Journal of Electronic Testing (2024) 40:539–556
https://doi.org/10.1007/s10836-024-06136-4

RESEARCH

# An Automatic Software Testing Method to Discover Hard-to-Detect Faults Using Hybrid Olympiad Optimization Algorithm

**Leiqing Zheng[1] · Bahman Arasteh[2,3] · Mahsa Nazeri Mehrabani[4] · Amir Vahide Abania[5]**

Received: 24 March 2024 / Accepted: 21 August 2024 / Published online: 10 September 2024


**Abstract**
The enhancement of software system quality is achieved through a process called software testing, which is a time and cost-intensive stage of software development. As a result, automating software tests is recognized as an effective solution that can simplify time-consuming and arduous testing activities. Generating test data with maximum branch coverage and fault discovery capability is an NP-complete optimization problem. Various methods based on heuristics and evolutionary algorithms have been suggested to create test suites that provide the most feasible coverage. The main disadvantages of past approaches include inadequate branching coverage, fault detection rate, and unstable results. The main objectives of the current research are to improve the branch coverage rate, fault detection rate, success rate, and stability. This research has suggested an efficient technique to produce test data automatically utilizing a hybrid version of Olympiad Optimization Algorithms (OOA) in conjunction with genetic algorithm (GA) operators theory. Maximum coverage, fault detection capability, and success rate are the main characteristics of produced test data. Various experiments have been conducted on the nine standard benchmark programs. Regarding the results, the suggested method provides 99.92% average coverage, a success rate of 99.20%, an average generation of 5.76, and an average time of 7.97 s. Based on the fault injection experiment's results, the proposed method can discover about 89% of the faults injected by mutation testing tools such as MuJava.

**Keywords** Software testing · Branch coverage · Hard-to-cover codes · Olympiad optimization algorithms · Branch coverage · Fault detection score

## 1 Introduction

The process of software testing is crucial to ensuring the quality of software. Software testing can be conducted either manually or automatically. Manual tests are more time-consuming and expensive, whereas automated methods reduce testing time and cost. Due to its importance, automated software testing is considered a noteworthy challenge and concern within the research field. Implementing conventional manual testing methods can prove to be extremely expensive and time-consuming when it comes to real-world, large-scale software systems. Detecting faults in software development using source-code level testing merely leads to identifying 50% of the faults. Using automatic software testing can significantly decrease cost and time. The study focuses on the issue of implementing optimal test cases automatically at the source-code level.

Generating test data automatically with maximum branch coverage in the shortest possible time is the primary optimization challenge in the field; this challenge motivates researchers to suggest effective and automatic test data generators. Choosing a limited subset of the combination of all the possible inputs from the entire set with maximum coverage of program branches is considered an NP-complete problem [1, 2]. Various heuristic and evolutionary

Responsible Editor: Y. K. Malaiya

✉ Bahman Arasteh
bahman.arasteh@yahoo.com

[1] Rizhao Polytechnic, Rizhao City, Shandong Province, China

[2] Department of Software Engineering, Faculty of Engineering and Natural Science, Istinye University, Istanbul, Turkey

[3] Department of Computer Science, Khazar University, Baku, Azerbaijan

[4] Department of Electrical and Computer Engineering, University of Tabriz, Tabriz, Iran

[5] Department of Computer Engineering, Tabriz Branch, Islamic Azad University, Tabriz, Iran

algorithms have been suggested for creating test datasets that provide the maximum coverage due to the nature of this research issue [3–5]. Previous methods have faced significant drawbacks, including failure to achieve maximum branch coverage, lack of stable results in various executions, poor average success rate in data productions, and high execution time. The present study has the following objectives:

- Generating test data with maximum code coverage.
- Improving the fault detection rate of the test data.
- Improving the success rate of heuristic-based test-generation methods.
- Improving the stability of the results of the test data generation methods.
- Improving the success rate of the test data generation method.

This research has introduced an efficient approach to generate test data automatically. The Olympiad optimization algorithm (OOA) is employed to achieve this objective, particularly concerning the GA operators. An algorithm known as OOA, which is a heuristic discrete algorithm [18], was designed in this study to address the problem of generating test data. The main objective of the proposed approach is to generate test data that achieves the maximum branch coverage within a given time limit. The following are the noteworthy achievements of this research:

- Enhancing the Olympiad optimization algorithm (OOA) with crossover and mutation operators for generating the optimized test data.
- Generating optimal and stable test data that offers maximum branch coverage and fault discovery rate in the shortest possible time.
- Covering the hard-to-cover codes of a program to discover hard-to-find faults by the suggested hybrid OOA.
- Developing a tool to test a program automatically with a high success rate.

The following sections of this paper are arranged as follows: Sect. 2 outlines the basic theoretical concepts and provides a brief review of related works. Section 3 reports and explains the mentioned method in detail. Section 4 discusses the outcomes of executing the program, which is implemented with various methods. Finally, the study ends in Sect. 5, where results and directions for future research are provided.

## 2 Related Work

In [3], test data were generated randomly by the researchers. The most significant drawbacks of this strategy are thought to be the extremely time-consuming for reaching the necessary coverage. Furthermore, these procedures yielded different results for the number of faults found. To improve outcomes, researchers, therefore, suggested a symbolic execution-based approach [4, 5]. A testing methodology, which is called symbolic execution, works well for automatically producing test data inputs that cause software errors. One of the main advantages of symbolic execution is the production of concrete test inputs; the test data generated has high coverage. Pointer and array values cannot be determined via symbolic techniques. As a result, data creation has become a difficult problem. A simulated annealing approach was used in [6] to solve the test-data creation issue. In this paper, the simulated annealing approach generates ideal test data by transforming the challenge of testing data production into an optimization problem. The primary drawbacks of this strategy are its poor coverage, Poor efficiency, and placement in the local optimal region. This approach is suitable for behavioral assessments.

A technique was suggested in [2] for generating test data by using a genetic algorithm (GA). GA was used in this technique to choose the best routes. The goal of this study's fitness function, known as the similarity function, was to ascertain how similar the traveled route was to the goal path. Route (path) optimality is the state in which test data is executed while adhering to the route. In other words, a path's optimality increases with its follow-up. The obtained results indicated improvement in test data production. The amount of time needed to discover the best route is decreased when GA is used. In [7], a different approach based on GA for generating test data was put out. GA was used in this study to get the best possible test results. The algorithm was used in parallel by researchers to increase efficiency and effectiveness. Subsequently, the suggested method's coverage was examined and assessed using six benchmark programs. The outcomes showed that test data output has improved.

One of the main issues with GA is that chromosomes do not strive to develop themselves and may only improve by mutation operator. A chromosome's subsection cannot be evaluated in GA; the fitness function in GA only assesses the whole chromosome. Because of this, the GA and blind search algorithms are comparable. An automated test-data creation technique has been presented as a potential remedy to these issues [8]. This approach enhanced the GA using reinforcement learning as a memetic search technique. This enhanced GA concentrates on the population's best chromosomes, and Q-learning has been used to direct this search procedure. In this strategy, repeated sub-sections inside a

chromosome result in the execution of the mutation operator. Based on results from experiments, this hybrid approach outperforms GA in terms of success rate and coverage. The particle swarm optimization (PSO) technique was used by researchers in [9] to generate test data. PSO algorithm and regression analysis were also suggested in [10] as methods for producing test data in a similar spirit. Furthermore, the PSO method was used in [11] to provide test data with a different objective function because of its ease of use and rapid convergence. According to the outcomes of these searches, branch distance functions do a poor job of covering critical routes[1], or fault-prone routes.

The shuffled frog leaping algorithm (SFLA) is introduced in [12] to produce effective structural test data. The suggested SFLA algorithm has a fast rate of convergence and is easy to implement. Branches coverage is employed as the fitness function in the proposed SFLA to produce useful test data. Seven benchmark programs were utilized to compare the proposed SFLA's performance with those of the Genetic algorithm (GA), particle swarm optimization (PSO), ant colony optimization (ACO), and artificial bee colony (ABC). According to the findings, the SFLA-based method outperforms the GA, PSO, and ACO regarding the average branch coverage, success rate, and average number of iterations for covering all branches. The efficiency of many methods, including genetic algorithm, particle swarm, simulated annealing, and artificial bee colony (ABC), has been assessed and contrasted in [13]. The fitness function in this research is a distance function depending on branch coverage. According to the outcomes of studies using the conventional benchmarks, ABS has higher coverage and a success rate than the SA, GA, PSO, and ACO. In fact, when generating the best test data, the ABC algorithm outperformed the other algorithms.

An approach using the ant colony optimization (ACO) algorithm was put out in [14] to generate the best test data possible with the greatest possible branch coverage. This technique created a special fitness function depending on coverage. The experiment findings show that this approach has greater coverage, convergence, and more stable outcomes. To cover the critical path of the program being tested, an ICA-based test-creation technique was presented [15] that uses an enhanced fitness function. The prior research, which is briefly discussed in Table 1, supports the claim that each of the methods that have been suggested so far has advantages and disadvantages of its own. Stated differently, the issue of data production for automated software testing cannot be addressed thoroughly. Thus, an automated test-generating technique using the Olympiad optimization algorithm was used in this research work. The preceding research gap is attempted to be addressed by the writers of this study. In Sect. 3, the suggested approach is covered in detail.

**Table 1** The main characteristics of the test generation methods

| Method | Merits | Demerits |
|---|---|---|
| Modified GA [2] | Higher Performance and Coverage | Lower success rate |
| Random search [3] | Simplicity of implementation | Lack of fitness function and lower coverage |
| SA algorithm [6] | Faster than a random search | Lower coverage and success rate |
| GA algorithm [7, 8] | Higher performance and coverage | Insufficient coverage |
| PSO algorithm [9–11] | Higher implementation speed and simplicity | Lower success rate and insufficient coverage |
| SFLA algorithm [12] | Higher stability, coverage, and success rate | High implementation cost and lower performance |
| ABC algorithm [13] | Higher speed and coverage | Lower stability |
| ACO algorithm [14] | Higher performance and coverage | Lower stability and performance |
| ICA algorithm [15] | Higher convergence speed and coverage | Lower stability |
| Scenario-based Method [16] | Scenario coverage | Design models are required |
| Hyper Heuristic Model-based Testing [17] | Higher performance and state coverage | The tradeoff between coverage and cost is required |

[1] Route and path have been used in a same meaning in this study.

## 3 The Suggested Method

As shown in Fig. 1, a heuristic-based method was suggested to generate program test data.

The source code of the tested program is statistically analyzed (as the first stage), and the structural information needed for the next steps is extracted. The source code is analyzed automatically, and the number and types of input data, program execution paths, and branches are determined. The second step generates optimal test data using OOA (Olympiad Optimization Algorithm). The fitness function used in this study was defined based on the branch coverage. The output is the test set (test suit) produced by the OOA. Finally, the effectiveness of the generated test data is evaluated in terms of branch coverage and fault discovery probability using a mutation test.

### 3.1 Source Code Parsing

In the first stage, the source code of a program is automatically analyzed. The program source code provides information on the number of inputs, their domain and data type,

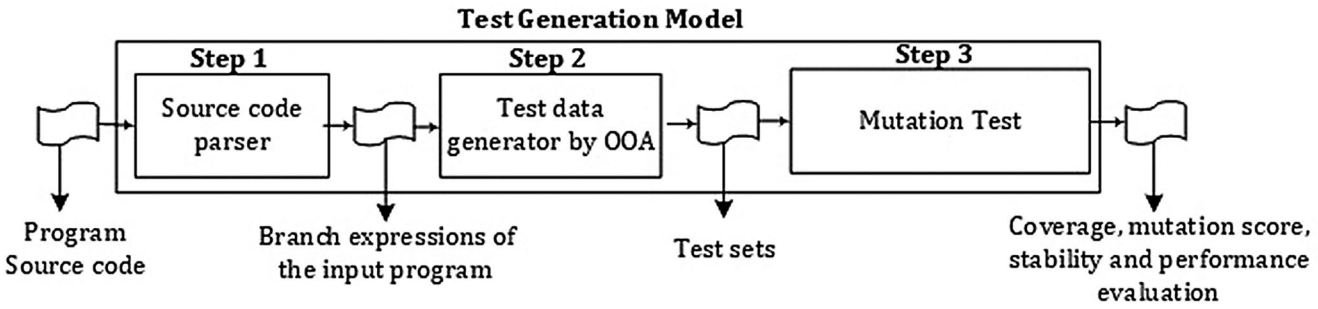



**Fig. 1** The model of the suggested method

```
public static int TriangleType(int a, int b, int c)
1    {
2        int train;
3        if (a <= 0 || b <= 0 || c <= 0)
4            return 0;
5        train = 0;
6        if (a == b)
7            train = train + 1;
8        if (a == c)
9            train = train + 2;
10       if (b == c)
11           train = train + 3;
12       if (train == 0)
13           if (a + b < c || a + c < b || b + c < a)
14               return 0;
15           else
16               return 1;
17       if (train > 3)
18           return 3;
19       if (train == 1 && a + b > c)
20           return 2;
21       else if (train == 2 && a + c > b)
22           return 2;
23       else if (train == 3 && b + c > a)
24           return 2;
25       return 0;
26   }
```

Source Code Parser

Structural information of the input program

| | |
|---|---|
| Input 1 : integer | a |
| Input 2 : integer | b |
| Input 3 : integer | c |
| Num. of Branch | 9 |
| Cyclomatic Complexity | 34 |
| Branch Instructions | - |

**Fig. 2** The input and output of the suggested parser for the triangle benchmark program

the number of conditional instructions in the program, and the expressions converted into conditional instructions. The first step in the suggested procedure is executed automatically by the implemented code. For this stage, the time complexity of the suggested technique is O(n). Figure 2 displays the input and output of the first stage of the recommended method.

### 3.2 Test Data Generation via OOA

This section discusses the suggested method for generating test data using OOA. OOA starts with several random initial populations, each referred to as students. The proposed OOA uses swarm-based imitation as its local and global search strategies [18]. This heuristic approach uses a divide-and-conquer framework to solve optimization problems using a population and group-based methodology. Under this method, every individual in OOA imitates a student's conduct in a classroom, especially those students getting ready for an Olympiad exam. The population is evolving because of the iterative teaching and learning processes in which its members (kind to students) engage. This algorithm combines local and global search techniques in a divide-and-conquer manner. Figure 3 depicts the steps of the suggested procedure.

Agents in the population are divided into smaller groups, and each of these subgroups uses a particular imitation strategy to investigate different areas of the total solution space. The student's implementation structure for the test generation problem is shown in Fig. 4. Students compete with one another as they share knowledge. Every student keeps track of their learning rate, indicating where they are in the learning process. Every student in the OOA is represented as a numeric array, and the student population comprises different solutions. As illustrated in Fig. 3, the first stage of OOA involves splitting the student body into n teams, each with

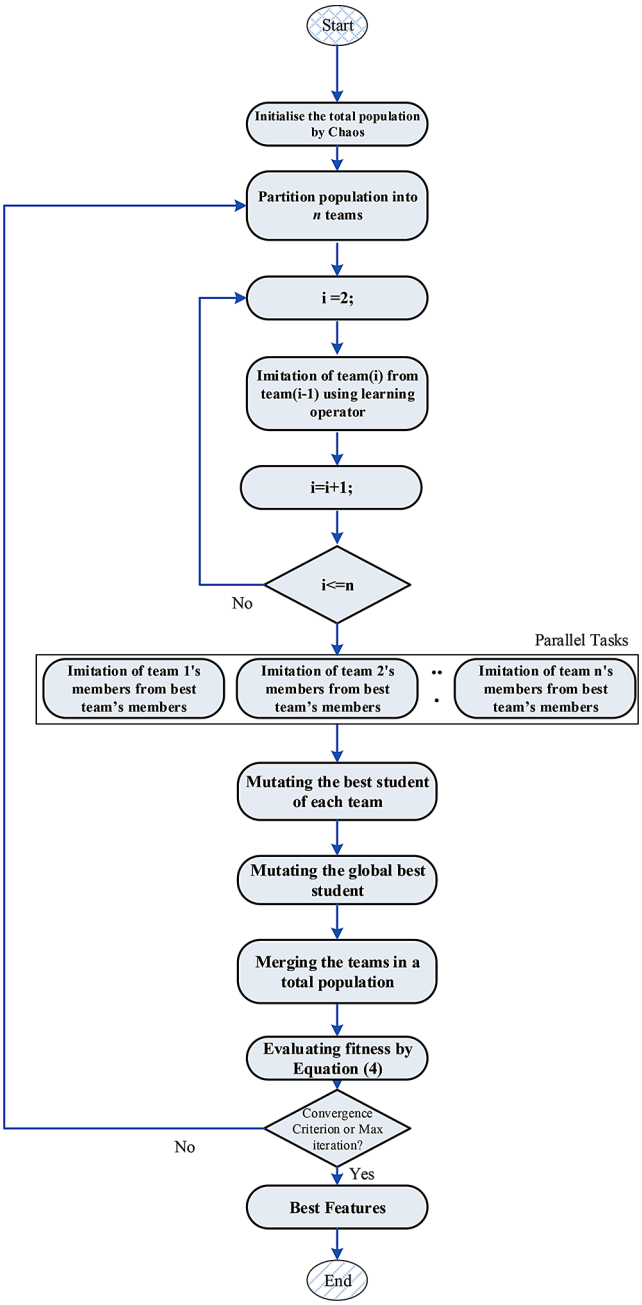



**Fig. 3** Workflow of the hybrid and OOA for test generation [18]

m students, after the sorting process. Students work in teams and explore the local solution space within each team.

The lead student of the first team is the world's best student; in each subsequent team, the lead student is the local best student.

In OOA, team members seek to learn from the best members of their neighboring team or the globe's best-performing team. A key element of OOA is the learning operator, which makes local and global search operations easier. To put it another way, learning is the main way that OOA finds the best answer while also trying to improve the general knowledge or fitness of the population. The learning operator aims to increase the population's level of knowledge. Students are grouped according to their knowledge, demonstrating OOA's effort to use this operator to increase population knowledge. The learning operator consists of four essential phases. Students on each team try to pick up knowledge from their teammates in the first step. Knowledge from the first team is shared with other teams after the learning operator's initial phase. The suggested learning operator, which transfers a team's knowledge to its neighboring team, is demonstrated in Algorithm 1. Knowledge moves from one team to its neighboring team stepwise through this sequential knowledge transfer process, similar to how the bubble sort algorithm operates.

```
Function studentType Olampiyad_Learn(BestStd,WorstStd)
{
    nVar= length(Student_array);
    LearnCount=ceil((30 * nVar)/100);   %imitation count
    count=0;
    i = 0;
    NewStd=WorstStd;
    while (count<=LearnCount and i < nVar)
     {
      aL=randi(nVar);
      if (WorstStd(aL)!=BestStd(aL))
        {
        NewStd(aL)=BestStd(aL);
        count=count+1;
        }
      i = i + 1;
      }
 Return(NewStd);
}
```

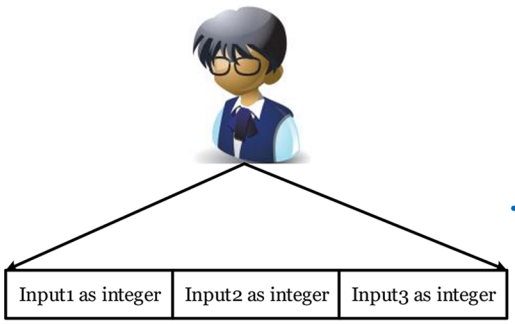
**Algorithm 1** The pseudo-code of the OOA's learning operator

### 3.3 Fitness Function

To compare the efficacy of the solutions and choose the best one, an objective function known as the fitness function has been created. Choosing the appropriate fitness function is one of the most crucial steps in solving optimization problems. The input program has *s* branches. The program's several branches are chosen using *bchi* variable. This research includes a collection of created test data in the test suite denoted by *TS*. If there are precisely *m* inputs, input data is given by the variable $X_k \epsilon$ *TS* (1 < k < m). In the suggested method, Eq. (1) Is used to calculate the fitness of a test set.

$$Fitness\,(X_k) = \frac{1}{[\delta + \sum_{i=1}^{s} w_i\, f(bch_i, X_k)]^2} \tag{1}$$

Since $\delta$ is a constant, its value can be found via trial and error. It has a value of 0.01. In source code, a variable denotes the weight of a branch. In this case, the branch distance function is denoted by f. Table 2 was used to calculate the branch distance ($f(bch_i)$). A branch instruction's conditional statement determines its distance function. Equation (2) is used




Structure of a student (agent) as array for the triangleType program

| | input1 | input2 | input3 |
|---|---|---|---|
| STD1 | 50 | 4 | 17 |
| STD2 | 2 | 1 | 21 |
| STD3 | 29 | 3 | 8 |
| STD4 | 12 | 5 | 2 |
| STD5 | 12 | 25 | 17 |
| STD6 | 20 | 19 | 12 |
| STD7 | 9 | 13 | 8 |
| STD8 | 1 | 1 | 14 |

Population of Students

**Fig. 4** The array structure of the implemented agent (student) in the suggested OOA for the *tiangleType* benchmark program

**Table 2** The predicates of the branch instructions and their distance value [12]

| No. | $\mathrm{bch_i}$(predicate) | Branch-Distance ( $f(bch_i)$) |
|---|---|---|
| 1 | Boolean | If true then 0 else $\delta$ |
| 2 | $p = q$ | If $(\mathrm{abs}(p - q) = 0)$ then 0 else $(\mathrm{abs}(p - q) + \delta)$ |
| 3 | $p \neq q$ | If $(\mathrm{abs}(p - q) = 0)$ then 0 else $\delta$ |
| 4 | $p < q$ | If $(p - q < 0)$ then 0 else $(\mathrm{abs}(p - q) + \delta)$ |
| 5 | $p \leq q$ | If $(p - q \leq 0)$ then 0 else $(\mathrm{abs}(p - q) + \delta)$ |
| 6 | $a > b$ | If $(q - p < 0)$ then 0 else $(\mathrm{abs}(q - p) + \delta)$ |
| 7 | $a \geq b$ | If $(q - p \geq 0)$ then 0 else $(\mathrm{abs}(q - p) + \delta)$ |
| 8 | $a$ and $b$ | $(f(p) + f(q))$ |
| 9 | $a$ or $b$ | $\min(f(p), f(q))$ |

to determine the fitness function of each test data in the test suit (TS).

$$Fitness\,(TS) = 1/\left[\delta + \sum_{i=1}^{s} w_i \min\{f\,(bch_i, X_k)\}_{k=1}^{m}\right]^2 \quad (2)$$

Distance function was used to determine the created test suite's fitness. The distance of a branch (bchi) was determined by Table 2. If the value of an expression is true via the test data, the distance vlaue will be zero, as shown in Table 2; if not, the variable value will be added to the expression's value. The $\delta$ value is 0.01. If all of the branches are covered by the test set (TS), then maximal coverage is achieved, and Eq. (2) is evaluated as $1/\delta$. In Eq. (2), branch weight ($w_i$) is the distance function's coefficient. Variable $w_i$ indicates the branch's weight in the branch instruction. Equation (3) is used to determine the branch weight of a branch instruction, which is the product of its predicate weight and nesting level. The weight of the ith branch is represented by $w_i$, while the equilibrium coefficient is shown by λ. The value

**Table 3** The operators' weight in the conditional expressions' predicate [12]

| Operator | Weight of operator ($\mathrm{w_r}$) |
|---|---|
| = = | 0.9 |
| <, <=, >, >= | 0.6 |
| Boolean | 0.5 |
| ! = | 0.2 |
| = = | 0.9 |

of λ has been fixed to 0.5 in the experiments. The following variables determine the branch weight:

- Branch level (nesting level).
- Expression weight.

Using Eq. (4), the nesting level, also known as the branch level, is the first element of the branch weight. The result of this equation indicates the branch's nesting level. It will be more challenging to access a branch with a greater nesting level. The variable $i$ represents a specific branch where $(1 \leq i \leq S)$, and the variable $nl_i$ represents the ith branch's nesting level. The lowest branch level of the program is indicated by the variable *nlmin*, which in this example is 1. The variable nlmax in the program indicates the maximum branch level. Equation (5) is used to normalize the level of each branch instruction.

$$w_i = \lambda \;\; wn'\,(bch_i) + (1 - \lambda)\;\; wp'\,(bch_i) \quad (3)$$

$$wn\,(bchi) = \frac{nl_i - nl_{\min} + 1}{nl_{\max} - nl_{\min} + 1}, \quad (4)$$

$$wn'\,(bch_i) = \frac{wn\,(bch_i)}{\sum_{i=1}^{s} wn\,(bch_i)} \quad (5)$$

The complexity of the expressions is shown by variable $w_p$, which is the second element of the branch weight. The expression weight of a branch is calculated using Table 3

and Eq. (6). There are *h* predicates in every expression. The expression weight is equal to the square root of the weight of the predicates when the branch instruction is made up of h predicates that have been joined together using the "and" operator. When h expressions are joined in a branch instruction using the "or" operator, the expression weight is equal to the minimum of the predicate weight. Equation (7), which divides the expression weight of each branch by the overall weight of the branches, was used to normalize the weight of an expression.

$$wp\left(bch_i\right) = \begin{cases} \sqrt{\sum_{j=1}^{h} w_r^2\left(c_j\right)}, \ if\ the\ conjunction\ is\ and, \\ \min\left\{w_r\left(c_j\right)\right\}, \ if\ the\ conjunction\ is\ or, \end{cases} \tag{6}$$

$$wp'\ \left(bch_i\right) = \frac{wp\left(bch_i\right)}{\sum_{i=1}^{s} wp\left(bch_i\right)} \tag{7}$$

The ith branch (1 < i ≤ s) is shown by bchi in Eq. (6). The number of predicates that are accessible in the corresponding branch is indicated by the variable *h*. In this case, $c_j$ denotes the corresponding branch's jth conditional predicate (1 < j ≤ h). Table 3 is used to ascertain the value of an operator whose weight is represented by the variable $w_r$ in the predicate. This table lists the operators that might be used in the different predicates.

**Table 4** The implemented test generator algorithms' configuration parameters and their values

| GA configuration parameters | Value |
|---|---|
| Length of chromosome | Num of arguments |
| Crossover rate | 0.80 |
| Mutation rate | 0.05 |
| Termination condition | 100 Iterations |
| ACO configuration parameters | Value |
| Pheromone ($\tau$) | 1.0 |
| The Q parameter's value | 1.0 |
| Pheromone weight(α) | 1.0 |
| Evaporation rate ($\rho$) | 0.1 |
| Stop condition | 100 Iterations |
| PSO configuration parameters | Value |
| InertiaWeight | 0.8 |
| C1 | 2 |
| C2 | 2 |
| Number of Particles | 25 |
| Termination condition | 100 Iterations |
| ABC configuration parameters | Value |
| Lower bound of the parameters | -3 |
| Upper bound of the parameters | 100 |
| Limit control paramter | Program dependent |
| Number of onlooker bees | 50% population |
| Termination condition | 200 Iterations |
| OOA configuration parameters | Value |
| Quantity of students | 40 |
| Quantity of teams | 10 |
| Size of teams | 4 |
| Rate of learning | Random values between [0.2–0.8] |
| Imitation count | 1 |
| Termination condition | 100 Iterations |

## 4 Results and Discussion

### 4.1 Implementation System

The suggested approach was implemented using Matlab. A functional language for computational activities is Matlab. Numerous computational, programming, and display activities can be carried out with it. The Matlab tool makes it simple to implement most computational approaches and problems, particularly those involving vector and matrix formulas. To create test data for this study, Matlab was used to implement the suggested method as well as additional techniques based on the GA, ACO, PSO, and SA algorithms. The configuration settings of the tested generator algorithms are displayed in Table 4. A computer system equipped with an Intel Corei7 CPU and 8GB of RAM was used to test the suggested approach as well as other approaches. The following are the evaluation criteria that were applied in this study:

1. Average coverage, or AC, measures how well program branches are covered by the test data created.
2. The success rate, or SR, indicates the likelihood that the test data produced will cover each program branch.
3. Average Generation Convergence, or AG, is a condition that shows how many iterations an algorithm will typically need to cover all program branches.
4. AT (average time): This index shows the average amount of time required to cover each program branch. Milliseconds (ms) are used to measure this criterion.
5. MS (mutation score): This criterion shows the likelihood that the generated test data will reveal a defect. Faults have been introduced into the application and are being tested using Mujava-based mutation testing tools.

Extensive series of experiments have been conducted to answer the following questions:

1. RQ1: Can OOA generate test data with higher branch coverage?
2. RQ2: Can OOA improve the SR of the test data generation methods?
3. RQ3: Can OOA decrease the AG and AT of the test data generation methods?
4. RQ4: Can OOA generate effective test data with higher fault discovery probability?

In order to provide optimal test data, the aforementioned algorithms consider the number of iterations as the termination condition. There can be up to 100 iterations of the algorithm. Additionally, each algorithm was performed 50 times on each benchmark program to determine the average value for these criteria.

## 4.2 Benchmark Programs

Nine conventional programs with varying levels of complexity were employed in this investigation. The features of each of these benchmark programs, which have also been utilized in other earlier studies, are displayed in Table 5. The benchmark programs' source codes are written in Matlab, C++ and C# programming languages. For these benchmark programs, test data was generated using test creation methods. A real-world application's source code is organized into classes, functions, and components.

The large programs (many millions of lines of code) comprise modules and then functions overall. On the other hand, a function should have between 20 and 100 lines of code, according to programming standards. The most well-known and often used benchmark programs in software testing studies were employed in this investigation. Furthermore, all programming constructs relevant to intricate real-world applications are included in these benchmarks. These benchmark programs use all operators and commands related to conditionals, loops, arithmetic, logic, and jumps. These programs' generated control flow diagram (structural view) indicates that their cyclic complexity is more significant than that of real-world applications.

## 4.3 Evaluation of Results

### 4.3.1 Response to the First Research Question (AC Criterion)

Numerous experiments were conducted to examine and assess the suggested approach in light of the criteria explained in Sect. 4.2. The average coverage of program branches by the generated test data is one of the evaluation metrics. The average branch coverage was determined by executing each trial-generating method 50 times. The average branch coverage for several benchmark programs, as determined by test data produced during 50 runs, is displayed in Fig. 5. The results show that the test data generated by the suggested strategy has a larger branch coverage in most benchmark programs. Data that has more branch coverage is better able to identify the faults. Compared to the AC criterion, it was discovered that the test data produced by the suggested method is more effective and efficient. The suggested method and other methods were run 50 times on the benchmark programs. Figure 5 displays the average branch coverage generated by various algorithms' runs. Compared to test data generated by previous approaches, the suggested algorithm achieved an average coverage of 99.92%, which is more significant.

In the programs with higher cyclomatic complexity such as *triangletype*, *CalDay* and *ComplexMethod* the proposed method provides 99.98%, 99.96% and 99.80%. The average coverage provided by the PSO and ACO is about 99.57% and 99.80% respectively. In *cal* benchmark program, our method obtained 99.98% coverage meanwhile PSO, ACO, ABC algorithms had 99.90%, 99.94%, and 99.99% respectively. The obtained coverage depend on the program under test, the number of program branches, and branch weight. The proposed algorithm outperformed the other methods in most benchmark programs.

The fault discovery probability of a data set is a function of its coverage. The higher the coverage, the higher the fault discovery probability. Regarding the structure of a program, some codes of the program are hard to cover. The optimal test data can cover the hard-to-cover codes of a program. Figure 6 shows the hard-to-cover parts of the triangleTypeas program (the most complex benchmark). The test data generated by the proposed method can better cover hard-to-cover codes.

Figure 7 indicates the hard-to-cover parts of the triangleType program for ABC and OOA algorithms. The cyclomatic complexity of this program is 34, whereas the average

**Table 5** Characteristics of the 10 benchmark programs

| Program | #Arg | #Arg.type | LOC | Cyclomatic complexity | Description |
|---|---|---|---|---|---|
| TriangleType | 3 | Int | 35 | 34 | Type classification for a triangle |
| CalDay | 3 | Int | 72 | 22 | Calculate the day of the week for a date |
| IsValidDate | 3 | Int | 41 | | Check a date is valid or not |
| cal | 6 | Int | 40 | 20 | Compute the days between two dates |
| Reminder | 2 | Int | 25 | 6 | Division and reminder of two numbers |
| printCalender | 2 | Int | 124 | 8 | Print calendar according to the input of date |
| IsPrim | 1 | Int Array | 30 | 6 | Finding prime numbers of a list of n numbers |
| Statistics | 1 | Int Array | 31 | 4 | Calculating average and standard deviation |
| Bessj | 2 | Real | 49 | 6 | Calculating Bessel function |
| ComplexMethod | 3 | Int, string, bool | 56 | 20 | Complex methods on different data type |

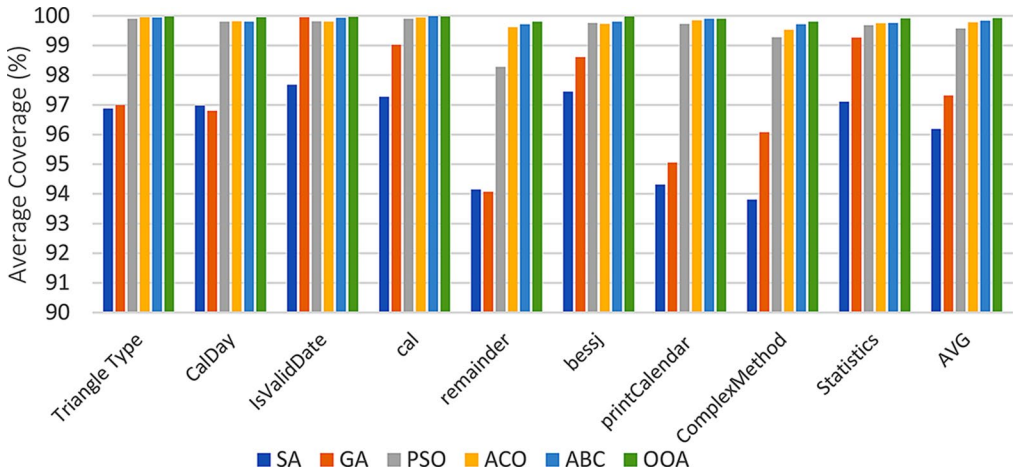



**Fig. 5** Average coverage provided by different methods during 50 times executions

```
public static int classify(int a, int b, int c)
{
    int train;
    if (a <= 0 || b <= 0 || c <= 0)
        return 0;
    train = 0;
    if (a == b)
        train = train + 1;
    if (a == c)
        train = train + 2;
    if (b == c)
        train = train + 3;
    if (train == 0)
        if (a + b < c || a + c < b || b + c < a)
            return 0;
        else
            return 1;
    if (train > 3)
        return 3;
    if (train == 1 && a + b > c)
        return 2;
    else if (train == 2 && a + c > b)
        return 2;
    else if (train == 3 && b + c > a)
        return 2;
    return 0;
}
```

Hard to cover and bug prone codes of triangle program

**Fig. 6** The hard-to-cover parts of the *triangleType* program for the SA, GA, PSO and ACO

cyclomatic of the real-world programs is less than 7. The reachability of each instruction in a program depends on its location in the program. The instructions into the nested branches (higher nesting level) have lower reachability. Furthermore, an instruction's reachability depends on the distribution of input data. In the conducted experiments, about 20% of hard-to-cover codes were not covered by the generated test data using SA and GA during 50 executions. About 0.1% of the hard-to-cover codes in the *traingleType* program were not covered by the proposed method. The proposed method covers about 70% of the uncoverable codes by SA, GA, PSO, and ACO.

```
public static int classify(int a, int b, int c)
{
    int train;
    if (a <= 0 || b <= 0 || c <= 0)
        return 0;
    train = 0;
    if (a == b)
        train = train + 1;
    if (a == c)
        train = train + 2;
    if (b == c)
        train = train + 3;
    if (train == 0)
        if (a + b < c || a + c < b || b + c < a)
            return 0;
        else
            return 1;
    if (train > 3)
        return 3;
    if (train == 1 && a + b > c)
        return 2;
    else if (train == 2 && a + c > b)
        return 2;
    else if (train == 3 && b + c > a)
        return 2;
    return 0;
}
```

The hard to cover part for ABC and OOA

**Fig. 7** The hard-to-cover parts of the *triangleType* benchmark program for the proposed method

#### 4.3.2 Response to the Second Research Question (SR Criterion)

A second series of experiments was conducted to answer the second research question. In this experiment, the success rate of the proposed method was evaluated and compared with the related techniques. Another metric for producing test data with 100% coverage is the success rate (SR). Each test data generator has been executed 50 times for every benchmark program. The SR of a test data-generating method is the average number of times the approach has achieved its maximum coverage. The test results are displayed in Fig. 8 about the SR criterion.

The results indicate that the proposed algorithm achieved a 99% success rate in triangleType. Regarding the isValidDat benchmark program, the proposed method's average SR is 99.20%. OOA performed better than the other algorithms in most of the benchmark programs; the results revealed that PSO, ACO, and ABC have higher SR than SA and GA. Therefore, it can be said that the suggested method generates more effective test data than alternative methods when considering the SR criterion.

#### 4.3.3 Response to the Third Research Question (SR Criterion)

The second criterion highlighting the expense and duration of creating test data is convergence speed. The number of iterations needed to generate test data with maximal branch coverage using a heuristic method is a good indicator of the algorithm's convergence. Effectiveness and efficiency will increase with the number of iterations required to create appropriate test data. The average convergence speed of several approaches over 50 executions is displayed in Fig. 9. Except for the isValidDate program, the findings indicated that the suggested approach has a lower convergence average across all benchmark programs. Therefore, it can be concluded that OOA performed better in this category. The suggested approach can produce ideal test data with optimum coverage in fewer iterations. The number of iterations needed to provide maximum

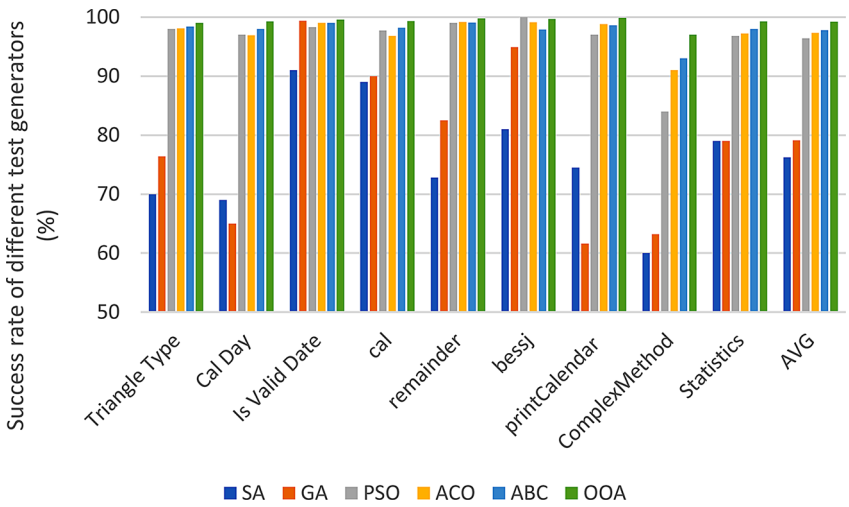



**Fig. 8** The average success rate of different test-generating methods for 50 times executions

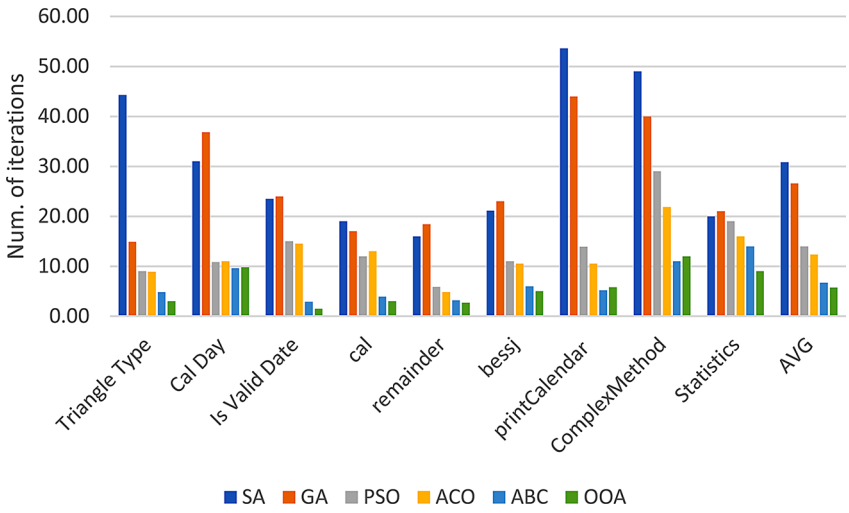



**Fig. 9** The average number of iterations to obtain maximum coverage by different test generators for 50 times executions

coverage throughout 50 runs is depicted in Fig. 9. The suggested approach provided the optimal test data with an average of 5.76 iterations, according to the results that were acquired. The data with the greatest branch coverage are referred to as the optimal test data in this study. Thus, the suggested algorithm is highly efficient and effective in terms of convergence speed.

Another criterion considered was the average execution time of each method in automatic test generation. The average execution times of 50 distinct runs in various applications are shown in Fig. 10. As a result, the average execution time was used to compare the suggested strategy with alternative approaches. Except for the *bessj* and *cal* programs, the suggested technique was executed faster for the majority of benchmark programs. Overall, the suggested approach has accomplished this requirement with success.

#### 4.3.4 Response to the Fourth Research Question (Fault Discovery)

The mutation test aimed to assess how well the test data produced by the suggested method worked. The program source code was mutated with a number of faults (mutants) by means of the Mujava tool [19, 20]. The inserted faults were found using the test data that were provided. MuJava assesses the test data's mutation score. The mutation operators used to introduce faults into the program are displayed in Table 6. Following the fault injection, the generated test data is used to test the faulty program and determine the fault detection rate (also known as the mutation score) of the test data. The ability of each test set to find the implanted faults is represented by the mutation score (MS). All mutation operators have been used for each instruction in the program source code. The fault injection probability in all instructions is the same. Figure 11 displays the mutation score (fault detection capabilities) of test data for each program. The findings validate the suggested method's

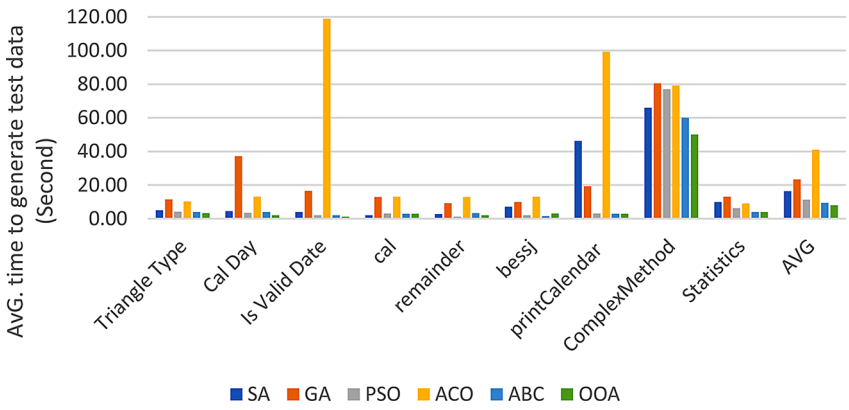



**Fig. 10** The average execution time of different test generation methods during 50 times executions

**Table 6** Mutation operators to inject fault in the program source code in the mutation test

| Operator | Description |
|---|---|
| AOR | Arithmetic operator replacement |
| AOI | Arithmetic operator insertion |
| AOD | Arithmetic operator deletion |
| ROR | Relational operator replacement |
| COR | Conditional operator replacement |
| COI | Conditional operator insertion |
| COD | Conditional operator deletion |
| LOR | Logical operator replacement |
| LOI | Logical operator insertion |
| LOD | Logical operator deletion |
| ASR | Assignment operator replacement |
| VDL | Variable deletion |
| CDL | Constant deletion |
| ODL | Operator deletion |

usefulness in generating fault detection data. This is because the suggested OOA covered about 70% of hard-to-cover codes (shown in Fig. 6) that were not covered by the SA, GA, PSO, and ACO.

**Table 7** Different mutation testing tools to evaluate the effectiveness of the test data generated by the OOA

| Tool Name | Release Year | Mutation Level | Mutation Operator |
|---|---|---|---|
| Mujava | 2004 | Byte code | Method and class |
| Muclipse | 2007 | Byte code | Method and class |
| PITest | 2011 | Byte code | Method |
| Jester | 2000 | Java code | Method |
| Jumble | 2007 | Byte code | Method |
| JavaLance | 2009 | Byte code | Method |
| Judy | 2010 | Java code | Method and class |

In order to confirm the efficacy of the suggested method, the OOA was used to generate the test data for other benchmark programs. The generated test data by OOA have been evaluated using different mutation testing tools (MuJava, Muclipse, PITest, Jester, Jumble, JavaLance, and Judy). Tables 7 and 8 illustrates the mutation testing tools.

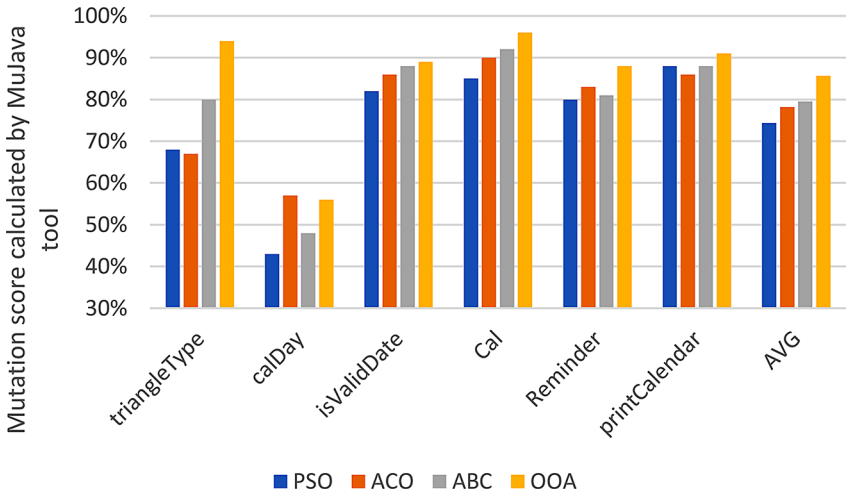



**Fig. 11** The mutation score was calculated by MuJava to evaluate the fault discovery rate of the generated test data by different methods

**Table 8** The second set of benchmark programs which were used to evaluate the OOA

| Program | Lines of code | Description |
|---|---|---|
| Bubble sort | 21 | Sorting algorithm |
| Factorial | 18 | Calculating the factorial of a number |
| Sub2 | 26 | Calculating the second root of a number |
| Fibonacci | 10 | Calculating the Fibonacci of a number |
| Perfect number | 21 | Perfect number detection |

Figure 12 shows mutation testing results on the second set of benchmarks using the generated test data by PSO, ACO, and OOA. The same mutation testing experiments have been conducted with different mutation testing tools. Different mutation scores were calculated by different mutation tools. The reason for the difference in the mutation score in different tools is related to the type and number of mutations created in different tools. As shown in Fig. 12, the mutation score of the generated test data by the OOA is higher than the generated test data by the other methods. Indeed, the generated test data by the OOA has a higher fault discovery capability. The results of different mutation testing tools (Pittest, Muclipse, Mujajav, Jester, Jumble, and JavaLancer) confirm the effectiveness of the proposed method. The mutation score of the same test data for the same benchmark program may differ in different mutation tools because of different numbers and types of injected faults. Table 9 shows the number of injected faults (mutants) in different programs by different tools.

Although different numbers of faults have been injected by different tools in the programs, the fault discovery rate of the proposed method is higher than the other methods. The average number of injected faults in seven programs by MuJava is 1136 which is higher than the other mutation tools.

The types of mutation operators used by the mutation testing tools are different. All in all, the test data generated by the proposed method can be used to discover the different faults injected by different methods. Benchmarking the proposed method by different mutation testing tools indicates its superiority to the other test generation methods. Figure 13 indicates the average mutation score of generated test data in all mutation testing tools by different methods. The average results indicate the superiority of the proposed method to other methods. Table 10 compares the success rate of different test-generating methods in different benchmark programs. Figure 14 shows an example of outputs generated by the developed test generation tool in this study.

**Table 9** The number of injected faults in the benchmark programs by different mutation testing tools

| | BubbleSort | Factorial | BinarySearch | Triangle | Sub2 | Fibonacci | perfect Number | Sum |
|---|---|---|---|---|---|---|---|---|
| Pitest | 13 | 10 | 16 | 44 | 23 | 9 | 9 | 124 |
| Muclipse | 81 | 47 | 108 | 310 | 106 | 40 | 67 | 759 |
| MuJava | 128 | 77 | 155 | 445 | 167 | 66 | 98 | 1136 |
| Jester | 10 | 9 | 13 | 47 | 24 | 11 | 11 | 125 |
| Jumble | 16 | 11 | 20 | 47 | 31 | 14 | 11 | 150 |
| JavaLancer | 19 | 12 | 26 | 67 | 59 | 20 | 11 | 214 |
| Major | 28 | 11 | 30 | 125 | 64 | 19 | 19 | 296 |
| Judy | 39 | 21 | 60 | 139 | 81 | 28 | 27 | 395 |

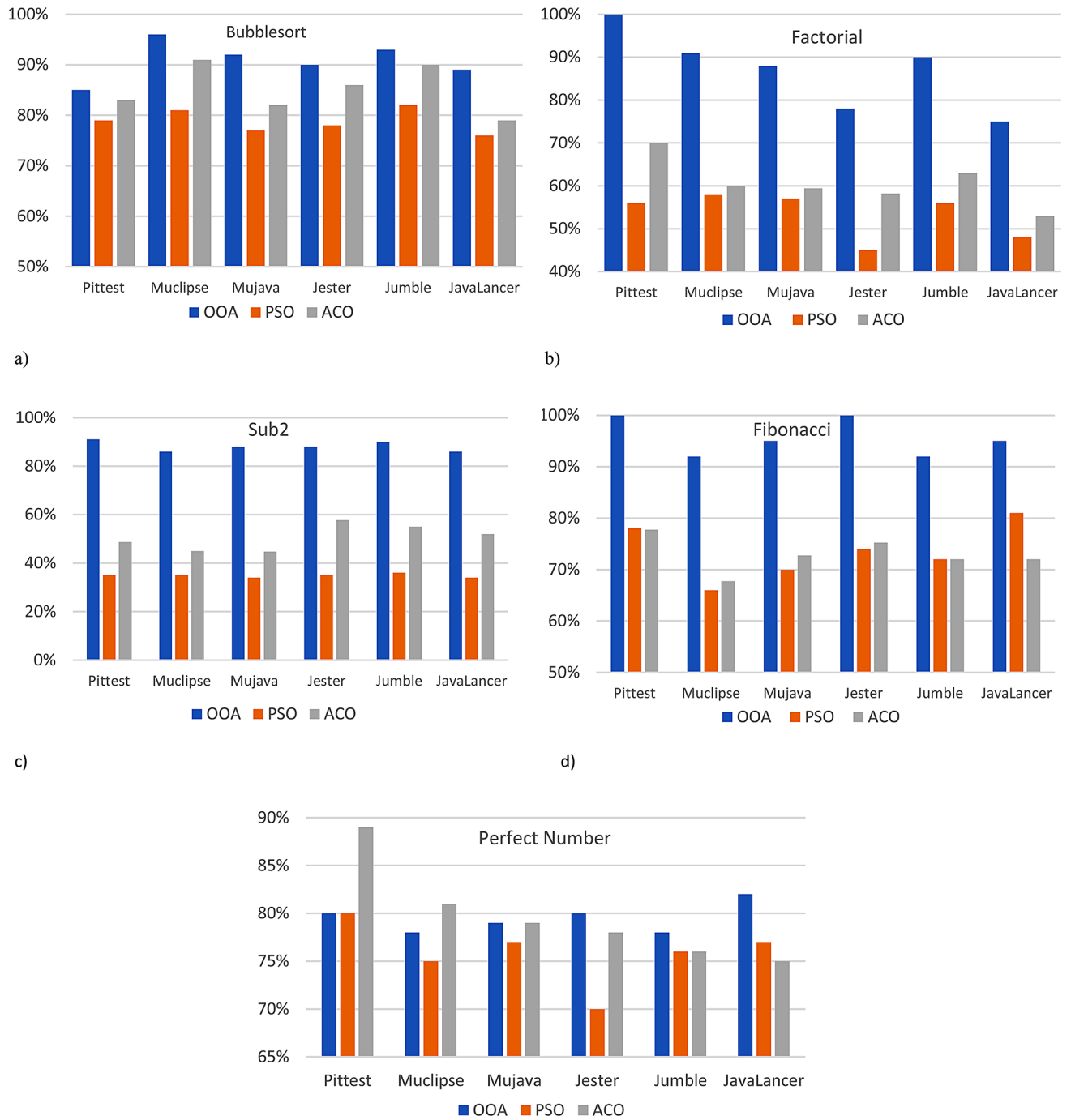



**Fig. 12** The mutation score of the generated test data by different test generation algorithms in different mutation testing tools. (**a**) Mutation score of the generated test data for the *bubblesort* program by different test generators. (**b**) Mutation score of the generated test data for the *Factorial* program by different test generators. (**c**) Mutation score of the generated test data for the *Sub2* program by different test generators. (**d**) Mutation score of the generated test data for *fibonacci* program by different test generators. (**e**) Mutation score of the generated test data for *perfect-number* program by different test generators

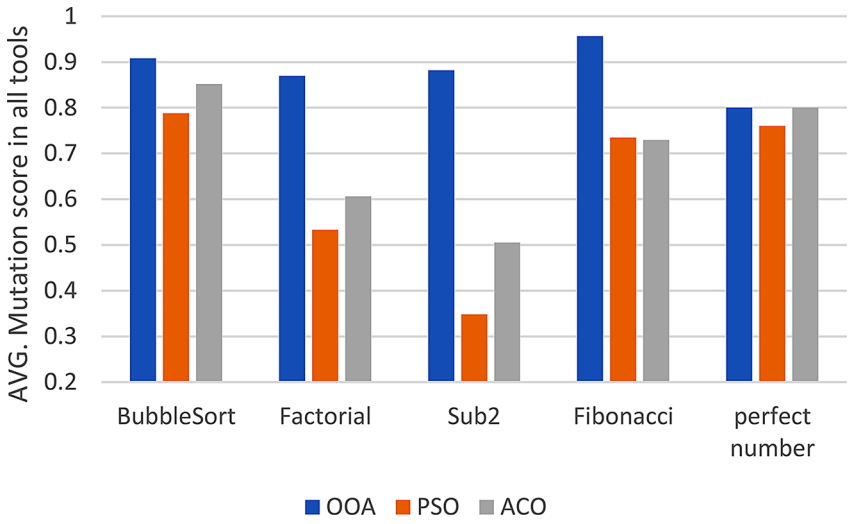



**Fig. 13** The average mutation score of the generated test data by different test generation algorithms in all mutation testing tools

**Table 10** The success rate (%) of different methods in generating test data with 100% branch coverage

| | ABC [13] | GA [7] | ACO [14] | PSO [9] | SA [6] | OOA |
|---|---|---|---|---|---|---|
| Triangle Type | 98.41 | 76.40 | 98.10 | 98.00 | 70.00 | 99.00 |
| bessj | 97.91 | 94.90 | 99.10 | 100.00 | 81.00 | 99.70 |
| Statistics | 98.00 | 79.00 | 97.20 | 96.83 | 79.00 | 99.28 |
| cal | 98.20 | 90.00 | 96.81 | 97.72 | 89.00 | 99.34 |
| ComplexMethod | 93.00 | 63.18 | 91.00 | 84.00 | 60.00 | 97.00 |
| remainder | 99.08 | 82.50 | 99.20 | 99.00 | 72.8.00 | 99.81 |
| Is Valid Date | 99.00 | 99.40 | 99.00 | 98.30 | 91.00 | 99.59 |
| printCalendar | 98.61 | 61.60 | 98.80 | 97.00 | 74.51 | 99.88 |
| Cal Day | 98.00 | 65.00 | 96.90 | 97.00 | 69.00 | 99.28 |

## 5 Conclusion

Automatic software test data generation with optimal branch coverage and fault discovery rate is an NP-complete problem. Concerning this issue, there are advantages and disadvantages to any heuristic-based test generation approach. The OOA technique was developed in this work to produce software test data automatically. Four criteria were utilized to assess the results: average branch coverage, average success rate, average execution time, and average fault discovery rate. Standard programs and fault injection tools like Pittest, Muclipse, Mujava, Jester, and Jumble have been used to evaluate the method. According to the results, the suggested method outperforms existing algorithms in terms of average coverage, success rate, average execution time, and fault detection capability. About 89% of the injected faults by different tools in the benchmark programs have been discovered by the OOA. The proposed method provides 99.92% coverage and 99.20% success rate. In the proposed method, all instructions are considered for coverage regardless of their sensitivity and error propagation; an instruction classification can cover only the error-prone instructions instead of all. Finding error-prone instructions for the input programs before test generation is one of the challenges for future studies. The initial population of the proposed OOA was

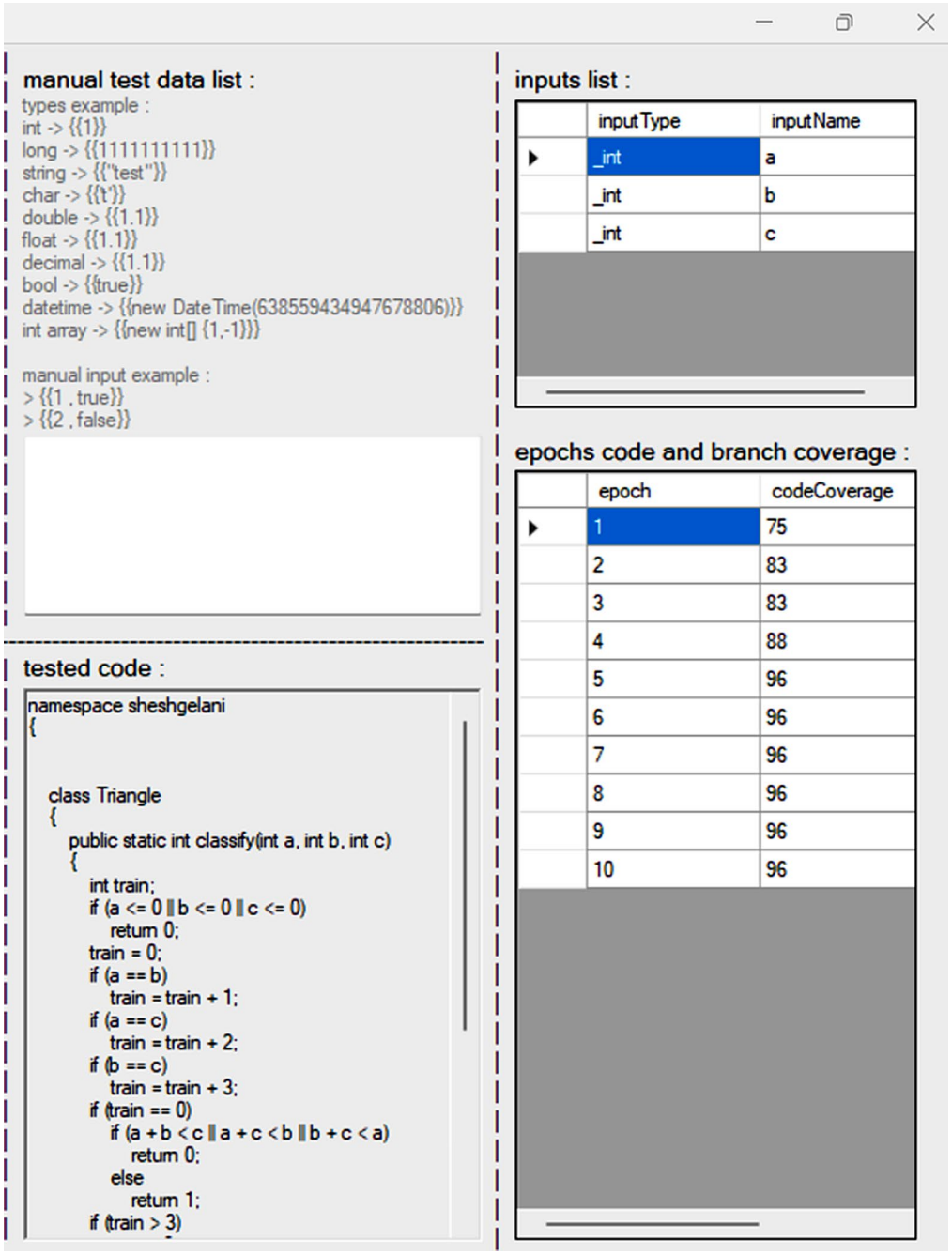



**Fig. 14** One of the interfaces of the test generation tool developed in this study

generated randomly, and the Chaose theory can be used to improve the performance of the suggested OOA in the test generation problem [23]. Considering the hard-to-find faults in the fitness function is suggested as another future study. Other hybrid evolutionary algorithms and their combinations suggested in [21, 22, 24] can be employed as a research direction to address the issue of automatically generating test data for software.

**Author Contributions** The proposed method was developed by B. Arasteh. The designed algorithm was implemented and coded by B. Arasteh and L. Zheng. The benchmark programs and related data was implemented by B. Arasteh and L. Zheng. The experimets and results analysis were performed by B. Arasteh, L. Zheng. The manuscripting was made by B. Arasteh, M. Nazeri, and A. V. Abania.

**Data Availability** Both the code used in the current study and the datasets created throughout the investigation are publicly available on Google Drive.
https://drive.google.com/drive/folders/1W7umJkrmlZkAoK_2yLbTGyFi3cYO_DKL?usp=drive_link.

## Declarations

**Ethical and Informed Consent for Data Used** The data used in this research does not belong to any other person or third party and was prepared and generated by the researchers themselves during the research. The data of this research will be accessible to other researchers.

**Competing Interests** The authors affirm that they did not receive any grants, funding while they were preparing this paper. There are no financial or non-financial conflicts of interests that the authors need to disclose.

## References

1. Ammann P, Offutt J' (2017) Introduction to Software Testing', Cambridge University Press, ISBN 978-1-107-17201-2
2. Lin JC, Yeh PL (2001) Automatic Test Data Generation for path testing using GAs. J Inform Sci 131(1):47–64
3. Khatun S, Rabbi KF, Yaakub CY, Klaib MFJ (2011) A Random search based effective algorithm for pairwise test data generation. Int Conf Electr Control Comput Eng 2011 (InECCE) 293–297. https://doi.org/10.1109/INECCE.2011.5953894
4. Marcelo M, Eler AT, Endo, Vinicius HS, Durelli, An empirical study to quantify the characteristics of Java programs that may influence symbolic execution from a unit testing perspective, (2016) Cristian C., Koushik S. S., Symbolic Execution For Software Testing: Three Decades Later, *Communications of the ACM*, Vol. 56 No. 2, Pages 82–90. 2013
5. Cristian C, Koushik SS (2013) Symbolic execution for software testing: three decades later. Communications of the ACM 56(2):82–90. 2013
6. Cohen MB, Colbourn CJ, Ling ACH (2003) Augmenting simulated annealing to build interaction test suites, In: Proceedings of the Fourteenth International Symposium on Software Reliability Engineering (ISSRE'03), pp. 394–405
7. Sharma C, Sabharwal S, Sibal R (2014) A Survey on Software Testing techniques using genetic algorithm. Int J Comput Sci 10(1):381–393
8. Esnaashari M, Damia AH (2021) Automation of software test data generation using genetic algorithm and reinforcement learning. Expert Syst Appl 183:115446
9. Mao C (2014) Generating Test Data for Software Structural Testing based on particle swarm optimization. Arab J Sci Eng 39(6):4593–4607
10. Kaur A, Bhatt D (2011) Hybrid particle swarm optimization for regression testing. Int J Comput Sci Eng Vol 3(5):1815–1824
11. Ahmed BS, Zamli KZ (2011) A variable strength interaction test suites generation strategy using particle swarm optimization. J Syst Softw 84:2171–2185
12. Ghaemi A, Arasteh B (2020) SFLA-based heuristic method to generate software structural test data. J Softw Evol, 32, 1
13. Aghdam ZK, Arasteh B (2017) An efficient method to Generate Test Data for Software Structural Testing using Artificial Bee colony optimization algorithm. Int J Softw Eng Knowl Eng 27(6):951–966
14. Mao C, Xiao L, Yu X, Chen J (2015) Adapting ant colony optimization to Generate Test Data for Software Structural Testing. J Swarm Evolutionary Comput 20:23–36
15. Arasteh B, Hosseini SMJ, Traxtor (2022) An Automatic Software Test Suit Generation Method inspired by Imperialist competitive optimization algorithms. J Electron Test 38:205–215. https://doi.org/10.1007/s10836-022-05999-9
16. Martou P, Mens K, Duhoux B, Legay A (2023) Test scenario generation for feature-based context-oriented software systems. J Syst Softw 197:111570. https://doi.org/10.1016/j.jss.2022.111570
17. Sulaiman RA, Jawawi DN, Halim SA (2023) Cost-effective test case generation with the hyper-heuristic for software product line testing. Adv Eng Softw 175:103335. https://doi.org/10.1016/j.advengsoft.2022.103335
18. Arasteh B, Sadegi R, Arasteh K, Gunes P, Kiani F, Torkamanian-Afshar M (2023) A bioinspired discrete heuristic algorithm to generate the effective structural model of a program source code. J King Saud Univ - Comput Inform Sci 35(8):1319–1578. https://doi.org/10.1016/j.jksuci.2023.101655
19. Hosseini MJ, Arasteh B, Isazadeh A, Mohsenzadeh M, Mirzarezaee M (2020) An error-propagation aware method to reduce the software mutation cost using genetic algorithm. Data Technol Appl 55(1):118–148. https://doi.org/10.1108/DTA-03-2020-0073
20. Shomali N, Arasteh B (2020) Mutation reduction in software mutation testing using firefly optimization algorithm. Data Technol Appl 54(4):461480. https://doi.org/10.1108/DTA-08-2019-0140
21. Gharehchopogh FS, Abdollahzadeh B, Arasteh B (2023) An Improved Farmland Fertility Algorithm with Hyper-Heuristic Approach for solving travelling salesman problem. CMES-Computer Model Eng Sci 135(3):1981–2006. https://doi.org/10.32604/cmes.2023.024172
22. Arasteh B, Sadegi R, Arasteh K (2021) Bölen: software module clustering method using the combination of shuffled frog leaping and genetic algorithm. Data Technol Appl 55(2):251–279. https://doi.org/10.1108/DTA-08-2019-0138
23. Arasteh B (2022) Clustered design-model generation from a program source code using chaos-based metaheuristic algorithms. Neural Comput Applic. https://doi.org/10.1007/s00521-022-07781-6
24. Arasteh B, Abdi M, Bouyer A (2022) Program source code comprehension by module clustering using combination of discretized gray wolf and genetic algorithms. Adv Eng Softw 173:0965–9978. https://doi.org/10.1016/j.advengsoft.2022.103252



**Leiqing Zheng** is a faculty member at Rizhao Polytechnic university. His research interest includes fault-tolerant systems, software development and AI.

**Bahman Arasteh** was born in Tabriz. He received the master's degree in software engineering from IAU Arak branch, and the Ph.D. degree in software engineering from IAU, Tehran Science and Research Branch, respectively. Currently, he is an associate professor at Istinye University, Istanbul, Türkiye. He has published more than 60 papers in refereed international journals and conferences. He is the coordinating editor of the Springer Journal of Electronic Test and the journal Assurance Engineering and Management. He is the reviewer of different international journals in Elsevier, Springer, Wiley, and Hindawi. His research interests include search-based software engineering, Software testing, artificial intelligence and optimization algorithms, software fault tolerance, and software security.

**Mahsa Nazeri Mehrabani** received a master's degree in software engineering from Amirkabir university of technology. She is a researcher in the department of computer engineering at Tabriz university. His research interest includes AI and dependable and fault-tolerant systems.

**Amir Vahide Abania** received the master's degree in software engineering from IAU Tabriz branch. His research interest includes AI and software testing.