Signal, Image and Video Processing (2024) 18:5333–5346
https://doi.org/10.1007/s11760-024-03236-8

ORIGINAL PAPER

Effective test-data generation using the modified black widow optimization algorithm

Bahman Arasteh[1] · Ali Ghaffari[1,4] · Milad Khadir[2] · Mahsa Torkamanian-Afshar[3] · Sajad Pirahesh[2]

Received: 26 July 2023 / Revised: 15 August 2023 / Accepted: 19 April 2024 / Published online: 10 May 2024

Abstract
Software testing is one of the software development activities and is used to identify and remove software bugs. Most small-sized projects may be manually tested to find and fix any bugs. In large and real-world software products, manual testing is thought to be a time and money-consuming process. Finding a minimal subset of input data in the shortest amount of time (as test data) to obtain the maximal branch coverage is an NP-complete problem in the field. Different heuristic-based methods have been used to generate test data. In this paper, for addressing and solving the test data generation problem, the black widow optimization algorithm has been used. The branch coverage criterion was used as the fitness function to optimize the generated data. The obtained experimental results on the standard benchmarks show that the proposed method generates more effective test data than the simulated annealing, genetic algorithm, ant colony optimization, particle swarm optimization, and artificial bee colony algorithms. According to the results, with 99.98% average coverage, 99.96% success rate, and 9.36 required iteration, the method was able to outperform the other methods.

Keywords Software-test generation · Black widow optimization algorithm · Branch coverage · Success rate · Stability

1 Introduction

Software testing is one of the most important stages of software quality assurance [1–4]. Software testing is the process of finding program bugs via real execution. The software test can be conducted manually or automatically. In real-world software, software testing activities might be quite time-and money-consuming. Thus, automatic testing may have several advantages over the latter approach, such as: reducing the amount of testing time, reducing the amount of testing cost, and improving the reliability of the software product [5, 6]. Automatically generating effective test data is one of the research fields in software engineering. The research issue that was addressed in this study is automatically generating effective test data for a unit of source code. In this study, branch coverage is considered as a criterion to evaluate the effectiveness of the generated test data. Finding a minimal subset of input data in the shortest amount of time (as test data) to obtain the maximal branch coverage is an NP-complete optimization problem in the field.

For generating test data with the greatest amount of branch coverage, several heuristic and evolutionary strategies have been introduced [7]. The difficulty in covering all program branches, the low average success rate in providing data with maximum coverage in multiple executions, the inability to provide consistent results in various executions, and the longer execution time are considered as the disadvantages of the prior techniques. The present study's objectives are as follows:

- Generation of test data with the most branch coverage
- Increasing the success rate in test-data generation with the maximum coverage
- Decreasing the required time of test data generation with maximum coverage

✉ Bahman Arasteh
Bahman.arasteh@istinye.edu.tr

1 Department of Software Engineering, Faculty of Engineering and Natural Science, Istinye University, Istanbul, Turkey

2 Department of Computer Engineering, Islamic Azad University, Tabriz Branch, Tabriz, Iran

3 Computer Engineering Department, Faculty of Engineering, Istanbul Topkapi University, 34087 Istanbul, Turkey

4 Department of Computer Science, Khazar University, Baku, Azerbaijan

In this research, we suggested a method for the automatic production of test data with the greatest branch coverage and minimal time by modifying and using the black widow optimization algorithm (BWOA). It should be emphasized that BWOA performs more effectively than similar swarm-based heuristic algorithms in software test generation. The following are the main contributions of the current paper:

- Modifying and applying BWOA for generating optimal test data
- Generating a maximum-coverage test suit with a minimum amount of time
- Implementing and providing an open-source tool for automatically generating coverage-based test data for a program source code.
- Generating consistent results with a low standard deviation

The remaining sections are arranged as follows: Sect. 2 provides a brief overview of relevant research. The suggested method is described in Sect. 3. The suggested method's results and their comparison to other relevant methods are the topics of Sect. 4. Section 5 provides the study's findings as well as recommendations for more research.

2 Related studies

Researchers [8] suggested an automatic method to test data generation using a random approach. The primary limitations of this method are thought to be the extremely high time needed for reaching the necessary coverage and the creation of repeating data. Additionally, this method did not yield the desired results in terms of the number of bugs found. Therefore, to get better outcomes, researchers presented a strategy based on symbolic execution [9]. An efficient testing method that enables the automated generation of test data inputs that result in software defects is symbolic execution. One of the key advantages of symbolic execution is the production of concrete test inputs, which have great coverage. Specific and high-coverage test data cause the issue in terms of bug finding. Array value and pointer inputs cannot be determined via symbolic techniques. To generate effective test data (high coverage), the simulated annealing technique was used [10]. The simulated annealing approach is used to produce optimal test data by transforming the test data generation problem into an optimization problem. The primary drawbacks of this approach are its poor performance, insufficient coverage, and local optimum placement. For behavioral testing, this approach is suitable.

A technique for generating test data was put out in [11] by utilizing genetic algorithms (GA). In this technique, the best test paths were chosen using GA. This study's fitness function called the similarity function, sought to quantify how closely the covered path resembled the intended one. Path optimality implies that the path was used when running the test data. In other words, a path's optimality increases with increasing path follow-up. The time needed to discover the best path is reduced as a result of the use of GA. Researchers used a parallel implementation of GA technique to increase efficiency and efficacy. After that, six benchmark programs were used to examine the suggested method's coverage. The results showed that test data output has improved.

One of the main issues with GA is that the chromosomes don't strive to get better on their own and can only get better through mutation. A portion of a chromosome cannot be evaluated in GA; only the full chromosome is taken into account by the fitness function. Thus, GA is similar to blind search algorithms. The GA with reinforcement learning as a memetic search strategy has been proposed [12] as an automated test-data creation method. This enhanced GA concentrates on the population's best chromosomes, and Q-learning has been employed to direct the search process. When a chromosome has duplicated parts, the mutation operator is used in the procedure. Experimental results show that in terms of success rate and branch coverage, this hybrid method works better than pure GA. Particle swarm optimization (PSO) was used by researchers in [13] to provide test data. The PSO technique was used with different fitness functions to generate the test data because of its ease of use and quick convergence rate.

A technique using the ant colony optimization (ACO) algorithm was proposed by [14] to provide the best test data to maximize branch coverage. This technique defined a special fitness function depending on coverage. The results of the studies demonstrated that this approach had larger coverage, faster convergence, and more stable outcomes. The performance of the artificial bee colony (ABC), genetic, simulated annealing, ant colony (ACO), and particle swarm algorithms in terms of creating usable test data is compared by the researchers [15]. This study uses a distance function using the branch coverage (as the fitness function). Results of trials indicated that, for this issue, the ABC algorithm's coverage, success rate, and convergence speed are, respectively, 99.94%, 99.92%, and 3.36 iterations. Overall, the ABC algorithm generated the best test data compared to the other algorithms. In [16], the shuffling frog leaping algorithm (SFLA) was used to provide a way for automatically creating test data. This method's fitness function included branch coverage. It has been evaluated using the seven prescribed benchmarks. In comparison to prior evolutionary and heuristic algorithms like GA, PSO, ACO, and ABC, the findings showed that this technique had several benefits. The SFLA-based method can generate test data with the fewest duplicates and 99.99% branch coverage. Additionally, with a 99.97% success rate, it produces the best test results.

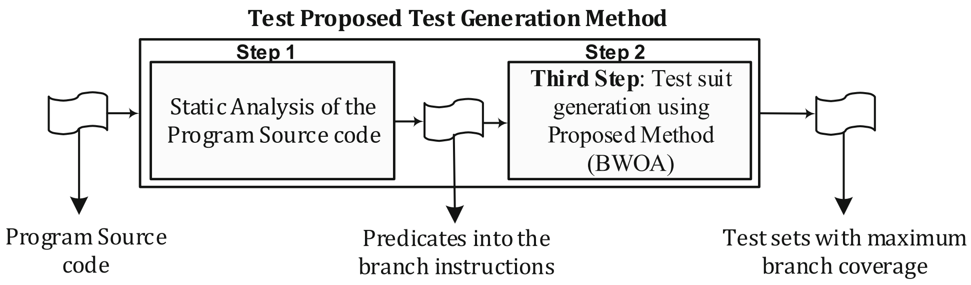

Test Proposed Test Generation Method
Step 1
Static Analysis of the Program Source code
Step 2
Third Step: Test suit generation using Proposed Method (BWOA)
Program Source code
Predicates into the branch instructions
Test sets with maximum branch coverage

Fig. 1 The workflow of the proposed test generation method

Table 1 Different test generation methods based on heuristic algorithms

Method	Advantages	Disadvantages
Simulated annealing algorithm (SA) [10]	Higher speed than random search	Low success rate and local optimum
Genetic algorithm (GA) [11, 12]	Parallel implementation	Low success rate, high run-time and
Particle swarm optimization algorithm (PSO) [13]	High-speed and simple implementation	Low stability
Ant colony optimization algorithm (ACO) [14]	Considering the weights of branches	Low stability and High runtime
Artificial bee colony optimization algorithm (ABC) [15]	Appropriate branch coverage low run time	Low stability

All in all, the existing test-data-generating methods have their advantages and disadvantages which are briefly summarized in Table 1. In other words, the issue of effective test-data generation for automatic software tests has not yet been entirely resolved. Consequently, an automated test-data generation method utilizing the modified and customized version of the BWOA optimization algorithm is proposed in this research work. The authors try to fill the gap in earlier studies. In Sect. 3, the suggested method is explained.

3 Proposed method

Figure 1 shows the steps involved in the method suggested for the automated creation of unit-level test data. The source code of the program is statically examined in the first stage, and the necessary structural data is extracted for the next steps. The number and data type of the program (function) inputs, the number of branch instructions, and the conditional expression for each branch are extracted in the first step. At this stage, the program's source code is statically analyzed and the branches that need to be covered are extracted. If the program has n lines of code, the time cost of this stage will be O(n). The static analysis phase is executed as the first step and the obtained results are provided for the next steps of software testing. In the second phase, the conditional expressions of the branch instructions are extracted. The third step uses BWOA to generate the best test data. The coverage of program branches was used to determine the objective function of this study. The main goal of the suggested method is to generate test data with the maximum branch coverage in the shortest amount of time.

3.1 Test data generation by BWOA

In this paper, by capitalizing on BWOA, we developed an automatic method for producing optimal test data. Algorithm 1 indicates the pseudo code of the developed BWOA. It should be noted that BWOA was inspired by black widow spiders' unique mating behavior [17]. The way of producing automatic test data by BWOA is described in this section. This algorithm operates based on the initial random population of the spiders.

Algorithm 1. The pseudo code of the developed BWO to generate software test data

```
function BWO
    % Input: Predicates of the branch instructions of the input program
    % Output: bestSolution and bestFitness
    % Initialize parameters
    Intializing the maxIterations;
    populationSize is set based on the number of branch instructions;
    lb = BWO lowerBound;
    ub = BWO upperBound;

    % Initialize the population
    population = lb + (ub - lb) * rand(populationSize, dimension);

    % Main optimization loop
    for iter = 1 to maxIterations
        % Evaluate the fitness of each solution in the population using Equation 2;
        fitness = evaluateFitness(population);
        % Update the best solution and fitness
        bestFitness = min(fitness);
        bestSolution = Best individual of the current population;
        Update the population;
        plotProgress(iter, bestFitness);
        % Check stopping criteria
        if bestFitness < parameters.threshold
            break;
        end
    end
end
```

Each member of the initial population (one piece of random test data) is a spider. In the beginning, BWOA starts with the initial spider population. That is, each spider indicates test data. The male spiders are eaten by the black widow after mating. She then transports the sperms that were previously kept in her own sperm cavities and releases them into the egg sacs. Spiders emerge from their eggs after 11 days. They spend a few days to a week living on the mother net. Cannibalism is seen among the population of newborn siblings during this period. The phases of BWOA are shown in Fig. 2.

Initial population: each spider is denoted by $1*N_{var}$ array as a test data. The structure of each spider is shown in Fig. 3. For starting the algorithm, a widow matrix in the $N_{pop}*N_{var}$ size with the initial population of spiders is produced. Next, the pair of parents are randomly selected so that they do the mating operation. As mentioned above, the black widower is eaten during or after mating.

Reproduction since spider pairs are independent of each other, almost 1000 eggs are produced in each mating in real life. However, those baby spiders survive and are stronger. This method includes an exclusive stage, i.e. cannibalism. In this stage, weak members of the population are removed using fitness. This stage leads to the initial convergence. An array, namely alpha should be created for reproduction. The initial value of the widow array is determined using random numbers. Next, children are produced using α in Eq. (1) where x_1 and x_2 indicate the parents and y_1 and y_2 refer to the children. Finally, children and mothers are added to a matrix and are ranked based on their fitness. The best individuals are added to the new population. This procedure is applied to all the pairs.

$$\left\{\left\{\begin{array}{l} y_1 = \alpha \times x_1 + (1-\alpha) \times x_2 \\ y_2 = \alpha \times x_2 + (1-\alpha) \times x_1 \end{array}\right\}\right. \quad (1)$$

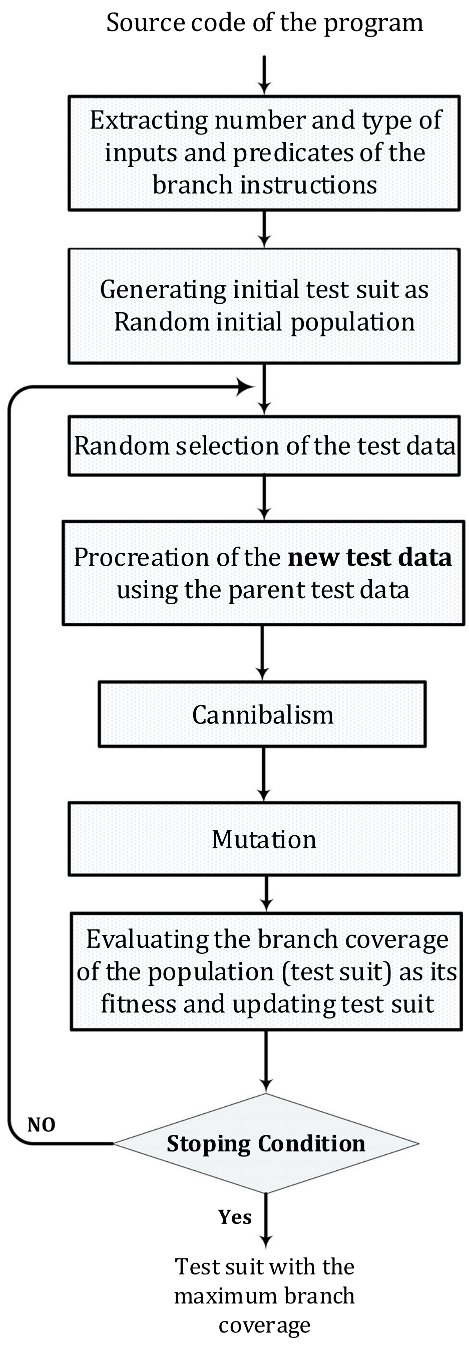
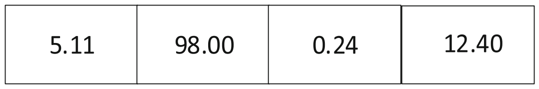

Source code of the program
Extracting number and type of inputs and predicates of the branch instructions
Generating initial test suit as Random initial population
Random selection of the test data
Procreation of the **new test data** using the parent test data
Cannibalism
Mutation
Evaluating the branch coverage of the population (test suit) as its fitness and updating test suit
Stoping Condition
NO
Yes
Test suit with the maximum branch coverage

Fig. 2 Process of BWOA for test data generation

Cannibalism and mutation there are three types of cannibalism BWOA. The first type is sexual cannibalism in which the black widow (a member with higher fitness) eats its male pair. In this type of cannibalism, male and female spiders can be distinguished from each other using the fitness function. The second type of cannibalism is between siblings in which stronger spiders eat their own weaker siblings. Each spider indicates produced test data. In this type of cannibalism, a



5.11	98.00	0.24	12.40

Fig. 3 An instance of black widow indicating test data for a program with 4 decimal inputs

CR (cannibalism rating) is devised based on which the number of survivors can be determined. In this stage, a certain number of individuals are randomly selected from the population. As depicted in Fig. 4, each of the generated test data randomly exchanges two elements in the array. The value of Mutepop specifies the mutation rate. Attaining the state of convergence and stable conditions without any variations is tantamount to achieving the state in which the algorithm has generated data with top fitness (maximum coverage of program branches).

3.2 Objective function

The objective of this study is to identify a small sample of test data that has the greatest amount of branch coverage. For comparing them to one another and choosing the best test results, an objective function is constructed. The problem's nature determines which function should be used. In other words, it can be maintained that selecting an appropriate objective function (Fitness function) is one of the critical stages in optimization. The fitness function affects the operation of the heuristic algorithm in the search space. In the field of creating test data, branch weight is seen to be one of the most useful factors. This function displays the weight for a branch predicate after the input values have been allocated to the variables. Similarly, to that, this function was applied in this paper to navigate the issue with the test data generation process. The input program for this function contains S branches (conditional and loop instructions). The $\mathrm{bch_i}$ denotes the ith branch instruction. If the number of inputs is equal to m, each input will be determined by the variable $\mathrm{X_k \varepsilon TS}$ $(1 \leq \mathrm{K} \leq \mathrm{m})$. Equation (2) was used to compute the fitness of test data generated by BWOA in each iteration.

$$\mathrm{Fitness}(\mathrm{X_k}) = \frac{1}{\left[\varnothing + \sum_{i=1}^{s} w_i . f(\mathrm{bch}_i, \mathrm{X_k})\right]^2} \tag{2}$$

Ø is a constant that is set at the runtime according to the program. The value of this parameter was 0.01 in this study. The variable w indicates the weight of program branches. f function, as the distance function, is computed via Eq. (3) for a test dataset. The fitness (degree of coverage) of the test

Table 2 Distance function of the predicates into the branch instructions

No	Input predicate	Value of distance function ($f(bch_i)$) for the input predicate
1	Boolean	If the value of Boolean predicate is true then 0 else k
2	~ a	Negation is propagated over a
3	a = b	If the value of abs(a − b) = 0 then 0 else abs(a − b) + k
4	a ≠ b	If the value of abs(a − b) = 0 then 0 else k
5	a < b	If the value of b is bigger than a then 0 else abs(a − b) + k
6	a ≤ b	If the value of b is bigger or equal to a then 0 else abs(a − b) + k
7	a > b	If the value of a is bigger than b then 0 else abs(b − a) + k
8	a ≥ b	If the value of a is bigger or equal to b then 0 else abs(b − a) + k
9	a and b	f (a) + f (b)
10	a or b	min(f (a), f (b))

dataset generated by BWOA is determined by Eq. (3).

$$Fitness\,(TS) = 1/\left[\emptyset + \sum_{i=1}^{s} w_{i}.\min\{f(bch_i.X_k)\}_{k=1}^{m}\right]^2 \tag{3}$$

Equation (3) will be assessed as 1/∅ if the resulting test dataset (TS) can cover all of the branches. The maximum fitness value is as a result attained. The suggested method considers data with the highest fitness function value to be more effective data. Equation (3) states that the distance function is used to calculate the fitness function. When values are supplied to the variable inputs, the distance function provides the presumable standard deviation for a conditional instruction (if or while instructions). For the expressions inside the program's conditional instructions, the distance function is calculated following Table 2. If the conditional expression in Table 2 is an integer based on the produced inputs, the distance function's value will be zero; otherwise, the value of the δ variable will be added to the conditional expression's value. The value of the δ variable in this investigation was 0.1 [13]. Branch weight is applied for measuring the reachability degree of program branches (conditional commands). If the branch weight for a given conditional command is high, its reachability (running probability) will be low; as a result, the test data-generating algorithm should make more endeavors for accessing that branch. The effective factors on branch weights are as follows:

- The branch's nesting weight

Table 3 Weights of the available operators in the conditional predicates

Operator	Weight
= =	0.9
≥ , > , ≤ , <	0.6
Boolean	0.5
! =	0.2

- Branch's predicate weight (complexity)

Equation (4) is used to calculate the nesting weight of a branch instruction. It shows what level a certain branch is at. Access to that branch will be more challenging given the increased nesting level. The variable i in Eq. 4 stands for the program's lowest nesting level, $\mathrm{nl}_{\mathrm{min}}$, and the highest nesting level, $\mathrm{nl}_{\mathrm{max}}$. Similar to Eqs. (3) and (5) is used to normalize the branch's nesting weight. The entire weight of the branches is divided by the weight of the ith branch in this equation.

$$wn(bch_i) = \frac{nl_i - nl_{min} + 1}{nl_{max} - nl_{min} + 1} \tag{4}$$

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

The predicate weight reveals the degree of complexity of the predicates inside the branches. If the input data show that these predicates have true values, then they will be covered. Predicate weight is calculated using Eq. (6) and Table 3. Two possible modes in this equation are as follows:

- If the branch consists of h condition expressions that have been connected by the *and* operator, the predicate's overall weight is equal to the second root of the sum of the predicates.
- If the relevant branch includes h conditions that have been connected using the *or* operator, the condition predicates with the lowest weight amount will be picked.

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

In Eq. (6), the variable $(1 \le i \le s)$bch$_i$ stands for the ith branch. For calculating the predicate weight (complexity) for the ith branch, h variable is used which refers to the number of predicates in the branch. c_j, $(1 \le j \le h)$ denotes the ith condition and $\mathrm{W_r}$ indicates the weight of the condition whose value is set according to Table 3. Also, Eq. (7) is used to

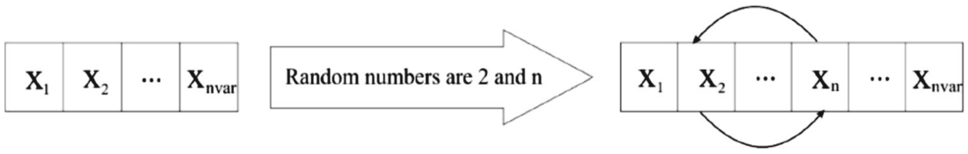


Fig. 4 Mutation operator on the population elements

normalize the predicates' weight.

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

The weight of each branch was finally determined using Eq. (8). It implies that the predicate weight of the branch as well as its nested weight are included in each branch's weight. The weight of the ith branch is indicated by Wi. Predicate weight and nested weight were both identical in this study, and the value of α was 0.5.

$$w_i = \propto .wn'(bch_i) + (1- \propto).wp'(bch_i) \tag{8}$$

4 Results

4.1 Implementation platform

The outcomes of putting the suggested strategy into practice are reported, assessed, and debated in this section. The MATLAB platform was used to put the suggested technique into practice. It is true that this software is extensively applicable and offers the chance for many computational operations, demonstration operations, and programming. Here, the results obtained from implementing and executing the proposed method via Mat Lab software are analyzed and discussed. MATLAB Version 2021 was used in this study. The computer system's specs are as follows, and it will be utilized to implement the suggested method: Intel corei7, 8 GB of RAM. The proposed method was implemented on Windows 10 operating system. Software test-data generation has recently made extensive use of algorithms including GA, SA, ACO, PSO and SA. The Matlab platform was used to implement the suggested method and also the methods based on GA, ACO, ABC, PSO and SA and SA. All the methods were implemented and executed on the same computer with the identical operating system and the same specifications, namely Intel corei7 and 8 GB of RAM. The criteria used to evaluate the method proposed in this study are as follows:

1. Average coverage (AC) measures how well program branches are covered by the test suit that is generated. Higher scores for this criterion denote greater performance.
2. The average number of iterations required by a specific technique to cover all program branches is represented by the average generation (AG) criteria. A certain strategy performs better when it receives lower scores for this criterion.
3. The average time (AT) criterion describes the amount of time needed to cover all program branches. Milliseconds are used to measure this criterion (ms). Lower values for this criterion indicate the greater performance of the specified approach.
4. Success rate (SR) measures the chance that the test results will cover every program branch. Greater performance is indicated by higher scores on this criterion.

The number of iterations is taken into consideration as the termination condition in all algorithms to generate test data. A maximum of 300 iterations of the algorithms were performed for each execution. Additionally, each algorithm was run 10 times (each execution includes 300 iterations). The average results and standard deviations were then calculated and compared following 10 executions.

4.2 Benchmarks

In this study, 6 programs with differing degrees of complexity, which have been used in related works, were taken into consideration. Table 4 lists the features of these six benchmark programs. The benchmark programs' source code is written in the C++ programming language. The programs' test datasets were created using the GA, SA, ABC, ACO, PSO, and GA algorithms. The parameters of the implemented optimization algorithms were shown in Table 5.

4.3 Results evaluation

A comprehensive set of experiments were carried out on the implemented techniques once the suggested approach and the other algorithms (GA, SA, PSO, ACO, ABC). Six benchmark programs that were presented and discussed in Table 3 were used in the trials to generate test data for each approach. The benchmark program included a variety of programming constructs, including loops, conditional structures, and several types of data. In earlier research, these programs were heavily utilized. The evaluation criteria mentioned in sub-Sect. 4.2 are AC, AG, SR, and AT. The average coverage (AC) of branches by the generated test data is regarded as the

Table 4 The features of the benchmarks used to evaluate the test generating methods

Programs	#Args	Args. type	LOC	Specification
TriangleType	3	Integer	31	This program classifies a three-integer input integer as a type of triangle
valDay	3	Integer	72	This software determines the weekday
IsValidDate	3	Integer	41	This program Checks an input date is valid or not
cal	6	Integer	26	This program takes two dates and compute the days between them
remainder	2	Integer	17	This program calculates the remainder of two integer division
printCalender	2	Integer	124	This program prints calendar according to the inputs

Table 5 Calibration parameters of different replication algorithms that have been adjusted experimentally

Algorithms	Parameters	Value
Genetic algorithm(GA)	Number of chromosomes	Depend on the program Line of code
	Length of chromosome	Number of program inputs
	Crossover rate	0.7
	Mutation rate	0.05
	s	100
Ant colony optimization (ACO)	Number of ants	Number of program inputs
	Initial pheromone (τ)	1
	Q parameter	1
	Pheromone power weight (α)	1
	Evaporation rate (ρ)	0.05
	Number of iterations	100
Artificial bee colony algorithm (ABC)	Number of bee	Depends on the program line of code
	Number of food source	Number of Bee/2
	Limit parameter	15
	Number of iterations	100
Simulated annealing algorithm (SA)	Colony size	Depends on the program lines of code
	Start temperature	1
	Linear cooling parameter	0.8
	Number of iterations of temperature	50
Particle swarm optimization algorithm (PSO)	Number of particles	Depends on the program lines of code
	Inertia weight	0.8
	Inertia weight damping ratio	0.99
	Particle.C1 and particle.C2	1.8
	Number of iterations	100
Black widow optimization algorithm (BWO)	Colony size	Depends on the program line of code
	Procreate rate	0.6
	Cannibalism rate	0.4
	Mutation rate	0.4
	Number of iterations	100

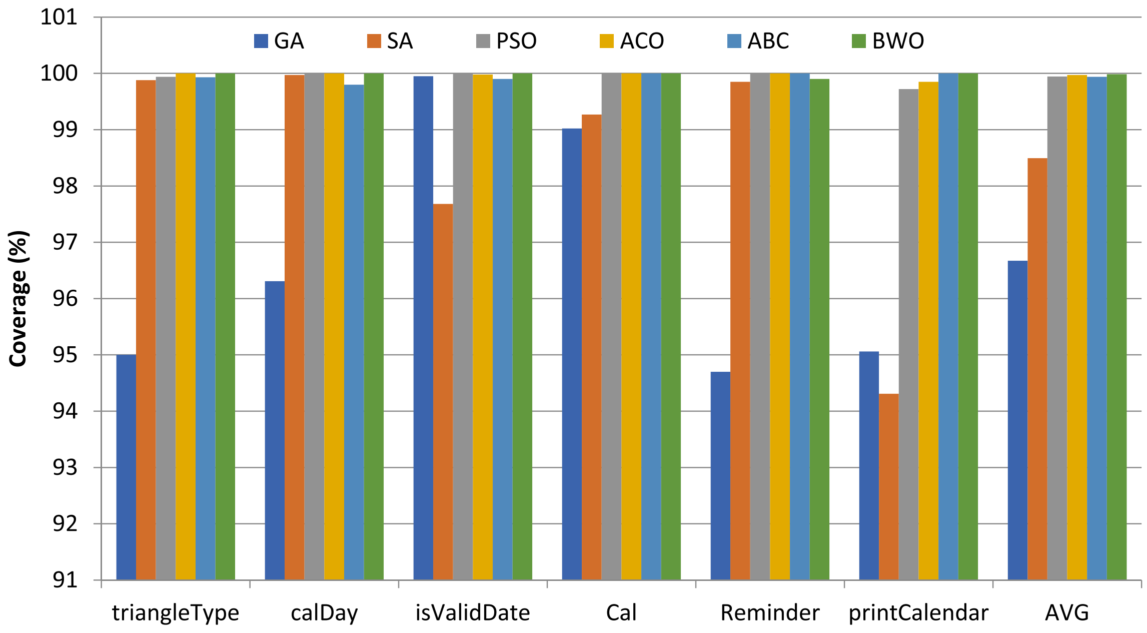


Fig. 5 AC of the test suit generated for each program by test-generating methods

Table 6 Average branch coverage of the test suit generated by different methods

Criteria	GA	SA	PSO	ACO	ABC	Proposed method (BWO)
Average coverage (AC %)	96.67	98.49	99.94	99.97	99.93	99.98

first and most crucial requirement. Each test data-generating technique for each benchmark program was run ten times to calculate the average branch coverage. Then, the average value of the results was taken into consideration. The average branch coverage by the generated test data using various approaches is shown in Fig. 5. The test data generated using the suggested method in all programs except the *remainder* program reached 100% coverage, according to the findings attained. Test data with a 99.90% coverage was produced for the *remainder* program. Table 6 gives the average coverage of the test data generated by different methods. As given in Table 6, the proposed method was able to generate test data with an average coverage of 99.98%.

Another factor used to assess how well the suggested strategy performed was the success rate. It shows the extent to which the suggested strategy was successful in producing test data that completely covered all of the program branches. Another series of trials were run as part of this investigation. In other words, each test data generation technique for each benchmark program was used ten times. The success rate of the method is defined as the average number of times the generated data reaches 100% coverage. The success rates of several test data generation techniques are shown in Fig. 6. The suggested technique outperformed the other methods on the benchmark programs concerning this benchmark, according to the experiment's findings. Table 7 gives the average success rate for different methods. On average, the proposed method, with 99.96% probability, was able to generate test data with 100% coverage.

Convergence speed is another criterion that was used for evaluating and investigating the proposed method. This criterion focuses on the time and expense of producing test data with the greatest coverage. The number of iterations necessary for a specific technique to provide test data with the greatest amount of branch coverage is known as convergence speed. Higher convergence speed is indicated by fewer repetitions. A technique that uses fewer repetitions produces test data more quickly. Figure 7 shows the typical number of iterations needed across 10 distinct executions to provide the best test results (maximum coverage). The results reveal that, after ABC algorithm, has the lowest number of iterations. Hence, it can be maintained that it has the highest speed in test data production. The average convergence speed of different test-generating methods is shown in Table 8.

Stability is another evaluation criterion that was used concerning heuristic algorithms. A method based on heuristic algorithms may produce efficient test data in one execution; however, it may have poor data production in its next executions. Thus, in this study, we executed test data production methods 100 times and computed their average results along with standard deviation. Figure 8 shows the stability of different methods in producing test data in *TriangleType* program

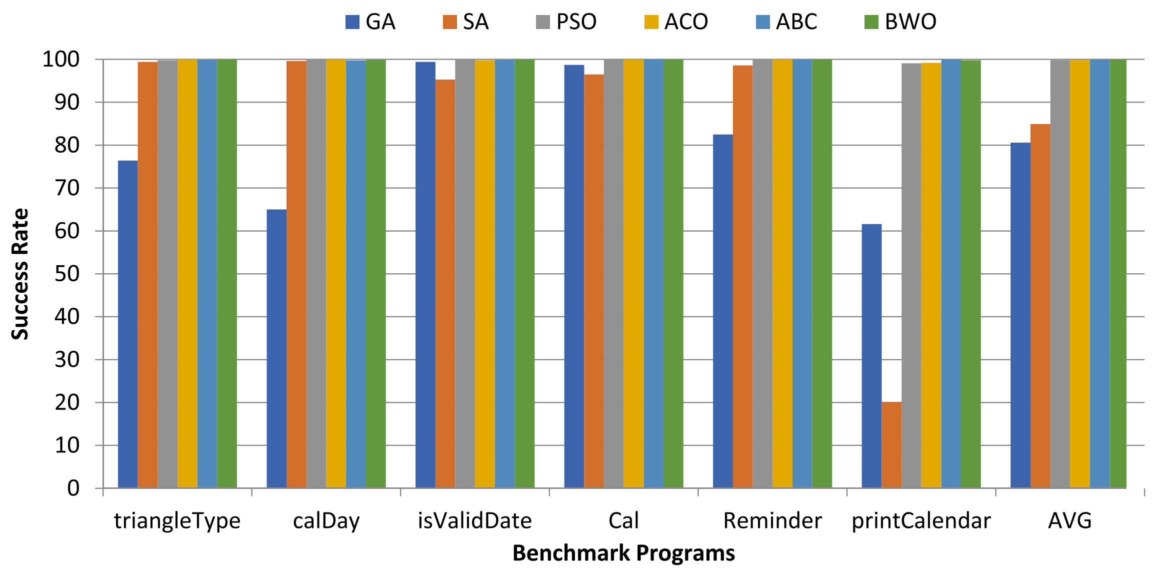


Fig. 6 Success Rate of different algorithms in producing the test suits which have 100% coverage

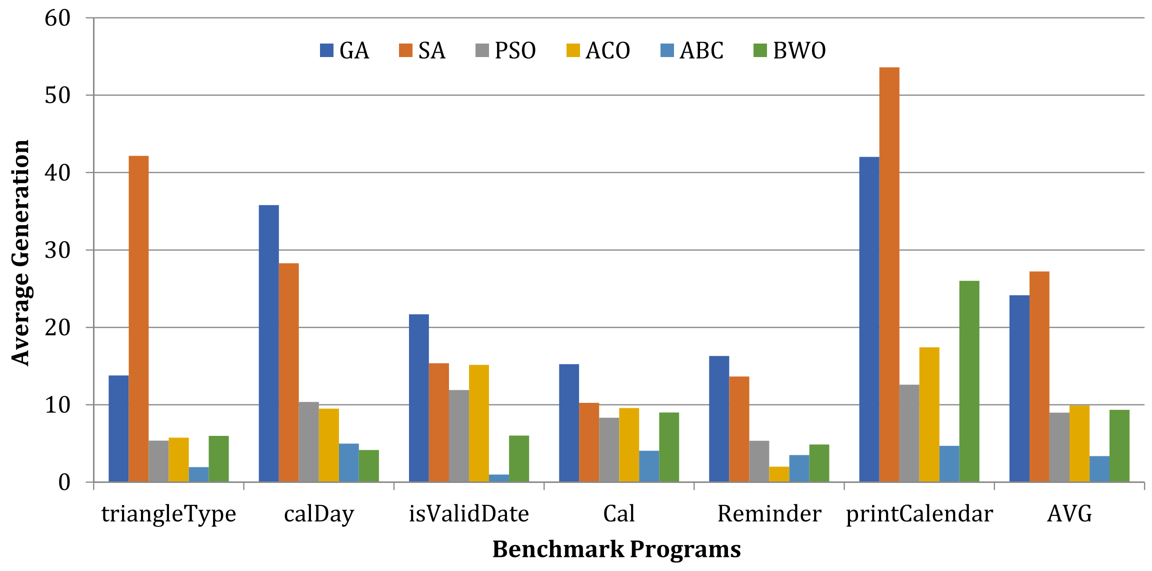


Fig. 7 The number of required generations (convergence speed) by each evolutionary method to generate maximal-coverage test data

Table 7 Average success rate of different algorithms

Criteria	GA	SA	PSO	ACO	ABC	Proposed method (BWO)	Best method
Success rate (SR %)	80.60	97.88	99.81	99.83	99.92	99.96	BWO

Table 8 Iterations required by different algorithms to reach maximum branch coverage

Criteria	GA	SA	PSO	ACO	ABC	The proposed method (BWO)	Best method
Average generation (AG %)	24.14	27.22	8.98	9.90	3.36	9.35	ABC

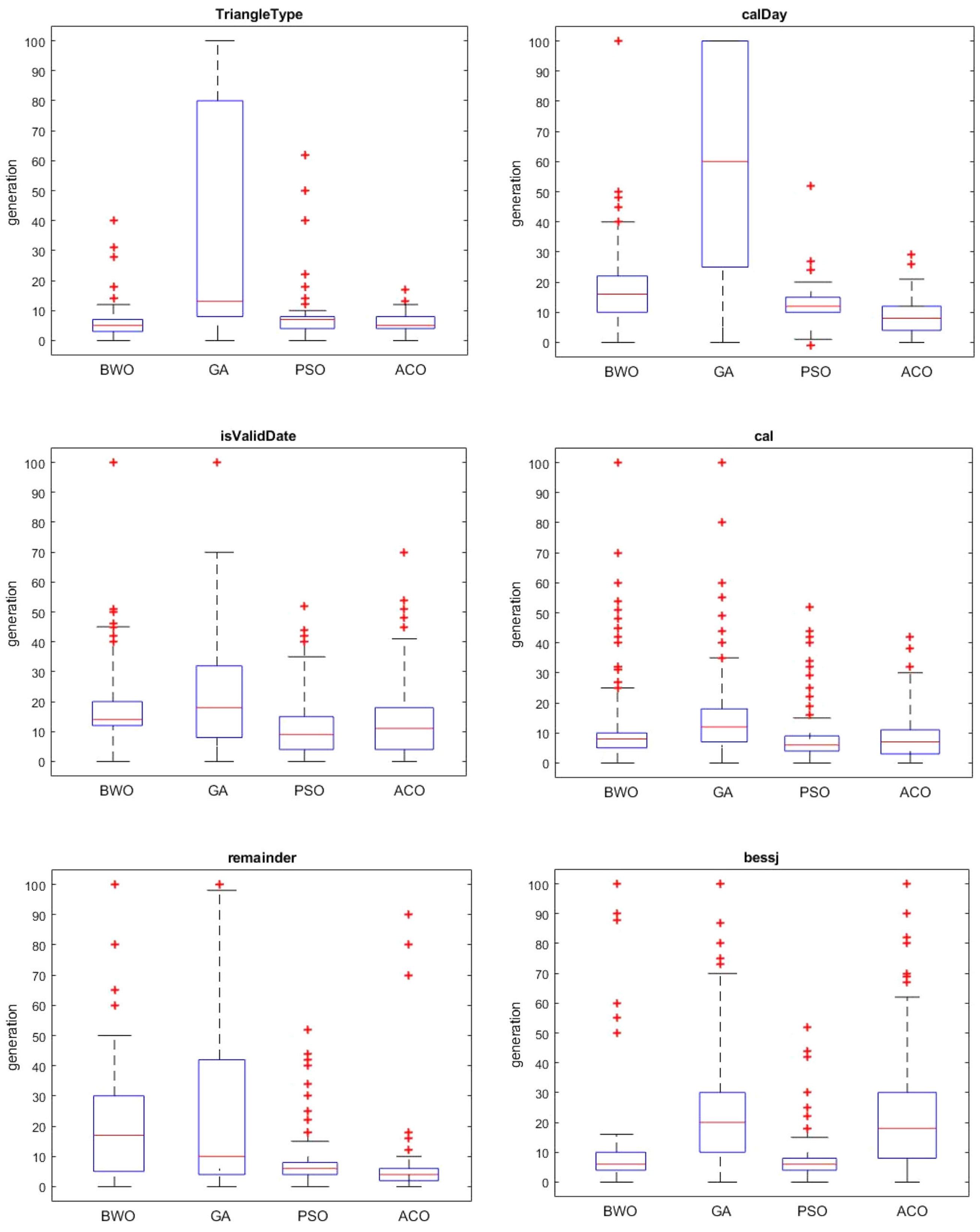


Fig. 8 The stability of different test generation method in different benchmark program

Table 9 The automatically generated test suit for *remainder* program that calculates the remainder of dividing two integers input

#Test data	Input1	Input2
1	82	20
2	33	7
3	94	4
4	64	51
5	93	99
6	7	83
7	10	22
8	8	64
9	64	46
10	53	62
11	80	79
12	61	51
13	21	95
14	73	66
15	76	17
16	0	36
17	67	25
18	15	9
19	60	0
20	61	15

for different methods. The convergence regarding *TriangleType* program for 4 algorithms, i.e. GA, PSO, ACO and the proposed method was examined and analyzed. As depicted in Fig. 8, the stability of the proposed method in the *TriangleType* program was better than those of other methods. Regarding *calDay* program, stability of the proposed method as well as ACO was better than other methods. Figure 8 illustrates stability of the proposed method as well as the other methods on *isValidDate* program. It was discovered from the findings that ACO and the suggested technique outperformed the other algorithms. Furthermore, the stability of the algorithms on *Cal* program is also shown. In this benchmark program, different algorithms had quite similar results. Nonetheless, the obtained results for the proposed method slightly improved the results of other methods. PSO and ACO had more stability in the *remainder* program. Moreover, in the ending part of Fig. 8, the stability of the algorithms regarding *printCalendar* program is illustrated. PSO and ACO algorithms had better results and GA had the poorest performance on this program. Table 9 shows the generated test suit for the *remainder* program which includes 2 input data. These data were generated automatically by the proposed method.

According to the obtained results for the different methods on the benchmark programs, it can be maintained that the black widow optimization algorithm (BWOA) had lower performance than ACO in some cases and programs. However, in general, BWOA was more stable than the other methods. The consideration and analysis of the convergence criterion in all 6 benchmarks revealed that the proposed method, on average, had earlier (faster) convergence in most benchmark programs. Thanks to the lower dispersion and deviation of the proposed method, it should be highlighted that it is generally more stable than the other methods. Table 10 indicates the execution time of different test generation algorithms. The proposed BWO based test data generator is superior to the other algorithms in terms of execution time. The benchmark programs selected in this study includes the real-world structures. All benchmarks include conditional structures (if and loop structures), arithmetic and logical operators. The software complexity (cyclomatic and Halstead metrics) of the selected benchmark programs is higher than the normal application programs. Furthermore, the authors have conducted other experiments on the other benchmark programs. The selected benchmark programs include different arguments with different datatypes. Different type arguments force the test data generation algorithms. Indeed, the selected benchmark programs are not simple to cover programs. Table 11 shows the average coverage of the generated test data by different algorithms for other real-world benchmark programs.

Test data generation is a discrete NP-complete problem. All the existing heuristic algorithms may have lower performance in the test generation problem. In this study, the algorithms have been selected based on their structural features that are suitable for the structure of the test generation problem. The final population of the selected algorithms includes the best generated test suite. The authors have implemented different test generation algorithms in the same software and hardware platform. The selected test generation heuristic algorithms have different features. Some of them have evolutionary features (GA and SA); the others have swarm intelligence features. Regarding the conducted results, GA, SA, PSO, ACO, ABC and BWO algorithms have higher suitability, performance, success rate and convergence than the similar algorithms. Hence, in this study, these algorithms have been selected to sort out the test generation problem. The priority of the generated test cases by a test generation algorithm is one of the important parameters that should be taken into account [6, 18]. Taking the test priority into account is considered as the one of future study.

Table 10 Average required time to generate test suit with maximum branch-coverage in different methods (Second)

Benachmark	GA	SA	PSO	ACO	ABC	BWO
triangleType	10.83	3.77	0.19	6.22	0.1758	0.1701
calDay	35.73	1.79	0.35	12.84	0.2375	0.1330
isValidDate	11.68	2.43	0.54	19.94	0.1944	0.1586
Cal	11.41	0.73	0.50	11.18	0.1367	0.1051
Reminder	6.09	1.01	0.17	10.49	0.0931	0.0827
printCalendar	35.48	35.38	1.41	96.27	0.1826	0.1839

Table 11 Average coverage of the generated test suit by different algorithms

Benachmark	GA (%)	SA (%)	PSO (%)	ACO (%)	ABC (%)	BWO (%)
IsLeapYear (1 Arg.)	37	37	42	58	60	74
CheckDivisibility (3 Args.)	100	100	100	100	100	100
aes data encryption (2 Args.)	25	39	61	67	66	67
ComplexMethod (3 Args.)	41	48	79	78	81	81
ElevatorStatus (3 Args.)	69	72	81	80	84	87

5 Conclusion

To automate software testing, a BWOA-based technique was suggested for the generation of test data. On six common benchmark programs, the suggested method was assessed. Branch coverage, average number of generations, success rate, and stability were used to evaluate the experiment's findings. The suggested technique outperformed ABC, ACO, PSO, SA, and GA in the experiments, according to the evaluation criteria. Regarding the results, the proposed method can be used as one of the successful test generation methods by researchers and software developers. High convergence, high coverage, high stability, and success rate are the main merits of the method compared to the existing methods. In addition, the use of a hybrid algorithm such as the combination of ACO and BWOA is suggested as a future study. Modifying the fitness function of this study to cover the strategic and error-prone codes of a program instead of all coverage is considered another future study. The optimization algorithms and methods suggested in [3, 19–24] can be used to generate software test data. The error-propagation rate of the program instructions can be taken into account in the future studies.

Author contributions The proposed method was developed and discretized by B. Arasteh and M. Khadir. The designed algorithm was implemented and coded by B. Arasteh, M. Khadir and A. Ghaffari The implemented method code was adapted and benchmarked by B. Arasteh and M. Khadir. The data and results analysis were performed by B. Arasteh and A. Ghaffari. The manuscript of the paper was written by B. Arasteh and Mahsa Torkamanian-Afshar and S. Pirahesh.

Data availability The data relating to the current study is available via the following link: https://drive.google.com/drive/folders/1vfB0XnplA7t771tFUz4aap_kc8j-bek1?usp=sharing.

Declarations

Conflict of interest The authors declare that no funds, grants, or other support were received during the preparation of this manuscript. The authors have no relevant financial or non-financial conflict of interest.

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.

References

1. Arasteh, B., Miremadi, S.G., Rahmani, A.M.: Developing inherently resilient software against soft-errors based on algorithm level inherent features. J. Electron. Test. **30**, 193–212 (2014)
2. Arasteh, B., Pirahesh, S., Zakeri, A., Arasteh, B.: Highly available and dependable e-learning services using grid system. Procedia Soc. Behav. Sci. **143**, 471–476 (2014). https://doi.org/10.1016/j.sbspro.2014.07.519
3. Keshtgar, S.A., Arasteh, B.B.: Enhancing software reliability against soft-error using minimum redundancy on critical data. Int. J. Comput. Netw. Inf. Secur. **9**(5), 21 (2017)
4. Pan, R., Bagherzadeh, M., Ghaleb, T.A., et al.: Test case selection and prioritization using machine learning: a systematic literature review. Empir. Softw. Eng. **27**, 29 (2022). https://doi.org/10.1007/s10664-021-10066-6
5. Khanna, M., Chaudhary, A., Toofani, A., et al.: Performance comparison of multi-objective algorithms for test case prioritization during web application testing. Arab. J. Sci. Eng. **44**, 9599–9625 (2019). https://doi.org/10.1007/s13369-019-03817-7
6. Khanna, M., Chauhan, N., Sharma, D.K., Toofani, A.: Test case prioritisation during web application testing. Int. J. Comput. Appl.

Technol. **56**(3), 230–243 (2017). https://doi.org/10.1504/IJCAT.2017.088200
7. Pradhan, S., Ray, M., Swain, S.K.: Transition coverage based test case generation from state chart diagram. J. King Saud Univ. Comput. Inf. Sci. **34**(3), 993–1002 (2022). https://doi.org/10.1016/j.jksuci.2019.05.005
8. Khatun, S., Rabbi, K., Yaakub, C., Klaib, M. J.: A Random search based effective algorithm for pairwise test data generation. In: International Conference on Electrical, Control and Computer Engineering 2011 (InECCE) (2011)
9. Eler, M.M., Endo, A.T., Durelli, V.H.: An empirical study to quantify the characteristics of Java programs that may influence symbolic execution from a unit testing perspective. J. Syst. Softw. **121**, 281–297 (2016)
10. Cohen, M. B., Colbourn, C. J., Ling, A. C.: Augmenting simulated annealing to build interaction test suites. In: 14th International Symposium on Software Reliability Engineering, 2003. ISSRE (2003)
11. Lin, J.-C., Yeh, P.-L.: Automatic test data generation for path testing using GAs. Inf. Sci. **131**(1–4), 47–64 (2001)
12. Esnaashari, M., Damia, A.H.: Automation of software test data generation using genetic algorithm and reinforcement learning. Expert Syst. Appl. **183**, 115446 (2021)
13. Mao, C.: Generating test data for software structural testing based on particle swarm optimization. Arab. J. Sci. Eng. **39**, 4593–4607 (2014)
14. Mao, C., Xiao, L., Yu, X., Chen, J.: Adapting ant colony optimization to generate test data for software structural testing. Swarm Evol. Comput. **20**, 23–36 (2015)
15. Aghdam, Z.K., Arasteh, B.: An efficient method to generate test data for software structural testing using artificial bee colony optimization algorithm. Int. J. Softw. Eng. Knowl. Eng. **27**(06), 951–966 (2017)
16. Ghaemi, A., Arasteh, B.: SFLA-based heuristic method to generate software structural test data. J. Softw. Evol. Process **32**(1), e2228 (2020)
17. Hayyolalam V., Kazem A.: Black Widow Optimization Algorithm: a novel meta-heuristic approach for solving engineering optimization problems. Eng. Appl. Artif. Intell. **87** (2020)
18. Khanna, M., Chauhan, N., Sharma, D.K.: Search for prioritized test cases during web application testing. Int. J. Appl. Metaheuristic Comput. (2019). https://doi.org/10.4018/IJAMC.2019040101
19. Arasteh, B., Fatolahzadeh, A., Kiani, F.: Savalan: multi objective and homogeneous method for software modules clustering. J. Softw. Evol. Process **34**(1), e2408 (2022)
20. Arasteh, B., Sadegi, R., Arasteh, K.: Araz: a software modules clustering method using the combination of particle swarm optimization and genetic algorithms. Intell. Decis. Technol. **14**(4), 449–462 (2020)
21. Arasteh, B.: Software fault-prediction using combination of neural network and naive bayes algorithm. J. Netw. Technol. **9**(3), 94–101 (2018)
22. Arasteh, B., Sadegi, R., Arasteh, K.: Bölen: software module clustering method using the combination of shuffled frog leaping and genetic algorithm. Data Technol. Appl. **55**(2), 251–279 (2021)
23. Bouyer, A., Beni, H.A., Arasteh, B., Aghaee, Z., Ghanbarzadeh, R.: FIP: a fast overlapping community-based influence maximization algorithm using probability coefficient of global diffusion in social networks. Expert Syst. Appl. **213**, 118869 (2023). https://doi.org/10.1016/j.eswa.2022.118869
24. Arasteh, B.: Clustered design-model generation from a program source code using chaos-based metaheuristic algorithms. Neural Comput. Appl. **35**, 3283–3305 (2023)