94% in terms of line, branch, and mutation coverage, respectively. SA-DynaMOSA achieved equivalent or higher coverage on most projects in terms of line and mutation coverage; only a few projects exhibited lower coverage. More specifically, compared with AVM-DynaMOSA, the coverage achieved through our approach on average was significantly higher in 20.59% of the classes tested for line, 20.59% for branch, and 25.00% for mutation. The improvement in mutation coverage was greatest among three coverage criteria. Among the projects with superior coverage achieved by our proposed algorithm, *jiggler* exhibited the most significant increases, of 63.61% on average, in line, branch, and mutation coverage; by contrast, *weka* exhibited the greatest decreases, of 16.16% on average, in line, branch, and mutation coverage. However, the performance of AVM-DynaMOSA and SA-DynaMOSA exhibited small differences in terms of branch coverage. That is, among 49 projects, SA-DynaMOSA outperformed AVM-DynaMOSA on 14 projects, but it was inferior to AVM-DynaMOSA on 15 projects. The rest of the projects achieved equivalent coverage using both algorithms. Moreover, the mean coverage achieved by SA-DynaMOSA still surpassed that of AVM-DynaMOSA. From the aforementioned results, this study concluded that the SA-DynaMOSA algorithm has the potential to effectively improve line, branch, and mutation coverage for multi-criteria test case generation.

Table 7. Branch coverage achieved for each project

Project Name	Classes	Branch coverage		Increase-rate	Statistics	
		AVM-DYNAMOSA	SA-DYNAMOSA		SA-DYNAMOSA versus AVM-DYNAMOSA	
					%>0.50	%<-0.50
gfarcegestionfa	2	67.59%	79.55%	17.70%	100.00%	
imsmart	2	76.11%	86.11%	13.14%	50.00%	
beanbin	1	78.72%	82.98%	5.41%	100.00%	
byuic	2	40.08%	40.81%	1.82%	50.00%	
apbsmem	7	88.35%	92.29%	4.45%	28.57%	
schemaspy	3	68.65%	69.35%	1.03%	66.67%	
diffi	2	90.23%	91.43%	1.33%	50.00%	
lilith	3	80.91%	79.90%	-1.24%		33.33%
resources4j	1	70.27%	81.08%	15.38%	100.00%	
jhandballmoves	3	96.53%	100.00%	3.60%	33.33%	
hft-bomberman	2	94.68%	94.12%	-0.59%		50.00%
dom4j	5	83.93%	81.70%	-2.66%		80.00%
openjms	2	73.61%	72.76%	-1.15%		50.00%
lhamacaw	5	78.01%	78.91%	1.15%	20.00%	
echodep	2	83.33%	81.41%	-2.31%		50.00%
battlecry	2	84.70%	80.15%	-5.37%		50.00%
openhre	3	92.96%	92.35%	-0.65%		33.33%
twfbplayer	2	91.09%	90.32%	-0.84%		50.00%
wheelwebtool	7	66.90%	65.76%	-1.72%		71.43%
xbus	2	81.32%	79.28%	-2.51%		50.00%
at-robots2-j	2	82.86%	83.36%	0.60%	50.00%	
jiggler	5	55.30%	92.80%	67.81%	80.00%	
dcparseargs	1	96.25%	91.25%	-5.19%		100.00%
jcvi-javacommon	6	64.16%	64.97%	1.27%	50.00%	
quickserver	1	95.00%	91.67%	-3.51%		100.00%
heal	3	80.59%	78.83%	-2.18%		66.67%
weka	3	62.44%	51.25%	-17.91%		100.00%
liferay	2	69.99%	68.24%	-2.50%		50.00%
firebird	3	82.89%	83.83%	1.13%	66.67%	
Mean over all projects		81.96%	82.65%			
No. cases SA-DYNAMOSA significantly better than AVM-DYNAMOSA						14 (20.59%)
No. cases SA-DYNAMOSA significantly worse than AVM-DYNAMOSA						15 (22.61%)

Table 8. Mutation coverage achieved for each project

Project Name	Classes	Mutation Coverage		Increase-rate	Statistics	
		AVM-DYNAMOSA	SA-DYNAMOSA		SA-DYNAMOSA versus AVM-DYNAMOSA	
					%>0.50	%<-0.50
gfarcegestionfa	2	70.67%	81.48%	15.29%	100.00%	
water-simulator	2	91.16%	91.71%	0.61%	50.00%	
imsmart	2	73.33%	90.00%	22.73%	50.00%	
jdbacl	1	94.17%	97.50%	3.54%	100.00%	
beanbin	1	91.24%	93.43%	2.40%	100.00%	
inspirento	3	90.82%	91.98%	1.28%	33.33%	
byuic	2	39.42%	40.01%	1.50%	50.00%	
gangup	4	98.13%	97.54%	-0.60%		25.00%
apbsmem	7	94.14%	96.47%	2.48%	14.29%	
bpmail	1	96.25%	97.50%	1.30%	100.00%	
corina	5	75.46%	74.85%	-0.81%		60.00%
diffi	2	86.94%	89.65%	3.11%	50.00%	
lilith	3	76.69%	75.57%	-1.45%		66.67%
lotus	1	98.79%	97.18%	-1.63%		100.00%
diebierse	1	99.47%	100.00%	0.53%	100.00%	
jhandballmoves	3	98.30%	99.05%	0.76%	33.33%	
dom4j	5	80.88%	80.05%	-1.02%		60.00%
openjms	2	81.42%	80.04%	-1.70%		50.00%
lhamacaw	5	61.15%	62.63%	2.43%	20.00%	
battlecry	2	87.49%	85.04%	-2.80%		50.00%
fim1	2	54.93%	54.49%	-0.81%		50.00%
wheelwebtool	7	61.17%	60.67%	-0.82%		42.86%
javathena	1	93.42%	94.74%	1.41%	100.00%	
xbus	2	78.89%	77.47%	-1.80%		50.00%
jiggler	5	49.11%	88.76%	80.73%	80.00%	
jcvi-javacommon	6	56.45%	56.77%	0.57%	33.33%	
quickserver	1	88.30%	86.91%	-1.58%		100.00%
heal	3	81.22%	82.69%	1.81%	33.33%	
weka	3	69.33%	59.08%	-14.77%		66.67%
Mean over all projects		82.68%	83.62%			
No. cases SA-DYNAMOSA significantly better than AVM-DYNAMOSA						17 (25.00%)
No. cases SA-DYNAMOSA significantly worse than AVM-DYNAMOSA						12 (17.65%)

2) Multi-Criteria Coverage and Search Performance Achieved by the Algorithm (RQ2)

a) Multi-Criteria Coverage Results

The multi-criteria coverage results are compared between AVM-DynaMOSA and SA-DynaMOSA in Table 9. As Table 9 indicates, the multi-criteria coverage achieved by SA-DynaMOSA was improved compared with that achieved by DynaMOSA. The proposed algorithm achieved higher coverage on 19 (33.93%) of the projects under test, with 12 (21.43%) projects exhibiting lower coverage. Among the projects with superior coverage achieved by the proposed algorithm, *beanbin* exhibited the most significant increases (21.13%), whereas *weka* exhibited the most significant decreases (17.76%) for multi-criteria coverage. For the remaining projects, the multi-criteria coverage achieved by the proposed algorithm was the same as that achieved by AVM-DynaMOSA. Moreover, the mean multi-criteria coverage of SA-DynaMOSA increased by 0.52% compared with that of AVM-DynaMOSA. From the aforementioned results, this study concluded that for multi-criteria coverage, the proposed local search algorithm had a positive effect on improving the efficiency of many-objective search-based test case generation; furthermore, it did not cause the search performance of the algorithm to decline significantly.

Table 9. Comparison of multi-criteria coverage between AVM-DynaMOSA and SA-DynaMOSA

Project Name	Classes	Multi-Criteria Coverage		Increase-rate	Statistics	
		AVM-DYNAMOSA	SA-DYNAMOSA		SA-DYNAMOSA versus AVM-DYNAMOSA	
					%>0.50	%<-0.50
templateit	1	72.22%	74.31%	2.88%	100.00%	
gfarcegestionfa	2	77.57%	79.76%	2.82%	50.00%	
imsmart	2	74.99%	87.11%	16.16%	50.00%	
jdbacl	1	96.47%	99.17%	2.80%	100.00%	
beanbin	1	69.45%	84.12%	21.13%	100.00%	
jsecurity	5	88.35%	88.84%	0.56%	20.00%	
byuic	2	41.04%	41.27%	0.56%	50.00%	
jni-inchi	1	83.86%	81.30%	-3.06%		100.00%
apbsmem	7	90.01%	92.98%	3.30%	28.57%	
diffi	2	89.75%	91.35%	1.79%	50.00%	
glengineer	6	77.84%	85.41%	9.72%	83.33%	
lotus	1	93.47%	92.05%	-1.52%		100.00%
resources4j	1	82.01%	89.15%	8.71%	100.00%	
jhandballmoves	3	98.28%	99.70%	1.45%	33.33%	
hft-bomberman	2	94.04%	93.35%	-0.74%		50.00%
dom4j	5	83.52%	82.24%	-1.53%		40.00%
openjms	2	76.37%	75.49%	-1.15%		50.00%
lhamacaw	5	67.34%	68.71%	2.04%	20.00%	
battlecry	2	86.77%	85.46%	-1.51%		50.00%
openhre	3	74.34%	83.92%	12.88%	33.33%	
twfbplayer	2	91.07%	93.19%	2.33%	50.00%	
wheelwebtool	7	60.73%	58.65%	-3.43%		57.14%
javathena	1	66.30%	69.14%	4.28%	100.00%	
xbus	2	80.08%	80.50%	0.52%	50.00%	
jiggler	5	86.97%	91.26%	4.93%	100.00%	
dcparseargs	1	87.99%	76.68%	-12.85%		100.00%
jcvi-javacommon	6	61.28%	60.94%	-0.55%		50.00%
quickserver	1	93.28%	87.24%	-6.47%		100.00%
heal	3	78.83%	79.34%	0.65%	33.33%	
weka	3	65.74%	54.07%	-17.76%		100.00%
liferay	2	71.74%	70.92%	-1.14%		50.00%
Mean over all projects		81.89%	82.41%			
No. cases SA-DYNAMOSA significantly better than AVM-DYNAMOSA						19 (33.93%)
No. cases SA-DYNAMOSA significantly worse than AVM-DYNAMOSA						12 (21.43%)

b) Search Efficiency Comparison

The search performance results are compared between AVM-DynaMOSA and SA-DynaMOSA in Table 10. Although the coverage of testing criteria of the final test suite was important, the search efficiency of the corresponding algorithm was also crucial. Therefore, the area under the curve (AUC), a statistical value representing the search efficiency of the test case generation algorithm, was also used for comparison. An algorithm with a high search efficiency (or AUC value) is preferred because it has the potential to produce a test suite with high coverage in a short time. As indicated in Table 10, SA-DynaMOSA achieved a higher AUC value than AVM-DynaMOSA for most tested projects, with an average improvement of 0.65%. Compared with AVM-DynaMOSA, the AUC value achieved by SA-DynaMOSA was on average significantly higher in 38.24% of the tested classes. This demonstrated that the proposed local search algorithm was valid. As the efficiency of test case generation algorithm improved, test cases with high coverage were efficiently generated in the end.

C. Threats to validity

To reduce the randomness of the results generated by the improved algorithm, the experiments were repeated 10 times. The results obtained from the experiments were averaged and the mean values were selected for analysis and comparison. However, such repeated experiments only lowered but did

not entirely remove the influence of the randomness of the improved algorithm. Additionally, a proportional change of no more than 0.5% was considered a nonsignificant change or no change. Therefore, projects or classes with an increase rate of less than 0.5% were removed when the coverage and search efficiency of the algorithms were compared. Moreover, many parameters might have affected relevant results during the experiments. To ensure the validity of the experimental results, the same parameter settings used in this study were adopted across all of the algorithms under comparison; additionally, all algorithms were run in the same framework for controlled experiments. Furthermore, the data sets selected for the experiments have been widely applied in similar studies. However, the selected data sets did not contain the most recent open-source projects. With the aim of enhancing the generalizabilty of the research findings, this study evaluated both algorithms according to the three most common coverage criteria – namely line, branch, and mutation. To determine the effectiveness of the proposed algorithm on other coverage criteria, such as MC/DC, additional experiments and further analysis are required.

Table 10. Search efficiency of AVM-DynaMOSA and SA-DynaMOSA for each project

Project Name	Classes	Multi-Criteria Coverage		Increase-rate	Statistics	
		AVM-DYNAMOSA	SA-DYNAMOSA		SA-DYNAMOSA versus AVM-DYNAMOSA	
					%>0.50	%<-0.50
templateit	1	71.62%	73.68%	2.88%	100.00%	
gfarcegestionfa	2	74.61%	78.72%	5.51%	50.00%	
water-simulator	2	84.25%	86.19%	2.30%	50.00%	
imsmart	2	73.64%	76.60%	4.02%	50.00%	
jdbacl	1	95.64%	98.23%	2.70%	100.00%	
omjstate	1	90.57%	91.21%	0.70%	100.00%	
beanbin	1	68.07%	82.00%	20.46%	100.00%	
inspirento	3	85.30%	89.84%	5.32%	66.67%	
jsecurity	5	86.66%	87.10%	0.51%	20.00%	
byuic	2	39.70%	37.87%	-4.59%		50.00%
jni-inchi	1	83.11%	80.59%	-3.03%		100.00%
apbsmem	7	81.62%	85.16%	4.33%	42.86%	
xisemele	1	84.94%	82.42%	-2.96%		100.00%
corina	5	71.19%	71.92%	1.02%	80.00%	
schemaspy	3	64.16%	65.06%	1.41%	33.33%	
diffi	2	85.27%	89.56%	5.04%	50.00%	
glengineer	6	74.42%	80.48%	8.15%	83.33%	
follow	2	68.77%	67.88%	-1.30%		100.00%
lilith	3	77.05%	77.70%	0.84%	33.33%	
lotus	1	92.60%	91.27%	-1.44%		100.00%
resources4j	1	80.85%	87.18%	7.82%	100.00%	
jhandballmoves	3	95.47%	96.60%	1.19%	33.33%	
hft-bomberman	2	92.02%	90.25%	-1.93%		100.00%
dom4j	5	78.47%	79.12%	0.83%	40.00%	
openjms	2	73.12%	72.66%	-0.63%		50.00%
lhamacaw	5	61.33%	62.54%	1.97%	40.00%	
echodep	2	71.34%	68.80%	-3.57%		50.00%
battlecry	2	66.03%	76.35%	15.63%	50.00%	
fim1	2	56.27%	56.93%	1.18%	50.00%	
openhre	3	71.35%	81.85%	14.71%	100.00%	
twfbplayer	2	87.48%	89.27%	2.04%	50.00%	
wheelwebtool	7	55.11%	51.53%	-6.49%		71.43%
javathena	1	65.57%	68.11%	3.87%	100.00%	
at-robots2-j	2	79.91%	78.72%	-1.49%		50.00%
jiggler	5	71.05%	81.41%	14.58%	100.00%	
dcparseargs	1	85.52%	72.58%	-15.14%		100.00%
classviewer	2	87.90%	86.71%	-1.35%		100.00%
jcvi-javacommon	6	57.40%	56.54%	-1.50%		33.33%
quickserver	1	91.54%	86.11%	-5.94%		100.00%
heal	3	76.37%	75.78%	-0.76%		33.33%
weka	3	51.52%	49.15%	-4.60%		66.67%
liferay	2	66.87%	64.74%	-3.19%		50.00%
pdfsam	1	81.70%	83.29%	1.95%	100.00%	
firebird	3	85.00%	83.00%	-2.35%		66.67%
Mean over all projects		78.80%	79.45%			
No. cases SA-DYNAMOSA significantly better than AVM-DYNAMOSA						26 (38.24%)
No. cases SA-DYNAMOSA significantly worse than AVM-DYNAMOSA						18 (26.47%)

V. Conclusion

Test case generation technology serves as a critical basis for software testing. How one can obtain test cases accurately with high coverage in a limited time is always a challenge in the field of testing. The search-based test case generation technique has become a heated topic in this field. DynaMOSA in EvoSuite is the most advanced and effective search-based test case generation algorithm at present. However, DynaMOSA uses the AVM for local search, which follows a greedy strategy that considers each input variable of an optimization function independently and attempts to optimize it. Problems with this kind of search are that it can easily become stuck in a local optimal solution and its search capability becomes inadequate in the late stage. Such constraints may lead to a dramatic drop in search performance. To solve these problems, this study incorporated an adaptive local search strategy with simulated annealing. Compared with the AVM, our proposed algorithm had superior neighborhood searchability for candidate solutions during test generation. The proposed algorithm was experimentally validated on selected open-source data sets from SF110. The results indicated that, compared with AVM-DynaMOSA, the coverage achieved by our approach on average was significantly higher in 20.59% of the tested classes for line, 20.59% for branch, 25.00% for mutation, and 33.93% for multicriteria. The mean coverage for line, branch, mutation, and multi-criteria of SA-DynaMOSA increased by 0.55%, 0.69%, 0.94%, and 0.52%, respectively, compared with AVM-DynaMOSA. In terms of the algorithms' search efficiency, SA-DynaMOSA also outperformed AVM-DynaMOSA and produced a higher AUC value.

References

[1] M. H. Chen, M. R. Lyu, and W. E. Wong, "An empirical study of the correlation between code coverage and reliability estimation," Proceedings of the 3rd International Software Metrics Symposium, pp. 133–141, March 1996.

[2] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, and W. E. Wong, "Establishing structural testing criteria for Java Bytecode," Software: Practice and Experience, vol. 36, no. 14, pp. 1513–1541, November 2006.

[3] C. H. Lee and C. Y. Huang, "Applying cluster-based approach to improve the effectiveness of test suite reduction," International Journal of Performability Engineering, vol. 18, no. 1, pp. 1–10, 2022.

[4] J. P. Galeotti, G. Fraser, and A. Arcuri, "Extending a search-based test generator with adaptive dynamic symbolic execution," Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014, pp. 421–424, 2014.

[5] 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, 2014.

[6] H. R. Lourenço, O. C. Martin, and T. Stützle, "Iterated local search: Framework and applications," International Series in Operations Research & Management Science, pp. 363–397, 2010.

[7] K. Amine, "Insights into simulated annealing." In Handbook of Research on Modeling, Analysis, and Application of Nature-Inspired Metaheuristic Algorithms, IGI Global, pp. 121-139, 2018.

[8] R. A. Khanum, M. A. Jan, W. K. Mashwani, N. M. Tairan, H. U. Khan, and H. Shah, "On the hybridization of global and local search methods," Journal of Intelligent & Fuzzy Systems, vol. 35, no. 3, pp. 3451–3464, 2018.

[9] D. Liu and Y. Yang, "Particle Swarm Algorithm based on Chaos Local Search and Its Application." Computer Technology and Development, 2021.

[10] Y. Zhou, L. Kong, L. Yan, S. Liu, and J. Hong, "A multiobjective memetic algorithm for multiobjective unconstrained binary quadratic programming problem," Lecture Notes in Computer Science, pp. 23–33, 2021.

[11] H. Sharifipour, M. Shakeri, and H. Haghighi, "Structural Test Data Generation using a memetic ant colony optimization based on Evolution Strategies," Swarm and Evolutionary Computation, vol. 40, pp. 76–91, 2018.

[12] H. C. Wang, "A hybrid genetic algorithm for automatic test data generation." Master's thesis, Sun Yat-sen University (2006).

[13] K. Liaskos, and M. Roper, "Hybridizing evolutionary testing with artificial immune systems and local search," In 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, pp. 211-220, 2008.

[14] 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, 2015.

[15] M. Esnaashari and A. H. Damia, "Automation of software test data generation using genetic algorithm and reinforcement learning," Expert Systems with Applications, vol. 183, p. 115446, 2021.

[16] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software." In Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp. 416-419. 2011.

[17] C. V. Ramamoorthy, S. Ho, and W. T. Chen, "On the automated generation of program test data." IEEE Transactions on software engineering, vol. 4, pp. 293-300, 1976.

[18] Cadar, Cristian, and Koushik Sen. "Symbolic execution for software testing: three decades later." Communications of the ACM 56, no. 2 (2013): 82-90.

[19] T. Weber, S. Conchon, D. Déharbe, M. Heizmann, A. Niemetz, and G. Reger, "The SMT competition 2015–2018," Journal on Satisfiability, Boolean Modeling and Computation, vol. 11, no. 1, pp. 221–259, 2019.

[20] L. Li, "Research and implementation of software automatic test." In IOP Conference Series: Earth and Environmental Science, vol. 69, no. 1, p. 012159. IOP Publishing, 2017.

[21] 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.

[22] R. Ren, "Research on Automatic Generation of Test Cases Based on Symbolic Execution." Master's thesis, Xi'an University of Technology, 2018.

[23] M. Irlbeck, "Deconstructing dynamic symbolic execution." Dependable Software Systems Engineering, vol. 40, p. 26, 2015.

[24] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," ACM Computing Surveys, vol. 51, no. 3, pp. 1–39, 2019.

[25] P. Braione, G. Denaro, and M. Pezzè, "JBSE: A symbolic executor for java programs with complex heap inputs," Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 1018–1022, 2016.

[26] P. Braione, G. Denaro, A. Mattavelli, and M. Pezzè. "SUSHI: a test generator for programs with complex structured inputs." In 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), IEEE, pp. 21-24, 2018.

[27] M. Modonato, "Combining Dynamic Symbolic Execution, Machine Learning and Search-Based Testing to Automatically Generate Test Cases for Classes." arXiv preprint arXiv:2005.09317, 2020.

[28] G. Yang, A. Filieri, M. Borges, D. Clun, and J. Wen, "Advances in symbolic execution," Advances in Computers, pp. 225–287, 2019.

[29] 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, 2018.

[30] S. Vogl, S. Schweikl, G. Fraser, A. Arcuri, J. Campos, and A. Panichella, "EVOSUITE at the SBST 2021 Tool Competition," In 2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST), pp. 28-29, 2021.

[31] B. Korel, "Automated Test Data Generation for programs with procedures," Proceedings of the 1996 international symposium on Software testing and analysis - ISSTA '96, vol. 21, no. 3, pp. 209–215, 1996.