Adapting Swarm Intelligence Based Methods for Test Data Generation

M. M. Dejam Shahabi[1]; S. E. Beheshtian[2]; P. Badiei[3]; R. Akbari[4,*]; S. M. R. Moosavi[5]

1- Department of Computer Engineering and Information Technology, Shiraz University of Technology, Shiraz, Iran, Email: m.shahabi@sutech.ac.ir
2- Department of Computer Engineering and Information Technology, Shiraz University of Technology, Shiraz, Iran, Email: beheshtian@sutech.ac.ir
3- Department of Computer Engineering and Information Technology, Shiraz University of Technology, Shiraz, Iran, Email: oossparsa@gmail.com
4- Department of Computer Engineering and Information Technology, Shiraz University of Technology, Shiraz, Iran, Email: akbari@sutech.ac.ir
*Corresponding author
5- Department of Computer Science, Engineering and Information Technology, Shiraz University, Shiraz, Iran, Email: smmosavi@shirazu.ac.ir

Received: 2020-08-15
Revised: 2020-12-08
Accepted: 2021-02-07

Abstract To achieve high-quality software, different tasks such as testing should be performed. Testing is known as a complex and time-consuming task. Efficient test suite generation (TSG) methods are required to suggest the best data for test designers to obtain better coverage in terms of testing criteria. In recent years, researchers to generate test data in time-efficient ways have presented different types of methods. Evolutionary and swarm-based methods are among them. This work is aimed to study the applicability of swarm-based methods for efficient test data generation in EvoSuite. The Firefly Algorithm (FA), Particle Swarm Optimization (PSO), Teaching Learning Based Optimization (TLBO), and Imperialist Competitive Algorithm (ICA) are used here. These methods are added to the EvoSuite. The methods are adapted to work in a discrete search space of test data generation problem. Also, a movement pattern is presented for generating new solutions. The performances of the presented methods are compared over 103 java classes with two built-in genetic-based methods in EvoSuite. The results show that swarm-based methods are successful in solving this problem and competitive results are obtained in comparison with the evolutionary methods.

Keywords: Test data generation, Firefly Algorithm, Particle Swarm Optimization, Teaching Learning Based Optimization, Imperialist Competitive Algorithm, EvoSuite

1. Introduction

Software plays an important role in every aspect of our life. Any malfunction or lack of precision in software can directly affect our lives and result in financial loss or even death in some cases. To prevent any potential damage, software needs to be tested before released. Software testing is expensive and can cost up to 50% of the software's development cost. As testing and quality assurance is a vital process for software, it cannot be overlooked, so we need ways to reduce the cost of it and yet improve the precision [1], [2]. Automated methods have been used to enhance this process. Automated methods, such as search-based software testing (SBST) have been greatly noticed in recent years [3]. The search-based methods formulate the test problem into a search problem and search the problem space to find the proper data to carry out the tests [4].

The importance of testing encouraged researchers to develop different types of methods and suits such as EvoSuite [5] to cope with its difficulties. The objective of test data generation is to have a test suite that maximizes coverage criteria [6]. Finding the input test data that best serves our purpose of testing a software's behaviour is a challenging task. Search-based methods are promising methods in test data generation. Search based methods solve an optimization problem using search algorithms to generate test cases and proper input data for them [6].

In this paper, four modified swarm intelligence algorithms (i.e. TLBO, PSO, Firefly, ICA) are used for automatic test suite generation. Previous studies showed that swarm intelligence methods have good performance in solving hard engineering problems. It seems that the swarm-based methods have the ability to show good performance for test suite generation in comparison to evolutionary methods such as GA. The idea of this work is to study the capabilities of swarm-based algorithms to search in discrete search spaces of TSG problem. The proposed methods are implemented in EvoSuite [5]. We select the EvoSuite because this tool showed better performance at most of the benchmarks in comparison with the other TSG tools.

The results obtained by the proposed methods are compared with the results of two built-in genetic algorithms (i.e. MonotonicGA, and StandardGA) in EvoSuite. The results show that the swarm-based methods have competitive performance in comparison with the genetic-based methods. The dataset used here is SF110 from *SourceForge* and eight coverage criteria are considered.

The main contributions of this work are:

- Adapting swarm-based method to search in a discrete search space of TSG problem in EvoSuite.
- Studying the behaviour of these algorithms for test data generation
- Presenting a new way to update solutions that are generated by individuals in a swarm.

The remaining of the paper is organized as follows: Related work is presented in the next section. Section 3 presents the swarm intelligence based methods for generating test data. The performances of the proposed methods in test data generation are reported in Section 4. Finally, Section 5 concludes this work and suggests some proposals as future works.

2. Related Work

Automatic test data generation methods can be categorized into three types: random methods, dynamic symbolic execution, and search-based methods. In this work, we focus on search-based methods. In recent years, different types of search-based methods have been presented for test data generation in literature. Search based methods use search algorithms for generating test data. Most of the works in this field are based on GA and its variants. However, some works employ swarm-based methods for test suite generation. It seems that GA is a good choice because it has shown good performance in finding optimum solutions in discrete search spaces.

Bruce et al. presented a tool called Dorylus for TSG [7]. Dorylus uses Ant Colony Optimization (ACO) algorithm. They focused on branch distance and Levenshtein distance as the fitness function and compared their method with the implemented methods in EvoSuite. The results showed Dorylus obtains better coverage in comparison with the built-in methods in Evosuite.

Rojas et al. extended EvoSuite to enable them to combine multiple criteria [8]. They stated that generating test cases for a single criterion may not be a good indicator for evaluating an algorithm. Therefore, they combined nine criteria and studied the performance of EvoSuite under this combination. The results showed that although combining the criteria decreases the constituent coverage criteria slightly, but a significant growth up is seen in the test suite.

Bruce et al. have presented a Tiered Ant Colony Optimization (TACO) for generating unit tests [9]. The proposed method has three tiers where the first and second tiers are used for goal prioritization, and test program synthesis respectively. The third tier generates the test data for the program. They compared TACO with the Randoop and EvoSuite tools. The experimental results showed that EvoSuite has better performance than TACO and Randoop.

Jatana and Suri presented an Improved Crow Search Algorithm (ICSA) for TSG [10]. The ICSA improved the search capability of the CSA by utilizing Cauchy random numbers. They compared the proposed method with some meta-heuristics. The results showed that the ICSA generates a better test suite in comparison with the other methods.

The performance of the Evolutionary Algorithms (EA) for test suite generation has been studied by Campos et al. in [11]. The authors stated that GA is the first choice in the software engineering domain. Hence, they focused on EA for test suite generation. They evaluated six versions of EA. The results showed that using a test archive helps the algorithms to obtain better coverage in comparison with the random test.

Rojas et al. presented a whole test suite approach that has been used in EvoSuite [12]. This method tries to generate test suites as a whole and optimize them along with iterations rather than the traditional method that would target coverage goals individually by generating separate test cases for them. This method proves to work much better, achieving up to 18 times the coverage than the traditional way. In addition to higher coverage, this approach also generates smaller test suites due to a reduction in search redundancy and overlapping in goal coverage of the test cases.

Shahabi et al. extended the EvoSuite by implementing Particle Swarm Optimization (PSO) and Teaching Learning Based Optimization (TLBO) [13], [14]. They studied the performance of these swarm-based methods for TSG in comparison with the GA-based methods of EvoSuite. Their study showed that these swarm-based methods generate competitive results in comparison with the GA methods. However, the GA methods surpass PSO and TLBO in most of the coverage criteria.

The applicability of unit test generation tools on industrial projects has been studied by Almasi et al. [15]. They studied the performance of EvoSuite and Randoop for finding faults in life insurance software. The results showed that Evosuite has better performance than Randoop.

Shamshiri et al. studied the behaviour of evolutionary search against the random search for TSG in object-oriented software [16]. They applied EvoSuite for unit test generation over 1000 classes randomly selected from SF100 projects. Their study showed that although evolutionary searches are better for covering complex branches, the random search may be enough for obtaining a good coverage level in most of the classes.

Oliveira et al. studied the effect of features of object-oriented classes on the effectiveness of the automated TSG tools [17]. They found that some object-oriented metrics such as coupling and number of methods make software to be hard to test by different techniques.

The performance of search-based unit test generation methods may depend on their ability in controlling diversity when exploring the search space. Albunian studied the effect of population diversity in unit test generation [18]. For this purpose, he examined different ways of diversity control in GA. His study showed that increasing the population diversity increases the length of the individual rather than improving the coverage.

In another study, Gay showed that combining coverage criteria helps the search-based test suite generation methods to generate test suites that are more effective for detecting faults in the CUTs [19]. He mentioned that search-based methods show good performance on the coverage criteria. But, they are ineffective in finding faults.

As another work for empowering the search-based test generation methods, Olsthoorn et al. used model seeding to add some information to the methods [20]. They used the proposed method for test generation on the Gson using EvoSuite.

In automating TSG, usually testing tools are used to save time and decrease the cost of testing. In recent years, some testing tools have been developed for this purpose. The effectiveness of automated testing tools such as EvoSuite against manual testing has been studied by Serra et al. [21]. They compared manual testing against EvoSuite, Randoop, and JTExpert in terms of mutation score, code coverage, and fault detection. The experiments showed that automatic test generation tools improve code coverage and mutation score in comparison with manual testing. However, improvement in fault detection is not significant.

Rudžionienė et al. presented a method that uses multiple search targets for test generation [22]. They presented their method as a tool and compared it with the other test generation tools that use search-based methods.

Almulla and Gay studied the effect of Adaptive Fitness Function Selection (AFSS) on the diversity of the generated test suites [23]. They found that using AFFS helps the automated TSG tools to generate more diverse test suites in comparison with the test suites that are generated using static fitness functions.

Using the information in the source code can help search-based test suite generation methods to produce better test cases [24]. As an example, Evers et al. used the commonality score to measure the distance between the execution path of a test case and the common/uncommon execution pattern observed during the execution of the software. Using commonality as an objective in EvoSuite, they found that the generated test cases have better commonality score.

3. Problem formulation

The problem considered in this work is known as test data generation where the objective is to generate a test suite to optimize some criteria. A test suite is a set of test cases. The test cases are used by software testers to test the class under test (CUT). Each test case is composed of different parts. The main parts are inputs, the sequence of statements, and execution conditions.

When we solve test data generation using swarm-based methods, we should model it as a search problem. For this purpose, an objective or fitness function should be defined. By defining the fitness function, a swarm-based method tries to optimize the fitness function by searching the search space and finding the optimal position in this space. Generally, the search problem is modeled as:

$$\max\ f_{test\_criterion}(Suite) \tag{1}$$

In this model, $test\_criterion$ is the coverage criteria that should be maximized. Hence, each algorithm tries to find a test suite that optimizes the selected criterion. Many criteria in software testing can be used as a fitness function. Here, we use eight criteria to assess the performance of the proposed algorithms: 1) line coverage, 2) branch coverage, 3) exception coverage, 4) weak mutation coverage, 5) output coverage, 6) method coverage, 7) method no-exception coverage, and 8) c-branch coverage.

4. The Proposed Methods

In recent years, researchers have proposed a large class of swarm-based optimization algorithms. These methods have shown satisfactory results in solving engineering problems. Usually, these methods have been used when computation time is important. These methods have the ability to provide near-optimal (in some cases even optimal) solutions in an acceptable time. Based on this fact, in this work, several swarm intelligence algorithms are adopted and implemented into the EvoSuite and their performances are evaluated. These methods are selected as representatives of swarm-based optimization techniques. We have tried to select methods with different optimization behaviours. The selected methods are different in terms of movement patterns, social/psychological impacts, neighbourhood, leader selection, and other important factors in optimization methods. These methods are based on:

1) Particle Swarm Optimization (PSO),
2) Teaching Learning Based Optimization (TLBO),
3) Firefly Algorithm (FA),
4) Imperialist Competitive Algorithm (ICA).

All the algorithms in their standard forms have been designed for optimizing continuous problems. Also, a strategy is proposed for the movement operator which makes it applicable to a discrete search space of the test data generation problem.

The overall structure of the proposed methods is presented in Fig. 1. Each method starts by initializing the swarm using the built-in module of EvoSuite. To use a certain algorithm for the optimization, an initial population is randomly generated using the provided methods in EvoSuite [25], [26]. This phase is similar in all the proposed algorithms. After that, the update phase (the main module) of the proposed method starts. For this module, four suggested algorithms (i.e. TLBO, PSO, Firefly, and ICA) are used separately.

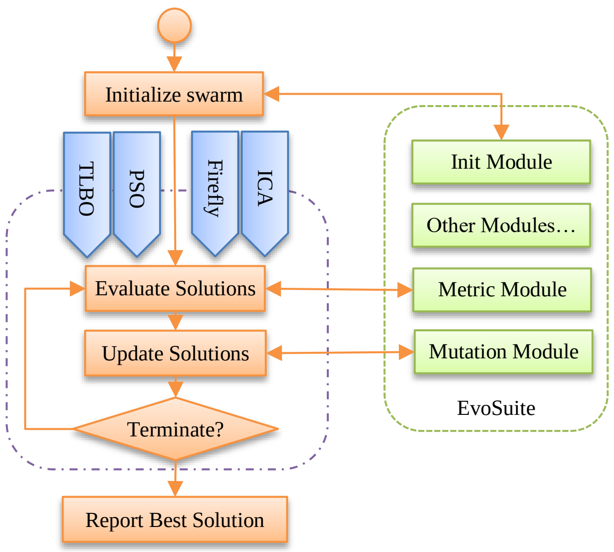

Fig. 1. The overall structure of the proposed methods.

The method updates the solutions cycle by cycle until the termination condition is met. To update the current solution, an optimization algorithm needs to compute the quality of its individuals based on a fitness function. The EvoSuite based on the selected metric calculates the fitness of an individual in the swarm. More precisely, the built-in metrics of EvoSuite such as code coverage is used as the fitness function. After termination, the best result is reported by the algorithm. The details of the swarm-based methods are described in the following sub-sections. However, we need to represent the movement operator and solution representation before the description of the proposed methods.

It should be noted that EvoSuite has many classes and has its complexities. However, for the sake of simplicity, the functionalities of EvoSuite which are directly used by the proposed method are shown by dummy names in the following figure. It is assumed that *Init* module initializes the individual, *Metric* module computes the fitness of individuals, and *Mutation* module is used to mutate individuals.

4.1. Discrete and Continues Space Operators

In every evolutionary algorithm, the evolutionary mechanism is done by several operators. The two mechanisms responsible for altering the individuals are called crossover and mutation. In discrete search spaces like what a classic genetic algorithm works in, individuals get crossed over with each other to generate offspring. Moreover, usually, a part of the population is mutated as well. On contrary, for algorithms in the swarm intelligence paradigm that works in continuous search spaces, the new generation is not born in every iteration but it is the same population that gains more experience in every iteration. This experience is improved by moving the individuals in the search space according to a movement strategy to find the best area (i.e. a point with the best fitness value) in the search space. The movement strategies are usually designed to work with continuous decision variables.

Most of the swarm intelligence based methods in their basic form have been proposed to work in continuous search spaces. The test data generation problem has a discrete search space. Hence, we need to adopt the proposed methods in such a way to work with this discrete problem. For this purpose, in this paper, a movement strategy is proposed that enables the swarm intelligence algorithms, which in nature are of continuous type, to work in discrete search spaces. As in the classic swarm intelligence algorithm, this strategy is based on the main factor of the swarm intelligence paradigm which is the communication between individuals telling each other the best point they have found so far. The details of the movement strategy are given in Section 4.3.

4.2. Solution Representation

Solution representation plays an important role in the success of an optimization method. Hence, we need to pay more attention to this. The representation proposed by the EvoSuite designers has a good structure that appropriately shows the test suits and test cases. Hence, in this work, the authors prefer to use the representation which is the same as what has been used in the EvoSuite tool [25]. This representation has been basically proposed for GA-based methods. In the genetic coding model that has been used in these algorithms, every member is represented as a chromosome, and the attributes of each individual are determined by its genes. In terms of test data generation, test cases and test suites are both represented as chromosomes. At the test suite level, a chromosome's genes are corresponding to test cases. At the test case level, genes are statements in a test case. Statements are of various types: Method calls, primitive statements (variable declaration), constructor statements (that create classes), field statements (accessing public members of a class), and assignment statements.

4.3. Proposed Movement Operator

The operator requires two inputs, a source individual and a destination. A movement rate is also used in the movement that is set as the algorithm's parameter in the configurations. The operator makes a list of each individual's attributes (i.e. test cases or statements) and determines attributes exclusive to the destination. As this movement works on both test suite and test case levels; on the test suite level, to determine candidates, test cases are prioritized based on their coverage. Test cases with exclusive goal coverage have higher priority. However, on the test case level, there is no specific prioritization for the statements except their exclusive existence in the destination test case. Then according to the provided rate, a number of the attributes are added to the source from the destination. This process leaves the destination as it is and only changes the source by adding it to its list. It should be noted that as this movement operator works in discrete spaces, the movement is not in vector form unlike the classical algorithms of the swarm intelligence paradigm.

4.4. TLBO Algorithm

TLBO is the first swarm intelligence technique that is used in the update module of the proposed method given in Fig. 1. TLBO [27] is a successful algorithm in solving NP-Hard problems which has been proposed by Rao. The proposed TLBO method uses the representation given in Section 4.2 and updates solutions by incorporating movement operators into the update phases of the classic TLBO algorithm.

In general, this algorithm simulates the teaching and learning mechanisms in a class. In other words, a social and psychological phenomenon in a class between teacher and students is considered. The swarm (or population) in this algorithm consists of students and a teacher which is the best student in every iteration. Students both learn from their teacher and tutor each other. A student with a higher score has a higher coverage percentage in terms of software testing. The procedure of the proposed TLBO algorithm, which is based on classic TLBO, with the proposed movement operator, is described in Fig. 2. Apart from the initialization phase, the main body of the TLBO has two phases that are known as teaching and learning.

A) Initialization: The main task in the initialization phase is assigning the first values for the individuals of the swarm. This could be done randomly. The proposed algorithm receives a randomly generated population as input. For this purpose, the random method of EvoSuite is used. Each student of the class (i.e. population) represents a test case or a test suite depending on the level of optimization the algorithm is working at. Besides defining the first values of individuals, the parameters of the algorithm such as termination condition should be set at this phase.

```
Initialize the position of individuals using Evosuite Init module
    Set number of students, termination condition
While (termination condition not met)
    Calculate the mean of decision variables
    Identify the best solution as the teacher
    Identify the movement percentage based on the average and a random number
    Modify solution based on the best solution
    $X_{new} = X_{old} + r(X_{teacher} - (T_F)Mean)$
    If the new solution better than existing
        Accept the solution
        Mutate the solution
    Else
        Reject the solution
    End If
    Select two solutions randomly $X_i$ and $X_j$
    If $X_i$ better than $X_j$
        $X_{new} = X_{old} + r(X_i - X_j)$
    Else
        $X_{new} = X_{old} + r(X_j - X_i)$
    End If
    If the new solution better than existing
        Accept the solution
        Mutate the solution
    Else
        Reject the solution
    End If
End While
Return the best solution
```

Fig. 2. Pseudocode of the proposed TLBO algorithm

B) Teaching phase: By initializing, the TLBO updates individuals (here called students) through teaching and learning phases. The teaching and learning phases iterate cycle by cycle until the termination condition is met. At each cycle, all the students are evaluated using the *Metric* module of the EvoSuite, and the one with the highest score (i.e. coverage percentage) is chosen as the teacher. Then all the other students move toward the teacher to improve their fitness using the proposed movement pattern:

$$X_{new} = X_{old} + r(X_{teacher} - (T_F)Mean) \qquad (2)$$

where, X_{old} , X_{new} are the current and next positions of a student respectively. $X_{teacher}$ is the position of the teacher, r is a random parameter that brings stochasticity to the algorithm, and $(T_F)Mean$ shows the mean positions of the students. This means that all the students share their knowledge for updating positions with each other. The new solution is accepted and mutated if a better position in the search space has been achieved by the update formula. The solution is mutated by the *Mutation* module of EvoSuite. Otherwise, the new solution is discarded. This movement pattern models a greedy approach for updating positions.

C) Learning phase: The update phase continues with the learning phase. In this phase students tutor each other, meaning that each student chooses a classmate randomly and compares its score with it. In case that the chosen classmate X_i (by student X_j) has a higher score, the student moves toward it using:

$$X_{new} = X_{old} + r\left(X_i - X_j\right) \qquad (3)$$

Otherwise, the student moves away from the classmate using:

$$X_{new} = X_{old} + r\left(X_j - X_i\right) \qquad (4)$$

In the learning phase, the greedy approach is used to accept or reject the new solution. The solution is updated if it obtains better fitness.

D) Stopping conditions: As this algorithm, works iteratively for its optimization purpose, certain conditions stop the algorithm. At the end of every iteration, the whole population is evaluated and if a student scores a certain coverage percentage (set as a parameter to the algorithm), the algorithm stops and returns that student as the best-found answer. In addition to that, a certain number of iterations or a limited time for running can also be set as stopping conditions. If none of the conditions are met, the algorithm continues from step B.

4.5. PSO Algorithm

As the second method to update solutions, PSO is used. It can be said that PSO is a classic and well-known swarm-based method that has been extensively used by researchers to solve engineering problems. PSO 0like every swarm intelligence algorithm uses two search mechanisms to balance exploration and exploitation. At first and in the initial iterations, PSO mostly explores the search space to cover more ground. Throughout the search with a decrease in exploration, PSO focuses on the superior areas found in the search space and exploits them [29].

PSO works according to the following strategy shown in Fig. 3. After initialization, the method improves solutions by two phases: 1) determining personal and global best positions, 2) updating positions. By terminating the algorithm, the best solution is reported as output.

A) Initialization: The initial population is generated by randomly generating particles (i.e. test suites and test cases) and spreading them throughout the search space. Similar to other methods the provided random initialization method of EvoSuite is used for this purpose. After initialization, the personal best of each particle should be determined. These initial positions are considered as the particles' personal best. Also, the gbest is selected as the best initial position. Like the TLBO, the input parameters are set at this phase.

B) Determining personal and global best positions: After initialization, in this step, the fitness value for every particle is calculated and the particle with the best fitness value among the population is selected as the global best. The fitness value of the current position is compared to the best personal position's fitness value. If the current position has a better fitness (i.e. higher coverage) it is set as the personal best position.

C) Update (particle velocity and position): The first movement is a mutation done by the provided mutation method in EvoSuite which randomly adds or removes test cases in a test suite or statements in a test case. Particles have two consecutive movements. The first is toward the global best position based on the following formula which is adopted from the traditional PSO algorithm.

$$X_{t+1} = X_t + V_{gb} \tag{5}$$

$$V_{gb} = C_1(X_{gb} - X_t) \tag{6}$$

In the formula above, X_t is the current particle, X_{gb} is a copy of the global best particle and C_1 is the gb movement rate set as a parameter to the algorithm. $(X_{gb} - X_t)$ is attained by removing the mutual test cases from X_{gb}. C_1 determines the percentage of the exclusive test cases of the global best particle to be copied to the current particle. In this way, the resulting particle after the movement is closer to the global best particle in terms of its attributes (i.e. statements or test cases).

The second movement is done similarly except towards the personal best particle.

$$X_{t+1} = X_{t+1} + V_{pb} \tag{7}$$

$$V_{pb} = C_2(X_{pb} - X_{t+1}) \tag{8}$$

X_{t+1} is the resulting particle from the previous movement, X_{pb} is the personal best particle and C_2 is the pb movement rate set as a parameter to the algorithm.

The age is updated according to the number of iteration. This procedure is repeated for every particle in the population. As mentioned before the movement operator is a discrete operator which means unlike the classical PSO [28], [29] that particles move in vector forms, here there is no summation on movements and both movements are done separately.

```
Initialize the position of individuals using Evosuite Init module
    Set number of students, termination condition
While (termination condition not met)
        Identify the best solution as Global Best
        Identify the best position in each individual history as Personal Best
        Identify the movement percentage given as a parameter to the algorithm
        Modify solution based on Global and Personal Best
        $X_{t+1} = X_t + V_{gb}$
        $V_{gb} = C_1(X_{gb} - X_t)$
        $X_{t+1} = X_{t+1} + V_{pb}$
        $V_{pb} = C_2(X_{pb} - X_{t+1})$
        If the new solution better than existing
                Accept the solution
                Mutate the solution
        Else
                Reject the solution
End While
Return the best solution
```

Fig. 3. Pseudocode of the proposed PSO algorithm

D) Stopping conditions: As this process can go on forever without finding the absolute best position in the search space, there are several stopping conditions in place. Run time, the number of iterations, and a certain fitness value threshold are used for this purpose. The algorithm checks the stopping conditions and if not met, continues from step B. After termination of the algorithm, the global best particle is given as the output.

4.6. ICA Algorithm

The ICA is the third method that is used to update solutions. This algorithm is based on the classic ICA [30] with the proposed movement operator. ICA works according to the following strategy shown in Fig. 4. The imperialist competitive algorithm is inspired by the imperialistic behaviour of some dominant countries in history. Imperialism is the act of expanding a country's power beyond its borders. In this strategy, instead of directly ruling a country, the imperialist state controls it indirectly by some less obvious ways like controlling its market and economy.

The algorithm uses elements to represent the political phenomenon described earlier. The individuals are countries, which can be of two types, imperialist, and colony. A group of countries together form an empire and all the empires together form the population. Empires are consisted of imperialist and other countries as its colonies.

```
Initialize individuals using Evosuite Init module

    Set number of countries, termination condition
While (termination condition not met)
        Identify Empires based on their cost
        Identify distribute countries between empires
        Modify solution based on the emperor in each empire
        $X_{new} = X_{old} + r(X_{emperor} - X_{old})$
        **Mutate** random solution known as revolution function
        **Search** to replace the emperor with a better country based on their costs in each empire
        **Search** for the weakest empire, remove one colony from it, and add it to another empire
**End While**
**Return** the best solution
```

Fig. 4. Pseudocode of the proposed ICA algorithm

Regarding the problem representation, test suites and test cases are both represented as countries. The fitness function is the coverage percentage in a specific coverage criterion. After initialization, the ICA method updates solution through five phases: 1) choosing imperialists and forming empires, 2) update (or assimilation), 3) revolution, 4) empire competition, and 5) inter empire competition. By terminating the method, the best solution is reported as the output.

A) Initialization: Like other swarm-based algorithms, the initial swarm is a randomly generated population. This random method is the same as what is used in EvoSuite. Although, parameters of ICA are set at this phase.

B) Choosing imperialists and forming empires: A part of countries (set as a parameter) with the best fitness value (i.e. highest coverage percentage) is selected as imperialists. The rest of the countries in the world (population) are divided into different groups called empires, each empire is assigned to an imperialist and forms the colonies for the corresponding imperialist.

C) Update (assimilation): This phase optimizes the swarm using the proposed movement method. Colonies move toward their imperialist by receiving attributes from it.

The proposed movement operator gets the source colony and the imperialist as its inputs. A movement rate is defined as a parameter for this algorithm that determines what portion of the imperialist's attributes (i.e. test cases or statements) should be given to the colony to move the colony closer to the imperialist.

D) Revolution: In this phase, a mutation is done on all the population. This mutation again is the built-in mutation operator of the EvoSuite.

E) Intra empire competition: After the revolution, a colony might have reached a better point in the search space (a higher coverage percentage), in this case, the colony is selected as the new imperialist for that empire.

F) Inter empire competition: In this step, the total power for every empire is calculated and the weakest colony of the weakest empire is removed and added to another empire which itself is chosen by the Boltzmann formula (unlike the classic ICA).

$$totalPower_{empire} = coverage(IMPERIALIST) + \sum coverage(colonies\ of\ empire) \qquad (9)$$

$$boltzman\ val_{empire} = \frac{e^{-totalPower_{empire} * random}}{\sum totalPower_{empire}} \qquad (10)$$

The total power calculation for an empire takes two parameters; the imperialist power which is its coverage percentage and the summation of the power of the colonies. To choose a host for the removed colony, Boltzmann value for every empire is calculated and the empire with the highest value takes the removed colony. After this step if an empire is left without any colonies, that empire is eliminated and the imperialist is added to another empire again chosen by the Boltzmann value.

G) Stoppin-g conditions: If the algorithm runs for long enough, all the empires except the most powerful one collapse, and all the colonies are in the same empire with one imperialist. In this empire, all the colonies have the same fitness value as each other as well as the imperialist, in this case, the algorithm is stopped and the imperialist is given as the best-found answer. If otherwise, a certain stopping condition like running time, number of iterations, or a specific coverage percentage in a country will stop the algorithm and return the best (highest coverage percentage) country found, as the answer. If none of the stopping conditions are met, the algorithm continues from step C.

4.7. *Firefly Algorithm*

Firefly algorithm is another swarm intelligence paradigm algorithm that is inspired by the flashing behaviour of fireflies [31]. Firefly works according to the following strategy shown in Fig. 5. In nature, fireflies flash a chemically produced light to attract each other. In the firefly algorithm modeling, all fireflies are unisexual and any individual firefly can attract another one. The attraction between two fireflies is proportional to their brightness, the brighter one attracts the less bright firefly. In addition to the brightness, the distance between two fireflies has a negative correlation with the attraction between them. The last rule is that if there is a situation where there are no fireflies brighter than the given one, it will move toward another firefly randomly.

Regarding problem representation, both test suites and test cases are regarded as fireflies. The cost function is represented as the brightness of a firefly, the brighter it is the more coverage it has.

A) Initialization: The population with the given size as the algorithm's parameter is generated randomly via the given random method in the EvoSuite. Also, the parameters of the algorithm are set based on the determined configuration.

B) Update (movement): The proposed movement operator gets two fireflies as its inputs. Based on their brightness and distance the attractiveness between them is calculated to determine which one moves toward the other.

$$B = B_0 \,.\, e^{-\gamma \,.\, r^{\wedge}2} \approx \frac{B_0}{1+\gamma r^2} \qquad (11)$$

Y is a parameter set to the algorithm and the higher B is for a firefly the more attractive it looks to the other fireflies. r_{ij} is the distance representative between the two fireflies calculated by the formula below.

$$r_{ij} = \left(brightness_i - brightness_j\right)^2 \qquad (12)$$

According to the movement rate that is defined as a parameter for this algorithm, a portion of the destination firefly's attributes (i.e. test cases or statements) is given to the moving firefly. This comparison of attractiveness and movement is done for every pair of fireflies in the population.

C) Stopping conditions: Like the other algorithms, there are certain rules to stop the iterative procedure of the algorithm.

```
**Initialize** the individuals using Evosuite *Init* module
    Set number of fireflies, termination condition
**While** (termination condition not met)
    **For all fireflies**
        **For all other fireflies**
            **If another firefly is better than this one**
                $B = B_0 * e^{-r^2}$
                $X_{newThis} = X_{oldThis} + rB(X_{other} - X_{this})$
**End While**
**Return** the best solution
```

Fig. 5. Pseudocode of the proposed Firefly

The time for running, certain coverage (brightness) in a firefly, or the number of iterations. If none of the conditions are met at this step, the algorithm continues from step B.

5. Performance Study

In this section, the applicability and performance of the proposed swarm intelligence based methods are presented. First, the setting of the experiments and configuration of the methods are presented, and then the experimental results are reported. Finally, the overall performance sub-section gives the obtained results generally.

5.1. *Experimental Setup*

The proposed methods have been integrated into EvoSuite as a platform to test their performance. In addition to the four algorithms introduced in this paper, 2 algorithms of the genetic family (i.e. Standard GA and Monotonic GA) have also been studied and a comparison has been made among them. The proposed methods use built-in functionalities of EvoSuite for mutation, solution representation, objective function computation, and initialization of individuals.

The data set used here is downloaded from SourceForge, an online code repository. 103 classes are randomly selected from the SF110 data set. The selected classes have been coded in Java. The selected classes have different features (e. g. number of methods, line of codes, number of branches, and number of exceptions).

All the classes have been studied with all 6 algorithms. Each algorithm works in eight criteria on separate procedures. As there are about 5000 rows of outputs, the performance of each algorithm in each of the criteria has been summarized in the tables presented in this section. The results presented are attained with the following configurations for the algorithms are given in Table I. Some of the parameters are used in common and some of them are exclusive to an algorithm. The size of the swarm is set at 50, and methods terminate after a predefined number of iterations.

Each algorithm has been evaluated 10 times and the average results are computed and presented in Section 5.2. For a fair comparison, the total goals and achieved goals are used in the experiments. At each run, EvoSuite determines total goals and reports the number of goals achieved by an algorithm. The average performance is computed by dividing achieved goals by the total goals.

Table I. Configurations of the algorithms

TLBO	
Population Size	50
Teaching Factor	[random*2]
PSO	
Population Size	50
Personal Best Movement rate	15%
Global Best Movement rate	40%
ICA	
Population Size	50
Empire Population	5
Colonies to Empire Movement rate	20%
Firefly	
Population Size	50
B_0 (base attraction)	50%
Y	1

5.2. *Performance Results*

The proposed methods are compared with the standard GA and Monotonic GA based on different performance metrics to show their capabilities precisely. The standard GA and Monotonic GA are built-in methods of EvoSuite.

The performance of the PSO for test data generation in terms of four metrics has been studied by the authors in [13]. In this work, twenty classes have been selected randomly for comparison. The results have been reported for each class separately. Also, the results obtained by PSO have been compared by Standard GA and Monotonic GA. The results showed that the examined methods have different performances in some of the selected classes.

Besides the PSO method, the authors have designed another method based on TLBO for test data generation in EvoSuite [29]. In this work, the experiments have been extended to 50 classes and the methods have been compared in terms of four metrics. Similar to [13], the results have been presented in detail. We have seen the same behaviour from the test data generation methods in [13] and [14]. Here, the number of classes under test is extended to show the performance of the methods in a more accurate way. In addition, more performance metrics are considered. In this work, we assume that all the metrics have the same importance for a test designer. However, it may be possible for one to prioritize the coverage metrics.

The first metric is the line coverage. This metric has been widely used for evaluating the performance of the TSG methods. Line coverage shows the percentage of lines of codes (statements) that have been run by the test cases. The details of this metric and the others have been presented in [5]. The results of the proposed swarm-based methods against each other and GA based methods are presented in Table II. The Monotonic GA and Standard GA. obtain the best result for this metric.

Branch coverage is the second criterion used for comparison. This metric shows the percentage of decision points their true/false branches have evaluated by test cases. As shown in Table II, the standard GA method outperforms the others.

The exception coverage is another metric selected for comparison. This criterion is considered as the percentage of exceptions in the code that have been covered by the test cases. Monotonic GA and ICA obtain the best results. The standard GA and Firefly have the same performance.

The fourth criterion is mutation coverage. The best results are obtained by the GA-based methods. The swarm-based methods obtain nearly the same results.

The fifth metric is the output coverage. Standard GA and ICA obtain the best results for output coverage. The PSO and Firefly placed at the last ranks.

Method coverage is the sixth metric, which is used for comparison. Method coverage shows the percentage of methods that have been called by test cases. The ICA method shows superiority in terms of this metric. The PSO and TLBO obtain the second and third ranks respectively.

The no-exception coverage is another coverage criterion that is used here for comparison. The standard GA is placed at the first rank for no exception coverage while the second rank is obtained by PSO method. TLBO and Monotonic GA have competitive performance.

The last criterion used for comparison is the c-branch. Like the other metrics, the best results for c-branch obtained by the GA based methods. Next, the best result is obtained by ICA.

Table II. Comparison of the methods based on coverage criteria

		Method					
		Standard GA	Monotonic GA	Firefly	PSO	ICA	TLBO
Coverage Criteria	Line	57.80	57.94	54.44	54.25	55.57	55.43
	Branch	60.34	60.14	56.37	56.09	57.53	57.27
	Exception	97.99	98.63	97.99	97.69	98.59	97.86
	Mutation	62.55	61.83	60.10	60.38	60.91	60.56
	Output	54.91	53.86	52.12	51.41	53.87	53.03
	Method	84.90	84.40	85.26	86.80	87.18	86.24
	Exception	83.30	82.42	81.96	82.63	82.18	82.47
	C-branch	58.81	59.25	55.94	54.84	56.52	56.04
	Average	70.07	69.81	68.02	68.01	69.04	68.61

5.3. Overall performance

The lexicographic ordering method is used here to study the overall performance of the methods. In this method, the ranks of a method in each of the experiments are summed. Number 1 shows the first rank, the second rank is considered as 2, etc. The method with the lowest value is considered as the best method. The average score is the mean of ranks for the corresponding methods. The results of the lexicographic ordering are shown in Table III. From the table, standard GA and monotonic GA are at the first and second rank respectively. While ICA ranked third, the ranking for other swarm-based methods is as follows.

Table III. The results of the lexicographic ordering.

#Rank	Average Score	Algorithm
1	2.0	Standard GA
2	2.375	Monotonic GA
3	2.75	ICA
4	3.875	TLBO
5	4.75	PSO
6	4.875	Firefly

The average of the coverage based on all metrics is shown in Table IV. As can be seen from the table, the same result as the lexicographic ordering is obtained for the best algorithm (i. e. Standard GA). But, the worst algorithm based on average performance is PSO. However, the difference between the best and worst methods is about 2.06%.

Table IV. The average coverage based on all metrics.

Algorithm	Average Coverage
Standard GA	70.07%
Monotonic GA	69.81%
ICA	69.04%
TLBO	68.61%
PSO	68.01%
Firefly	68.02%

This means that swarm intelligence based methods have the potential to produce better performance in further studies by considering more efficient position updating methods, neighbourhood topologies, flying patterns, and many other contributions which have been presented in the literature.

6. Conclusions

Test data generation as one of the important and challenging tasks in the development of high-quality software was considered in this work. The EvoSuite test data generation suite was selected for the development and performance study of the proposed method. Previously, the EvoSuite team has developed some methods for test data generation based on the genetic algorithm. These methods have shown good performances.

Previous studies on the swarm-intelligence-based showed that this class of optimization methods could efficiently solve many engineering methods. Also, these methods have shown competitive performance in comparison with evolutionary methods such as GA in other engineering fields. Based on these facts, this work was aimed to design a new class of optimization methods in EvoSuite. In addition, the performances of these methods were studied in comparison with GA methods. Due to the discreteness of the test data generation problem, the GA based method could be efficiently adopted. However, most of the swarm intelligence based methods such as ICA, PSO, TLBO, and Firefly have been designed for continuous problems. Hence, these methods need to modify using proper methods to work in discrete search spaces. Also, it is possible to map a discrete problem into the continuous one and apply the swarm intelligence-based methods.

The applicability of the swarm-based methods was studied for test data generation as a discrete problem. The results showed that these methods were efficient to solve the problem at the hand. Although, the best results in most cases are obtained by the GA methods, the swarm-based methods show competitive results. Also, in some cases, the swarm-based methods provided better performances. We hope that by incorporating more efficient modifications in the swarm-based methods better performances could be achieved. Modifications in solution representation, movement patterns, social/psychological, neighbour selection are recommended. Also, it may be useful to use the potentials of other test data generation methods and incorporate them into the swarm-based methods.

7. References

[۱] رافع، وحید، اسفندیاری، سجاد. راهکاری نوین جهت تولید دنباله آزمون کمینه در فرآیند آزمون نرم افزار با ترکیب الگوریتم های جســـتجوی تپه نوردی و جستجوی خفاش. مجله مهندسی برق دانشگاه تبریز, ۴۶(۳), ۲۵-۳۵، ۲۰۱۶.

[۲] عســـگری عراقی، رافع، وحید، کلائی، اکرم. تولید مورد آزمون مبتنی بر مدل از تو صیفات تبدیل گراف با ا ستفاده از الگوریتم ج ستجوی پرتو. مجله مهندسی برق دانشگاه تبریز, ۴۹(۱), ۳۴۳-۳۵۶، ۲۰۱۹.

[3] M. Khari, P. Kumar, "An extensive evaluation of search-based software testing: a review", *Soft Computing*, Vol. 23, no. 6, pp.1933-1946, 2019.

[4] A. Aleti, I. Moser, L. Grunske, "Analysing the fitness landscape of search-based software testing problems", *Automated Software Engineering*, Vol. 24, no. 3, pp. 603-621, 2017.

[5] G. Fraser, A. Arcuri, “EvoSuite : Automatic Test Suite Generation for Object-Oriented Software,” 19th ACM SIGSOFT Symposium and 13th European Conference on Found. Software Engineering, 2011, pp. 416–419.

[6] P. McMinn, “Search-Based Software Testing: Past, Present and Future,” in 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, 2011, pp. 153–163.

[7] D. Bruce, D. Menéndez, H. D. Clark, D. Dorylus, "An ant colony based tool for automated test case generation", In International Symposium on Search Based Software Engineering, August 2019, Springer, Cham, pp. 171-180.

[8] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation", In International Symposium on Search Based Software Engineering, September 2015, Springer, Cham, pp. 93-108.

[9] D. Bruce, H. D. Menéndez, E. T. Barr, D. Clark, "Ant Colony Optimization for Object-Oriented Unit Test Generation", In International Conference on Swarm Intelligence, Springer, Cham, October 2020, pp. 29-41.

[10] N. Jatana, B. Suri, "An improved crow search algorithm for test data generation using search-based mutation testing", *Neural Processing Letters*, Vol. 52, no. 1, pp. 767-784, 2020.

[11] J. Campos, Y. Ge, G. Fraser, M. Eler, A. Arcuri, "An empirical evaluation of evolutionary algorithms for test suite generation", In International Symposium on Search Based Software Engineering, Springer, Cham, September 2017, pp. 33-48.

[12] J. M. Rojas, M. Vivanti, A. Arcuri, G. Fraser, "A detailed investigation of the effectiveness of whole test suite generation", *Empirical Software Engineering*, Vol. 22, no. 2, pp. 852-893, 2017.

[13] M. Mehdi, D. Shahabi, S. P. Badiei, S. E. Beheshtian, R. Akbari, and S. M. Reza, “On the Performance of EvoPSO : a PSO Based Algorithm for Test Data Generation in EvoSuite", In 2nd Conference on Swarm Intelligence and Evolutionary Computation (CSIEC), Kerman, Iran, 2017, pp. 129–134.

[14] M. M. D. Shahabi, S. P. Badiei, S. E. Beheshtian, R. Akbari, S. M. R. Moosavi, "EVOTLBO: A TLBO based Method for Automatic Test Data Generation in EvoSuite", *International Journal of Advanced Computer Science and Applications*, Vol. 8, no. 6, 2017.

[15] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application", In 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), May 2017, pp. 263-272.

[16] S. Shamshiri, J. M. Rojas, L. Gazzola, G. Fraser, P. McMinn, L. Mariani, A. Arcuri, "Random or evolutionary search for object-oriented test suite generation", *Software Testing, Verification and Reliability*, Vol. 28, no. 4, e1660, 2018.

[17] C. Oliveira, A. Aleti, L. Grunske, K. Smith-Miles, "Mapping the effectiveness of automated test suite generation techniques", *IEEE Transactions on Reliability*, Vol. 67, no. 3, pp. 771-785, 2018.

[18] N. M. Albunian, "M. Diversity in search-based unit test suite generation", In International Symposium on Search Based Software Engineering, Springer, Cham, September 2017, pp. 183-189.

[19] G. Gay, "Generating effective test suites by combining coverage criteria", In International Symposium on Search Based Software Engineering, Springer, Cham, September 2017, pp. 65-82.

[20] M. Olsthoorn, P. Derakhshanfar, X. Devroey,"An Application of Model Seeding to Search-Based Unit Test Generation for Gson", In International Symposium on

Search Based Software Engineering, Springer, Cham, October 2020, pp. 239-245.
[21] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H. C. Gall, A. Bacchelli, "On the effectiveness of manual and automatic unit test generation: ten years later", In 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), May 2019, pp. 121-125.
[22] G. Rudžionienė, Š. Packevičius, E. Bareiša, "Directed multi-target search based unit tests generation", In International Conference on Information and Software Technologies, Springer, Cham, October 2019, pp. 90-109.
[23] H. Almulla, G. Gay, "Generating Diverse Test Suites for Gson Through Adaptive Fitness Function Selection", In International Symposium on Search Based Software Engineering, Springer, Cham, October 2020, pp. 246-252.
[24] B. Evers, P. Derakhshanfar, X. Devroey, A. Zaidman, "Commonality-Driven Unit Test Generation" In International Symposium on Search Based Software Engineering", Springer, Cham, October 2020, pp. 121-136.
[25] G. Fraser, A. Arcuri, "Evosuite: On the challenges of test case generation in the real world", In 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, March 2013, pp. 362-369.
[26] G. Fraser, A. Arcuri, and P. McMinn, “Test Suite Generation with Memetic Algorithms,” Gecco’13 Proceeding of Genetic and Evolutionary Computation Conference, 2013, pp. 1437–1444.
[27] R. V. Rao, V. J. Savsani, D. P. Vakharia, “Teaching–learning-based optimization: A novel method for constrained mechanical design optimization problems”, *Comput. Des.*, Vol. 43, no. 3, pp. 303–315, 2011.
[28] R. Eberhart, J. Kennedy, “A New Optimizer Using Particle Swarm Theory,” pp. 39–43, 1995.
[29] D. YueMing, W. YiTing, and W. DingHui, “Particle swarm optimization algorithm for test case automatic generation based on clustering thought,” In 2015 IEEE International Conference on Cyber Technology in Automation, Control, and Intelligent Systems (CYBER), pp. 1479–1485.
[30] E. Atashpaz-Gargari, C. Lucas, “Imperialist competitive algorithm: An algorithm for optimization inspired by imperialistic competition”, 2007 IEEE Congress on Evolutionary Computation, CEC 2007, pp. 4661–4667.
[31] I. Fister, I. Fister Jr, X. S. Yang, J. Brest, "A comprehensive review of firefly algorithms", Swarm and Evolutionary Computation, 2013, pp. 34-46.