Journal of Electronic Testing (2022) 38:205–215
https://doi.org/10.1007/s10836-022-05999-9

Traxtor: An Automatic Software Test Suit Generation Method Inspired by Imperialist Competitive Optimization Algorithms

Bahman Arasteh[1] · Seyed Mohamad Javad Hosseini[2]

Received: 15 December 2021 / Accepted: 30 March 2022 / Published online: 11 April 2022

Abstract
Software testing refers to a process which improves the quality of software systems and also is one of time and cost consuming stages in software development. Hence, software test automation is regarded as a solution which can facilitate heavy and laborious tasks of testing. Automatic generation of test data with maximum coverage of program branches is regarded as an NP-complete optimization problem. Several heuristic and evolutionary algorithms have been proposed for generating test suits with maximum coverage. Failure to maximally branch coverage, poor success rate in test data generation with maximum coverage and lack of stable results are considered as the major drawbacks of previous methods. Enhancing the coverage rate of the generated test data, enhancing the success rate in generating the tests data with maximum coverage and enhancing the stability and speed criteria are the major purposes of the present study. In this study, an effective method (Traxtor) is proposed to automatically generate tests data by using imperialist competitive algorithms (ICA) optimization algorithms. The proposed method is aimed at generating test data with maximum branch coverage in a limited amount of time. The results obtained from executing a wide range of experiments indicated that the proposed algorithm, with 99.99% average coverage, 99.94% success rate, 2.77 average generation and 0.12 s average time outperformed the other algorithms.

Keywords Software testing · Automatic test data generation · Imperialist competitive algorithms · Branch coverage · Success rate

1 Introduction

Software testing is regarded as one of the most significant steps in the process of guaranteeing software quality [1]. Software testing may be conducted manually or automatically. Whereas manual tests need high time and cost, automatic methods reduce time and cost of testing. Given the significance of automatic software testing, it is considered as one of the remarkable challenges and concerns in this research domain. However, in huge real-world software systems, applying such a traditional manual testing would be extremely costly and time-consuming. Software testing at the source-code level may only detect 50% of errors in software development. In contrast, applying automatic software testing can notably reduce cost and time. Designing optimal test cases in the automatic manner at the source-code level is considered to be the problem addressed in this study. Indeed, automatic test data generation with maximum branch coverage at the minimum possible time is regarded as the main optimization issue. Selecting a small subset from the combination of all the possible inputs with maximum coverage of program branches is regarded as an NP complete problem.

Given the nature of this research problem, several heuristic and evolutionary algorithms have been proposed for generating test datasets with maximum coverage [2–10]. Failure to maximally branch coverage, poor average success rate in data productions with maximum coverage in different executions, lack of stable results in different executions and high execution time are considered as the major drawbacks of previous methods. The purposes of the present study are as follows:

Responsible Editor: V. D. Agrawal

✉ Bahman Arasteh
Bahman.arasteh@istinye.edu.tr; b_arasteh2001@yahoo.com

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

[2] Department of Computer Engineering, Sofian Branch, Islamic Azad University, Sofian, Iran

- Enhancing the coverage rate of the generated test data
- Enhancing the success rate in generating the tests data with maximum coverage
- Enhancing the result stability of test data generation methods
- Enhancing the speed of test data generation along with maximum coverage

In this study, an effective method (Traxtor[1]) is proposed to automatically generate tests data by using imperialist competitive algorithms (ICA) optimization algorithms. Indeed, ICA as a heuristic algorithm [11] is developed to sort out the test data generation problem in this study. The proposed method is aimed at generating test data with maximum branch coverage in a limited amount of time. The major contributions of this study are as follows:

- Applying imperialist competitive algorithm (ICA) for generating optimal test data
- Implementing open-source tool for automatic testing of a program with high success rate
- Producing optimal and stable test data with maximum coverage at the minimum time

The rest of the paper is organized as follows: in Sect. 2, basic theoretical concepts and a review of the related works are briefly discussed. Section 3 reports and elaborates on the proposed method in detail. Section 4 discusses the results obtained from executing the program via different methods. Finally, Sect. 5 draws the conclusion of the study and gives directions for further research.

2 Related Works

In [3], researchers used a random method for producing test data. Very high time requirement for achieving the intended coverage and the generation of repetitive data is regarded as the main drawbacks of this method. Also, desirable results in terms of the number of discovered faults were not obtained in these methods. Hence, researchers proposed a method based on symbolic execution so as to achieve better results [4, 5]. Symbolic execution is an effective testing technique that provides a way to automatically generate test data inputs that trigger software errors Concrete test inputs generation is one of the major strengths of symbolic execution the generated test data have high coverage. From a bug-finding perspective, concrete and high coverage test data triggers the bug. Symbolic methods are not able to determine array value and pointer inputs. Consequently, data generation has turned into a challenging issue. In [6] simulated annealing algorithm was uses for solving the problem of test-data production. In this study, by converting testing data generation problem into an optimization problem, optimal test-data is generated by the simulated annealing algorithm. Low performance and low coverage and being placed in local optimum are the main demerits of this method. This method is appropriate for behavioral testing.

By capitalizing on genetic algorithm (GA), a method was proposed in [2] for producing test data. In this method, GA was used for selecting optimal paths. In this research, fitness function was entitled similarity function which was aimed at determining the degree of similarity of the traveled path to the objective path. Path optimality indicates the fact that the path is followed in executing test data. That is, the higher path follow-up, the higher its optimality. The application of GA leads to the reduction of the required time for finding optimal path. Another method was proposed in [7] based on GA for producing test data. In this research, GA was used so as to achieve optimal test data. For enhancing efficiency and effectiveness, researchers implemented this algorithm in the parallel manner. Then, the coverage of the proposed method was analyzed and investigated on six benchmark programs. The obtained results indicated improvement in test data production.

One of major problem in the GA is that the chromosomes do not try to improve themselves and the may improve using only mutation operator. In GA, the fitness function evaluates only whole of chromosome; evaluating a subsection of a chromosome in GA is not possible. This makes the GA to be similar to the blind search algorithms. To address these problems, an automatic test-data generation method has been proposed [8]. In this method, reinforcement learning as a memetic search method was augmented to the GA. This augmented GA focuses on best chromosomes of population and Q-learning has been used for guiding these search process. In method, mutation operator is performed when there are duplicated sub-sections within a chromosome. Experimental results show that this hybrid method is better than GA in terms of the coverage and success rate. In [9], researchers applied particle swarm optimization (PSO) algorithm for producing test data. In a similar vein, PSO algorithm and regression analysis were proposed in [10] for generating test data. Moreover, thanks to its simplicity and high convergence speed, PSO algorithm was applied in [12] for producing the test data with another objective function. Regarding the obtained results in these studies, the coverage of critical path (fault prone paths) using branch distance functions is poor. A PSO search based test-generation method was proposed using an improved fitness function to cover the critical path of program under test [13].

[1] Traxtor is the name of a popular Turk football team in the historical city of Tabriz.

Table 1 Merits and demerit different test generation methods

Method	Merits	Demerits
Random search [3]	Simplicity of implementation	Lack of sufficient information for generating test data
Simulated annealing algorithm (SA) [6]	Higher speed than random search	Dealing with local optimum and low success rate
Genetic algorithm (GA) [2, 7, 8]	Parallel implementation	Low success rate and high runtime
Particle swarm optimization algorithm (PSO) [9, 10, 12, 13]	High speed and simple implementation	Different results in numerous repetitions
Ant colony optimization algorithm (ACO) [14]	Considering the weights of branches	High runtime and highly variable results
Artificial bee colony optimization algorithm (ABC) [15]	Appropriate coverage of branches and high speed	High variance in the obtained results for same applications

A method based on ant colony optimization (ACO) algorithm was proposed in [14] for producing optimal tests data which was aimed at maximizing branch coverage. A coverage based specific fitness function was defined in this method. The results of the experiments showed that the coverage, convergence speed and stability of results are higher in this method. In [15] the effectiveness of artificial honeybee colony (ABC), genetic, particle swarm, simulated annealing and ant colony (ACO) algorithms have been evaluated and compared to each other. In this study, distance function based on branch coverage is used as fitness function. The results of conducted experiments of the traditional benchmarks revealed that the coverage, success rate, average number of iterations in ABS are respectively 99.94%, 99.92%, 3.36. Indeed, ABC algorithm performed better than the other algorithms in producing optimal test data. In [16], the shuffled frog leaping algorithm (SFLA) was applied to develop an automatic test-data generation method. In this method, branch coverage was used into fitness function. An extensive series of experiments have been performed on the seven tradition benchmarks for evaluating this method. The results illustrated that this method has several merits over the previous evolutionary algorithms such GA, PSO, ACO and ABC. The SFLA based method can generate test data with 99.99% branch coverage with the minimal iterations. Also, its success rate in generating the optimal test data is 99.97%.

According to the previous studies, which are briefly mentioned in Table 1, it can be maintained that the method proposed so far have their own pros and cons. In other words, they have not been fully able to solve the problem of data generation for automatic software testing. Consequently, in this research study, an automatic test generation method using ICA optimization algorithm. The authors of this study make an effort to address the previous research gap. The proposed method is discussed and elaborated in Sect. 3.

3 The Proposed Method

As shown in Fig. 1, an automatic method (Traxtor) was proposed to generate structural test data. In the first step, source code of the program under test is statically analyzed and the required structural information is extracted for the next steps. In the second step, optimal test data is generated by means of ICA (imperialist competitive algorithm). The objective function, applied in this study, was defined based on the distance function and branch coverage. In fact, ICA navigates and directs the generated data so as to enhance the coverage of program branches. The output

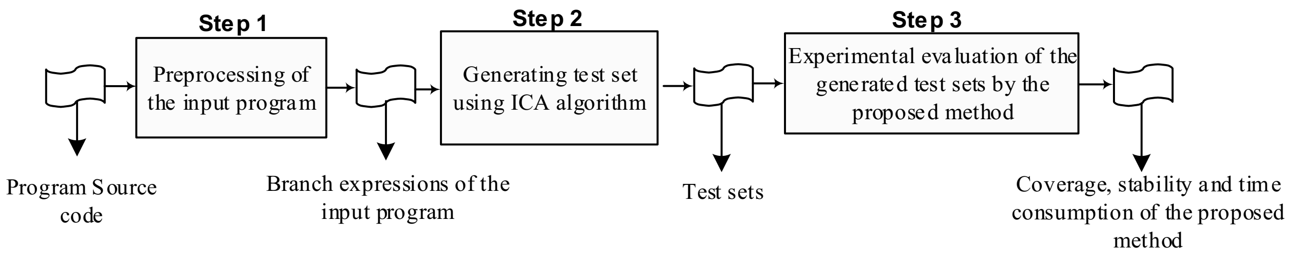
Step 1
Preprocessing of the input program
Step 2
Generating test set using ICA algorithm
Step 3
Experimental evaluation of the generated test sets by the proposed method
Program Source code
Branch expressions of the input program
Test sets
Coverage, stability and time consumption of the proposed method

Fig. 1 The process of the proposed method

of the second step is the test set (test suit) generated by the ICA.

3.1 Test Data Generation via ICA

As mentioned above, an automatic method was proposed in this study for generating optimal test data via ICA. In this section, we discuss the method of automatically generating test data by means of ICA. Like other evolutionary algorithms, this algorithm starts with a number of random initial populations each of which is referred to as a country. Each member of the population (individual or country) is a test data which was randomly generated in the initial population. A number of the best individuals, which are equal to elites in genetic algorithm, are selected as the imperialists. The best member of a population is the individuals (countries) whose fitness function is higher than those of other population members. The policies of assimilation, imperialistic competition and revolution establish the foundations of ICA. By imitating the social, economic and political evolution of countries and via mathematical modeling of some sections of this process, ICA presents operators in a regular format as an algorithm. These operators may facilitate the solving of complicated optimization problems. In fact, this algorithm views problem solutions in the form of independent countries and tries to gradually improve and optimize the solutions throughout a repetitive process; in this way, it is aimed at achieving the final optimal solution. As the time passes, colonies will become closer to the empires in terms of power. Hence, a sort of convergence is realized. The final limit of imperialist competition is when there is a unitary empire in the world with colonies which are highly close to the imperialist country in terms of position. The steps of the proposed method are as follows:

Input: programs to be tested and the data generated by ICA.

Output: the optimal suite and evaluation criteria (average coverage, success rate, average convergence and average time).

Steps:

1. Selection of a number of random test data as colonies and the establishment of initial empires

    • Specifying the structure of test data according to source code structure.
    • Constructing the initial population of countries (test data).
    • Determining absorption policy.
    • Determining revolution policy and revolution rate.
    • Defining cost function.

2. The movement of colonies (test data) towards the imperialist country (assimilation policy).
3. Computing the value of fitness function based on each test data.
4. In case there is a colony in an empire whose fitness function is greater than that of the imperialist, the position of the imperialist and the colony is replaced.
5. Computing the total fitness of an empire (by considering the cost of the imperialist and its colonies).
6. Selecting a colony from the weakest empire and allocating it to the empire with the highest probability of possession.
7. Eliminating weak empires.
8. If only one empire remains, the algorithm is stopped; otherwise, steps 2 to 6 are repeated.
9. The remaining elements in the empires are regarded as the best obtained test datasets.

In the proposed method, source-code of the input program is firstly analyzed and the number of parameters, paths and the weight of the branches are determined and valued. After the values of these parameters are specified and the main variables of ICA are randomly determined, the imperialist arbitrarily replaces a percentage of its array, which is parts of test data, with corresponding imperialist data. Then, the next action in the procedure of implementing ICA is to eliminate and replace weak colonies (countries) with strong colonizers. This measure is taken by computing the value of fitness function and selecting the individuals with a better cost function; this is technically referred to as the revolution. The imperialist competition refers to the fact that colonies or imperialists are doing their bests in absorbing the colonies of other empires. Throughout the execution of the ICA, low-power countries or colonies (test data) are randomly given (absorbed) by stronger empires. This procedure continues so that colonies are conformed and assimilated to their empires. In other words, the values of the imperialist and colonies' fitness functions gent closer to each other. In practice, this step is aimed at reaching a solution out of a large number of solutions. This operation is realized as a result of removing weak empires.

3.2 Fitness Function

Selecting an appropriate fitness function is regarded as a highly significant step in optimization problems. By using fitness function, the heuristic algorithm manages the population members and finally optimizes them. Distance function has been used as the fitness function in the proposed method. Since the criterion of branch coverage is considered to be one of the most effective indexes in investigating the structural testing of software, it was used in this study for computing fitness function. In this function, the input program has S branches (conditional instruction). Each branch of the program is determined by bchi variable. If the number of the input data (test data) is equal to m, each input will be determined with $X_k \varepsilon TS\ (1 \le K \le m)$. TS is the generated test set by the test generation method. Equation (1) is used for computing the fitness or optimality of a test data generated by ICA.

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

In this equation, ∅ is a constant value which is obtained through trial and error; its value, in this study, was 0.01. *w* variable denotes the weight of branches. *f* indicates distance function. Equation (2) is used for computing fitness function of a test dataset produced by ICA.

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

In case TS (test suit) can cover all the branches, Eq. (2) will be measured as 1/∅ and the highest fitness is achieved. In this issue, we intend to maximize the value of fitness function. According to Eq. (3), fitness function is made up of distance function; this function indicates the degree of presumed deviance of a conditional instruction after values of component inputs are assigned. According to studies [9], distance values of the branch predicates that are used in the conditional expression of a program is shown in Table 2. In case the conditional expression has true values based on the produced data, the value of distance function will be zero; otherwise, the value of δ variable will be added to the value of conditional expression. In this study, the value of variable was 0.1.

3.3 Weight of Branches

Weight of branches indicates the accessibility degree of program branches which are tested. The higher the weight of branches, the more should the algorithm try to reach the branch. The effective factors which have an impact on branch weight are as follows [9]:

- Nesting weight of the branches
- Predicate weight of the branches

The nesting weight which indicates the level of branches is determined by Eq. (3). The higher nesting weight, the more the difficulty of access to that branch.

$$wn(bch_i) = \frac{nl_i - nl_{min} + 1}{nl_{max} - nl_{min} + 1} \tag{3}$$

In this equation, variable i refers to the number of the ith branch. nl_i variable stands for the nesting level of the ith branch. nl_{min} denotes the lowest nesting level in the program and nl_{max} indicates the maximum nesting level in the program. Equation (4) was also used for normalizing the nesting weight of the branches. In this equation, the ith branch weight is divided on the total weight of the branches.

$$wn'(bch_i) = \frac{wn(bch_i)}{\sum_{i=1}^{s} wn(bch_i)} \tag{4}$$

Predicate weight indicates the complexity degree of the predicates of the branches. The predicates should have true values based on the produced input data (test data) so that they can be covered. Equation (5) and Table 3 were used for obtaining predicate weight. In this equation, the following two states may occur:

- If the respective branch includes h conditions which have been combined with each other via and operator, the total weight of the predicate will be equal to the square root of the total weights of the predicates.
- If the respective branch has h conditions which have been combined with each other via or operator, the lowest value out of the weight of the condition predicates will be selected.

$$wp(bch_i) = \begin{cases} \sqrt{\sum_{j=1}^{u} w_r^2(c_j)}, \textit{if conjunction is and} \\ \min\{w_r(c_j)\}, \textit{if conjunction is or} \\ \min\{w_r(c_j)\}_{j=1}^{h}, \textit{other wise} \end{cases} \tag{5}$$

Table 2 Branch function for different kinds of branch predicates

No.	Predicate	Branch distance function $f(bch_i)$
1	Boolean	If true then 0 else k
2	~a	Negation is propagated over a
3	a = b	If abs(a – b) = 0 then 0 else abs(a – b) + k
4	a ≠ b	If abs(a – b) = 0 then 0 else k
5	a < b	If a – b < 0 then 0 else abs(a – b) + k
6	a ≤ b	If a – b ≤ 0 then 0 else abs(a – b) + k
7	a > b	If b – a < 0 then 0 else abs(b – a) + k
8	a ≥ b	If b – a ≥ 0 then 0 else abs(b – a) + k
9	a and b	f (a) + f (b)
10	a or b	min(f (a), f (b))

Table 3 Weight of operators for determining the value of predicate weight

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

In Eq. 5, $bchi,(1 \leq i \leq s)$ indicates the i^{th} branch. For computing the weight of the i^{th} predicate, h variable states the number of available conditions within the branch. $c_j,(1 \leq j \leq h)$ indicates the i^{th} condition and w_r refers to the condition weight whose value is determined based on Table 3. Also, Eq. (6) was used for normalizing the weight of predicates in which the predicate weight of the respective branch is divided on the total weight of the predicates.

$$wp'\left(bch_i\right) = \frac{wp\left(bch_i\right)}{\sum_{i=1}^{s} wp\left(bch_i\right)} \tag{6}$$

Finally, Eq. (7) was used for computing the weight of each branch. This equation indicates that the weight of each branch is made up of the *nesting weight* plus the *predicate weight* of the branch. w_i refers to the i^{th} branch weight and α is the balance coefficient. In this study, the impact of nesting weight and predicate weight were assumed to be equal. The value α was 0.05.

$$w_i = \propto .wn'\left(bch_i\right) + (1 - \propto).wp'(bch_i) \tag{7}$$

4 Results and Discussion

4.1 Implementation System

Matlab program was used for implementing the proposed method. Indeed, Matlab is a practical and applied language for computational operations. It provides the opportunity for carrying out numerous computational, programming and demonstrational operations. A wide majority of computational techniques and problems, particularly the ones dealing with vector and matrix formulas can be easily implemented in Matlab program. In this study, the proposed method as well as other method based on ABC, ACO, PSO, GA and SA algorithms were implemented in Matlab for generating test data. The proposed method and the other methods were all implemented on the same computer system with 8 GB memory and Intel Corei7 CPU. The evaluation criteria used in this study are as follows:

1. AC (average coverage): it denotes the degree of branch coverage of program branches by the generated test dataset. The higher the value of this criterion, the better the effectiveness of the respective method (effectiveness criterion).
2. AG (average generation convergence): this criterion indicates the average number of required iterations for covering all the program branches by the respective algorithm. The lower the value of this criterion, the higher the performance of the respective algorithm (performance criterion).
3. AT (average time): this index refers to the average time needed for covering all the program branches. This criterion is measured in milliseconds (ms). Regarding this index, lower values demonstrate a desirable performance for the related algorithm (performance criterion).
4. SR (success rate): it denotes the coverage probability of all the program branches through the generated test data. The higher the value of this criterion, the better the effectiveness of the respective method (effectiveness criterion).

The number of iterations is considered to be the termination condition in the above-mentioned algorithms with respect to generating optimal test data. Maximum number of the iterations of the algorithms is 300. Furthermore, for computing the average value for these criteria, each algorithm was executed on each benchmark program for 10 times. Then, the average values and standard deviation were computed and compared with each other.

4.2 Benchmark Programs

In this study, 6 benchmark programs with differing degrees of complexity were used. Table 4 gives the characteristics of each of these benchmark programs which have been also used in other previous works. Source-codes of the benchmark programs are written in C++ programming language. ICA, ABC, ACO, PSO, GA and SA algorithms were used for generating test data of these benchmark programs. The real-world program's source code is portioned into the

Table 4 Characteristics of the 6 benchmark programs

Program	#Arg	#Arg.type	LOC	Description
TriangleType	3	Integer	31	Type classification for a triangle
calDay	3	Integer	72	Calculate the day of the week
IsValidDate	3	Integer	41	Check a date is valid or not
cal	6	Integer	26	Compute the days between two dates reminder
Reminder	2	Integer	7	Calculate the reminder of an integer division
printCalender	2	Integer	124	Print calendar according to the input of year and month

Table 5 The branch coverage (%) of the generated test suits by different methods in different benchmark program

Benachmark	GA	SA	PSO	ACO	ABC	ICA	Best Method
triangleType	95.00	99.88	99.94	100	99.93	100	ACO,ICA
calDay	96.31	99.97	100	100	99.8	99.96	PSO,ACO
isValidDate	99.95	97.68	100	99.98	99.90	100	PSO,ICA
Cal	99.02	99.27	100	100	100	100	PSO,ACO,ABC,ICA
Reminder	94.7	99.85	100	100	100	100	PSO,ACO,ABC,ICA
printCalendar	95.06	94.31	99.72	99.85	100	100	ABC,ICA

components, classes and functions; indeed, the large programs with million lines of code composed of modules and finally functions. On the other hand, regarding the programing standards, the size of a function should be about 20 to 100 lines of code. The benchmark programs used in this study are the most traditional and frequently used benchmark programs in the software test studies. Also, these benchmarks include all programing structures that can be used in the real-world complex software. All conditional, loop, arithmetic, logical and jump operators and instructions are used in these benchmark programs. Regarding the generated control flow graph of these programs (structural viewpoint), algorithm and computation of these programs (behavioral viewpoint), the source code complexity, similar results can be generated by the proposed method on the real-world programs.

4.3 Evaluation of the Results

A wide range of experiments were conducted for investigating and evaluating the proposed method based on the above-mentioned specific criteria which are discussed in Sect. 4.2. As mentioned above, one of these criteria is the average coverage of the program branches by the generated test data. Each test generation method was executed for 10 times for obtaining the average coverage of branches. Table 5 shows the average coverage of branches for different benchmark programs by the generated test data. According to the obtained results, in most benchmark programs, the generated test data by the proposed method (Traxtor) has higher branch coverage. Indeed, the data with higher branch coverage is more capable of detecting and discovering errors. It was found that the test data generated by the proposed method is more efficient and effective with respect to the criteria. In the first set of experiments, the proposed method along with five other methods was executed on the benchmark programs. Then, the average coverage of the generated data by each method was examined. The average coverage of the generated test suit in different executions of each algorithm is given in Table 5. The proposed algorithm achieved 100% coverage in *Remider*, *Cal, printCalendar***,** *isValiDate* and *triangleType* benchmark programs. It obtained 99.98% coverage in *calDay* program; regarding *calDay* benchmark program, the proposed algorithm obtained 99.96% coverage meanwhile ACO algorithm had 0.04% better coverage than the proposed algorithm in *calDay* program. Also, regarding *CalDay* program, PSO algorithms had slightly better coverage than the proposed algorithm. It should be noted that the obtained values by this criterion for evaluating the methods depend on the program to be tested, the number of program branches and branch weight. The proposed algorithm outperformed the other methods in 5 out of 6 benchmark programs. Figure 2 depicts the average coverage of the generated test by different methods. The average coverage of the test data generated by the proposed algorithm is 99.99% which is higher than the average value obtained by the other methods.

As mentioned above, success rate (SR) is another criterion in test data generation with 100% coverage. In this study, another set of experiments was carried out with respect to this criterion. Each test data generation method was executed on each benchmark program for 10 times. Average number of the times that the test data generation method achieved 100% coverage indicates SR of that method. Table 6 gives the results of experiments regarding SR criterion. As it may be inferred from the results, the proposed algorithm achieved 100% success rate in *triangleType, CalDay, Cal and Reminder* programs. Regarding isValidDat benchmark program, the proposed algorithm had high a SR which was close to 100%. ICA performed better than the

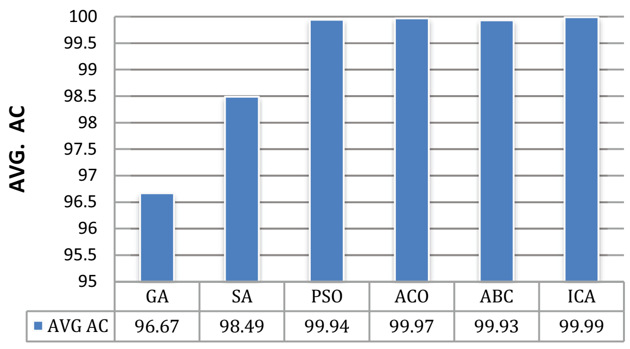

Fig. 2 Average coverage of test data generated by different methods

Table 6 The success rate (%) of different methods in generating test suit with 100% branch coverage

Benachmark	GA	SA	PSO	ACO	ABC	ICA	Best Method
triangleType	76.40	99.40	99.80	100	99.90	100	ACO,ICA
calDay	65.00	99.60	100	100	99.74	100	PSO,ACO,ICA
isValidDate	99.40	95.30	100	99.80	99.90	99.96	PSO,ICA
Cal	98.70	96.50	100	100	100	100	PSO,ACO,ABC,ICA
Reminder	82.50	98.60	100	100	100	100	PSO,ACO,ABC,ICA
printCalendar	61.60	20. 1	99.10	99.20	100	99.74	ABC,ICA

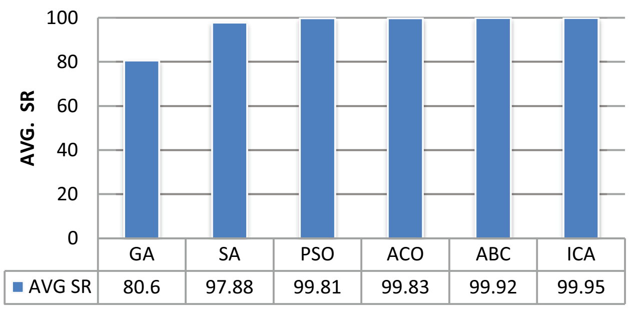

Fig. 3 Average success rate of different methods in achieving 100% branch coverage

other algorithms in printCalendar benchmark program; the results revealed that ACO algorithm has slightly better SR criterion. As shown in Fig. 3, the proposed algorithm, on the whole, had higher SR in achieving 100% branch coverage of the programs. Hence, by considering SR criterion, it can be argued that the proposed method generates more effective test data than the other methods.

Convergence speed is the next criterion which underscores the time and cost in generating test data. The number of required repetitions for a heuristic method in generating test data with maximum branch coverage indicates lower convergence speed of that algorithm. In fact, a method which can produce optimal test data within a fewer number of repetitions will be more efficient and effective. Table 7 gives the average convergence speed of different methods during 10 executions. The results indicated that the proposed algorithm had lower average convergence on all benchmark programs except for *isValidDate* program. Hence, it can be maintained that ICA demonstrated higher performance on this criterion. In other words, the proposed method can produce optimal test data with maximum coverage via fewer numbers of repetitions. Figure 4 shows the number of required repetitions for generating optimal test data (maximum coverage) on 10 different executions. Based on the obtained results, it can be concluded that the proposed method, via 2.7 average number of repetitions, was able to produce optimal test data. In this study, optimal test data refers to data which has maximum branch coverage. As a result, the proposed algorithm has high efficiency and effectiveness in terms of convergence speed.

Average execution time of each method in automatic test generation was taken into consideration as another criterion. Table 8 reports average time for 10 executions of different methods on different programs. Hence, the proposed method and the other ones were investigated in terms of average

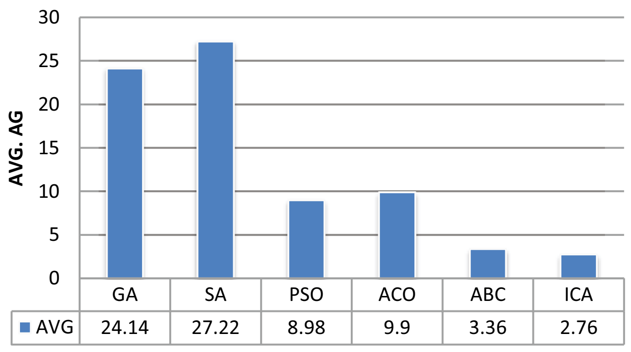

Fig. 4 Average number of executed repetitions by different methods for generating test data with maximum coverage

Table 7 Required average generation to produce test suite with maximum branch-coverage by different methods

Benachmark	GA	SA	PSO	ACO	ABC	ICA	Best Method
triangleType	13.79	42.17	5.36	5.76	1.94	1.7	ICA
calDay	35.80	28.29	10.37	9.51	4.99	3.6	ICA
isValidDate	21.69	15.37	11.90	15.16	0.99	2.4	ABC
Cal	15.24	10.26	8.33	9.58	4.06	3.1	ICA
Reminder	16.31	13.66	5.35	2.01	3.50	1.9	ICA
printCalendar	42.03	53.6	12.59	17.42	4.70	3.9	ICA

Table 8 Average required time to generate test suit with maximum branch-coverage in different methods (Second)

Benachmark	GA	SA	PSO	ACO	ABC	ICA	Best Method
triangleType	10.83	3.77	0.19	6.22	0.1758	0.1027	ICA
calDay	35.73	1.79	0.35	12.84	0.2375	0.1165	ICA
isValidDate	11.68	2.43	0.54	19.94	0.1944	0.1629	ICA
Cal	11.41	0.73	0.50	11.18	0.1367	0.0921	ICA
Reminder	6.09	1.01	0.17	10.49	0.0931	0.0725	ICA
printCalendar	35.48	35.38	1.41	96.27	0.1826	0.1912	ABC

Table 9 Overall average results for the proposed ICA and other methods based on 4 criteria

Criteria	GA	SA	PSO	ACO	ABC	Proposed method (ICA)	Best method
Average coverage (AC %)	96.57	98.49	99.94	99.97	99.94	99.99	ICA
Success rate (SR %)	80.60	84.92	99.82	99.83	99.92	99.94	ICA
Average generation (AG)	24.14	27.23	8.98	9.91	3.36	2.77	ICA
Average time (AT)	18.5367	7.5183	0.5267	26.1567	0.1700	0.1229	ICA

Table 10 Standard deviation of the generated tests suit's coverage in different methods

Criteria	triangleType	calDay	isValidDate	Cal	Reminder	printCalendar	Best method
Standard Deviation of the generated test suit's coverage in 10 times execution	0.37	0.11	0.21	0.13	0.02	0.019	ICA

execution time. It was observed that the proposed method had shorter execution time for all the tested benchmark programs except for *printCalendar* program; ABC algorithm had relatively better performance only on this program. In general, it can be pointed out that the proposed method had successful performance on this criterion.

Here, the average obtained results of the proposed method and other methods are compared with each other with respect to 4 evaluation criteria on the benchmark programs which are given in Table 9. The analysis of the results reveals that the proposed method had better effectiveness (AC and SR) and performance (AG and AT) than the other methods on four evaluation criteria (AC, SR, AG, AT). Given the nature of heuristic algorithms, the stability of the results produced by them should be investigated. In this study, the standard deviations of the test data produced by different methods were measured and evaluated. Table 10 shows the standard deviation values regarding the coverage of the generated test data. They were computed and measured on 10 different executions based on the criterion of coverage degree. Lower standard deviation values in different executions indicate higher stability of the respective method. As shown in Table 10, the proposed method has lower average standard deviation than ACO, PSO, GA and SA. Consequently, in line with the results, it can be mentioned that the proposed method (Traxtor) is more reliable in generating optimal test data. Furthermore, Table 11 gives the generated test suite for *Cal* benchmark program by the proposed method (Traxtor). This benchmark program has 6 input parameters which measures the number of days between two specific dates. The code of the implemented tool (Traxtor) was uploaded in [17] to use freely by the software testers.

In order to evaluate the effectiveness of the generated test data by the proposed method, the mutation test [18, 19] have been performed. In this series of experiments, a set of faults (bugs) have been automatically injected by Mujava tool [20] into each benchmark program; then the generated test data by the proposed method was used to find the injected bugs in each program. Mujava calculates the mutation score of the generated data for each program. The Mutation score depicts the capability of each test set in finding the injected bugs. Table 12 shows the mutation score (fault detection capability) of generated data for each program. The results confirm the effectiveness of the proposed method in generating bug detecting data.

Table 11 Generated test dataset for *Cal* benchmark program by the proposed method

#test data	Input1	Input2	Input3	Input4	Input5	Input6
1	13	1	1566	0	3	1525
2	12	11	1808	20	1	2109
3	1	12	1754	9	12	1921
4	0	2	1487	16	2	619
5	21	0	3000	31	10	3000
6	23	0	164	10	3	1803
7	10	3	0	17	7	1730
8	31	8	588	25	2	1939
9	2	1	2514	6	0	2852
10	9	12	143	20	5	1141
11	0	7	1228	21	5	2726
12	11	5	1337	10	12	645
13	0	12	0	5	10	1757
14	22	6	1873	14	6	2939
15	14	5	2625	25	12	2571
16	8	6	1996	24	12	1462
17	0	10	1687	11	12	1044
18	5	8	2435	6	6	3000
19	17	7	2665	2	0	1153
20	20	1	535	0	0	105

Table 12 The mutation score of the generated test data by the proposed method calculated by Mujava tool

	triangleType	calDay	isValidDate	Cal	Reminder	printCalendar
Mutation Score of the generated test data by the Traxtor	99.72%	92.20%	99.02%	99.00%	93.80	99.07

5 Conclusion

Heuristic algorithms are used for solving the problem of optimal test data generation of software. However, each heuristic algorithm has its own pros and cons with respect to this problem. In this study, ICA algorithm was used for automatic test data generation of software. Four criteria, namely average branch coverage, success rate, average number of generations and average execution time were used for evaluating the results. The results obtained from executing a wide range of experiments indicated that the proposed method (Traxtor) performed more efficiently than the other algorithms with regard to average coverage, success rate, average generation convergence and average execution time. As a direction for further research, other evolutionary algorithms and the combination of them may be applied for solving the problem of automatic test data generation of software. Also, fitness function can be varied and modified in future studies for developing more optimal methods for addressing this problem.

Funding The authors declare that no funds, grants, or other support were received during the preparation of this manuscript.

Data Availability The datasets generated during and the implemented code during the current study is available in the google.drive can be freely accessed by [17].

Declarations

Disclosure The authors have no relevant financial or non-financial interests to disclose. All authors contributed to the study conception and design.

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 Electrical Control Comput Eng 2011 (InECCE), 293–297. https://doi.org/10.1109/INECCE.2011.5953894.
4. Eler MM, Endo AT, Durelli VH (2016) 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, ISSN. 0164–1212

5. Cristian C, Koushik SS (2013) Symbolic Execution For Software Testing: Three Decades Later. Commun ACM 56(2):82–90
6. Cohen MB, Colbourn CJ, Ling AC (2003) Augmenting Simulated Annealing to Build Interaction Test Suites. Proc Fourteenth Int Symp Softw Reliab Eng (ISSRE'03), 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 3(5):1815–1824
11. Atashpaz-Gargari E, Lucas C (2007) Imperialist competitive algorithm: An algorithm for optimization inspired by imperialistic competition. Proc IEEE Congress Evol Comput (CEC 2007), Singapore, 4661–4667
12. Ahmed BS, Zamli KZ (2011) A variable strength interaction test suites generation strategy using particle swarm optimization. J Syst Softw 84:2171–2185
13. Sahoo RR, Ray M (2020) PSO based test case generation for critical path using improved combined fitness function. J King Saud Univ Comput Inf Sci 32(4):479–490
14. Mao C, Xiao L, Yu X, Chen J (2015) Adapting Ant Colony Optimization to Generate Test Data for Software Structural Testing. J Swarm Evol Comput 20:23–36
15. 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
16. Ghaemi A, Arasteh B (2020) SFLA-based heuristic method to generate software structural test data. J Softw Evol 32(1)
17. https://drive.google.com/drive/folders/1-i2k86j-PIio3CqwTsH-PSMLh1zGU3a4?usp=sharing
18. 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
19. 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
20. https://cs.gmu.edu/~offutt/mujava

Bahman Arasteh was born in Tabriz. He received master degree in software engineering from Azad University of Arak, Iran and the Ph.D. degree in software engineering from Islamic Azad University, Tehran Science and Research Branch, respectively. Currently, he is associated professor in Istinye University. His research interests include Software Engineering, Software testing, and Software Fault Tolerance.

Seyed Mohamad Javad Hosseini was born in Tabriz and received his master's degree from Arak Azad University and Ph.D. degree from Islamic Azad University, Tehran Science and Research Branch, respectively. He is faculty member in Islamic Azad University. His research interest include software test.