Journal of Electronic Testing (2022) 38:289–302
https://doi.org/10.1007/s10836-022-06008-9

A Source-code Aware Method for Software Mutation Testing Using Artificial Bee Colony Algorithm

Bahman Arasteh[1] · Parisa Imanzadeh[2] · Keyvan Arasteh[1] · Farhad Soleimanian Gharehchopogh[3] · Bagher Zarei[4]

Received: 6 February 2022 / Accepted: 1 June 2022 / Published online: 29 June 2022

Abstract
The effectiveness of software test data relates to the number of found faults by the test data. Software mutation test is used to evaluate the effectiveness of the software test methods and is one of the challenging fields of software engineering. In order to evaluate the capability of test data in finding the program faults, some syntactical changes are made in the program source code to cause faulty program; then, the generated mutants (faulty programs) and original program are executing with the corresponding test data. One of the main drawbacks of mutation testing is its computational cost. Indeed, high execution time of mutation testing is a challenging research problem. Reducing the time and cost of mutation test is the main objective of this paper. In the traditional mutation methods and tools the mutants are injected randomly in each instructions of a program. Meanwhile, in the real-world program, the probability of fault occurrences in the simple locations (instructions and data) of a program is negligible. With respect to the 80–20 rule, 80% of the faults are found in 20% of the fault-prone code of a program. In the first stage of the proposed method, Artificial Bee Colony optimization algorithm is used to identifying the most fault prone paths of a program; in the next stage, the mutation operators (faults) are injected only on the identified fault-prone instructions and data. Regarding the results of conducted experiments on the standard benchmark programs, Compared to existing methods, the proposed method reduces 28.10% of the generated mutants. Reducing the number of generated mutants will reduce the cost of mutation testing. The traditional mutation testing tools (Mujava, Muclipse, Jester, Jumble) can perform the mutation testing with a lower cost using the method presented in this study.

Keywords Software mutation testing · Mutation reduction · Fault-prone test paths · Artificial bee colony algorithm · Mutation score

1 Introduction

One of the most important considerations for software developers is ensuring the quality of their products. In this approach, software engineers employ software testing techniques in order to identify software flaws. The number of errors found by a test suite is a measure of its efficacy. One of the most difficult areas of research is determining the efficiency of software testing methodologies [2, 4, 12]. Mutation testing is a common way for determining the usefulness of test data. The success of a test set in terms of its capacity to find errors is measured using the mutation

Responsible Editor: V. D. Agrawal

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

Parisa Imanzadeh
p.imanzadeh@gmail.com

Keyvan Arasteh
keyvan.arasteh@live.com

Farhad Soleimanian Gharehchopogh
farahd@iaurmia.ac.ir

Bagher Zarei
Zarei.bager@iau.ac.ir

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

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

[3] Department of Computer Engineering, Urmia Branch, Islamic Azad University, Urmia, Iran

[4] Department of Computer Engineering, Shabestar Branch, Islamic Azad University, Shabestar, Iran

test [10]. Syntactic modifications are made in the primary source code using mutation operators in this test. These modifications are implemented as a fault (bug) injection, and the resulting program (mutated program) will be defective. The test data is used to run the created mutants (faulty programs) and the original program. It can be assumed that the mutant is eliminated (recognized) by the test data if the outputs of the main program and a mutated program vary. The goal is to eliminate all of the mutations that have been created. The success of the test set in terms of its capacity to detect injected flaws is measured using the mutation score (mutants).

The injection of each mutation operator in the program source code results in the creation of a different version of faulty program; each version simulates a certain real bug. The number of mutated versions of a program depends on the number of lines of source code and the number of injected faults. All of the generated mutants should be executed by the test set. One of the main drawbacks of mutation testing is its computational cost. Indeed, high execution time of mutation testing is a challenging research problem. Reducing the time and cost of mutation test is the main objective of this paper. In the traditional mutation methods and tools the mutants are injected randomly in each instructions of a program. Meanwhile, in the real-world program, the probability of fault occurrences in the simple locations (instructions and data) of a program is negligible. With respect to the 80–20 rule, 80% of the faults are found in 20% of the fault-prone code of a program [3, 11].

A suitable selection of mutations should be chosen from the created mutants to reduce mutation test costs. As a consequence, introducing mutation operators (faults) into a program's fault-prone regions yields correct results with a small number of mutations. Furthermore, inserting flaws in basic programs results in the generation of stillborn mutants that are detectable (killed) by all test sets (even poor test data). The suggested approach analyzes the program's source code statically to find the program's fault-prone regions. The suggested technique prevents the modification of non-fault prone (simple) codes, resulting in a significant reduction in the number of mutants. There are 2n execution routes (test path) in a program having n branch instructions. Each of these pathways' data and codes can be regarded a target for mutation operators. Identifying a program's fault-prone (complex) routes is an NP-hard complete. Nowadays, different heuristic algorithms are used to solve various problems in computer engineering [13, 17, 25]. The Artificial Bee Colony optimization technique is utilized in the first step of the proposed method to identify the most fault prone paths of a program. In the second stage, the mutation operators (faults) are injected solely on the identified fault-prone instructions and data. MuJava was used to do the mutation injection procedures [23].

The paper is organized as follows: Sect. 2 reviews the related studies on software mutation testing. Section 3 presents the proposed method. Section 4 is concerned with the simulation of the proposed method, the experiments and evaluation criteria. Also, this section discusses and analyses of the experimental results and the compares the proposed method with other methods. Finally, Sect. 5 concludes the findings of the study and presents directions for further research.

2 Background and Related Work

Different methods have been proposed by researchers for reducing the cost of mutation testing. A brief summary of some seminal methods is as follows:

- **Mutant sampling**: mutant sampling is one of the most straightforward strategies for lowering the number of mutations [6]. Mutant sampling takes a small selection of the created mutants and performs mutations on them. Different scholars have looked into the proportion of different samples ranging from 10 to 40% in 5% increments [30]. The effect of the 10% sample percentage was only 16 percent smaller than the complete set of produced mutants, according to the experimental data. As a result, mutation testing methodologies with a 10% sample percentage can be maintained as a viable choice for mutation. This is in line with King and Offutt's findings [19]. Papadakis and Malevris [28] studied the effectiveness of several mutation sampling methodologies through experiments (from 10 to 60 percent in 10 percent steps). The registered test's effectiveness loss varied from 6 to 26%, according to the researchers.
- ***Selective mutation**:* it is considered as another approximate technique which reduces the number of respective mutants. Random selection mutation, which was introduced by Acree et al. [1], is aimed at reducing the number of executive mutations. It randomly analyzes only a small portion of the mutations. Another strategy, known as limited mutation [26], examines only a certain number of mutations and neglects the other ones. One drawback of this method is related to the manner of selecting operators; also, they are not able to produce different good sets based on the specific purposes. Offutt et al. [26, 27] expanded this idea and investigated the effectiveness of different mutation operator sets. The related findings indicated that 5 operators out of 22 operators are sufficient for investigating the efficacy of mutation test. Barbosa et al. [5] proposed 6 operators for

determining the number of adequate mutation operators. Using these operators led to a set of 10 operators which reduced 65% of the mutations and test efficacy was not lost. Other studies have examined the efficacy of using mutation with only one or two mutation operators. Wong [30] examined the efficacy of using mutation with one or two assignment mutation operators in relation to dependent mutation operator. The experimental evaluation of this approach indicated that the number of respective mutations may be reduced up to 67% and only 5% of the test effectiveness is lost. Also, some experimental results indicated that using this type of mutations does not reduce the quality of the produced test cases. Zhang et al. [33] compared selective mutation with sampling mutation. They investigated three selective techniques vs. two sampling techniques. It was observed that sampling mutation was more effective than selective mutation. Finally, Zhang et al. [32] recommended that selective mutation and sampling mutation may be used together and along with each other so as to obtain promising results.

- **Minimum mutation sets***:* the results show that by targeting other mutations, a substantial number of mutations may be reductively destroyed [24]. Researchers have attempted to estimate the minimal number of mutations required to cover their whole set, which would be sufficient for calculating the minimum set's ability. Kintis et al. [20] were the first to incorporate the smallest number of alterations in the source code of programs in an experiment. Even for mutations that are barely destroyed, the acquired data show that just a tiny percentage of the created mutations (9%) is necessary to cover the full set (35%). Kurtz et al. [21] looked at this topic from both a theoretical and an experimental standpoint. To reduce the number of mutations, they adopted dynamic sharing. The x mutation is dynamically transformed to the y mutation given a test set; that is, the test cases that kill x will also kill y. The dynamic subset's experimental assessment in C programming language revealed that just 12% of the created mutations are required to cover the whole set. Finally, Kurtz et al. [21, 22] looked at whether dynamic and static analysis approaches may be utilized to estimate the relationship between common mutations. They discovered that static and dynamic analysis approaches should be used together to produce better results.
- ***Strong, weak and hard mutations***: in addition to limiting the number of respective mutations for managing mutation costs, researchers have developed different techniques for reducing the implementation cost of all the mutations in the available test set. One such technique is referred to as weak mutation technique which was proposed by Howden [16]. Weak mutation is aimed at reducing the required computational cost for preventing mutation by avoiding complete implementation of the main program and its mutations. For achieving this aim, the weak mutation defines the conditions which should be considered for the mutation as the killed mutation. For comparing the final output of the main program and the mutated program, the internal states of the programs are compared immediately after implementing the mutation or the mutated components. It should be noted that standard mutation is known as strong mutation when it is compared with weak mutation. Woodward and Halewood [31] introduced the concept of hard mutation which is regarded as the one between strong and weak mutation. They argued that we can make comparisons on the internal states of the main program and its mutation at any points between the first implementation of the mutation and the end of the program. Different studies acknowledged the efficiency and productivity of weak mutations. Offutt and Lee (1991) developed a weak framework for FORTRAN77 program; then, they experimentally examined its performance and operation. The obtained results indicated that weak mutation also leads to the manual efficacy loss of mutation by considering fewer equal mutations. Offutt and Lee (1991, 1993) investigated the effectiveness and efficiency of weak mutations by implementing different techniques. They found that this method can be considered as an economical alternative for strong mutation. According to the conducted implementations, researchers proposed that the internal states of the main program and its mutations should be compared with each other after the first execution of the mutated expression or the main block which includes it. Cutigi et al. [7] made a systematic review that characterizes the state-of-the-art in mutation testing cost reduction. It examines the progress of research on this issue, as well as its underlying aims and methodology, and identifies cost-cutting metrics. The research is based on a group of 165 peer-reviewed articles, of which 146 give unique or updated methodologies and outcomes for lowering the cost of mutation testing. A list of six key cost-cutting aims is offered, along with 22 approaches. In the past, 18 measures were employed to quantify the gains and losses reported in experimental investigations. Table 1 illustrates the main features of the related methods.

3 The Proposed Method

Because software mutation testing is time-consuming and expensive, several recent research efforts have concentrated on this topic in order to resolve it. Indeed, the

Table 1 Merits and demerits of the proposed related works for reducing the number of mutations

The methods	Procedure	Merits	Demerits
Mutation sampling: (Offutt and Lee 1993) [1, 6, 19, 29, 30]	Selecting a subset of the generated mutations	Simplicity of conducting the test	Reduced test efficacy
Selective mutation, limited mutation: (DeMillo et al. 1980) [5, 8, 15, 20, 26–28, 30, 32, 33]	Selection of a small set of mutation operators	Maintenance of test effectiveness by reducing 65% of mutations	Low performance of this method alone and the need for combining it with mutation sampling
Minimum mutation sets: [9, 14, 20–22, 24]	It reductively destroys a large number of mutations by targeting other mutations	It requires a small section of produced mutations for covering the entire set	It is imprecise
Strong, weak and hard mutations: (Offutt and Lee 1991) [15, 16, 20, 31]	Weak mutation: it reduces the number of mutations by avoiding complete execution of the program. Strong mutation: it reduces the number of mutations by comparing the final output of the main program and the mutated program Hard mutation: it is an approach between strong and weak mutation	They are economical and require fewer computational resources	They need comparison and they might be imprecise if the total program is not executed

primary goal of such research is to reduce the amount of mutations. We suggested an effective technique for software mutation testing in this paper by utilizing an artificial bee colony algorithm. Only the codes and data of the identified fault-prone paths of the program source code were subjected to mutation operators in this procedure. Indeed, the suggested strategy prevents mutation of the program's non-fault prone (simple) codes, resulting in a significant reduction in the number of mutants. The proposed technique is depicted in Fig. 1.

3.1 Control Flow Graph

In the proposed method, the most fault prone paths of the program should be identified befor injecting mutation operators. As shown in Fig. 1, at the first step, the corresponding control flow graph (CFG) of the program source code should be generated. A CFG is a demonstration of all the possible paths and branches of a program. The graph includes a set of nodes and edges. Each node is defined as a block which includes a set of operators and operands that are executed continuously. In fact, in case an instruction is executed in the block, the entire block will be executed. The presence of directional edge among nodes indicates a possible executing path in the graph. If a node has more than one output edge, it is called branch. The CFG of a program is illustrated in Fig. 2.

The fault prone (complexity) of a path in a CFG is a function of its nodes' complexity. Hence, calculating the weight of nodes (fault prone metric of nodes) is required for calculating the weight of paths (fault prone metric of paths) in the CFG.

3.2 Node Weight

Node weight is utilized in this study to define the fault proneness (complexity) of a node in a CFG. The larger the weight of a node, the more complicated and fault-prone it is. The weight of nodes is calculated by adding the normalized weights of operators and operands. Node weight is indicated via Formula (1). The number of accessible operators and operands in a node are the most important elements that influence the weight of nodes.

$$W'(BB_i) = W'(N_i) + W'(M_i) + \lambda \begin{cases} \lambda = 0.5, \text{ Node have } \textit{if} \text{ instruction.} \\ \lambda = 1, \text{ Node have not } \textit{if} \text{ instruction.} \end{cases} \quad (1)$$

The weight of available operators in each node is denoted by $W(N_1)$ which indicates the total number of available operators in that node. The weight of available operands in each node is denoted by $W(M_i)$ which indicates the total number of available operands in that node. Then, the normalized weight of each of them is obtained and their total is measured. We used Formula (2) for normalizing operators' weights where the number of available operators in the

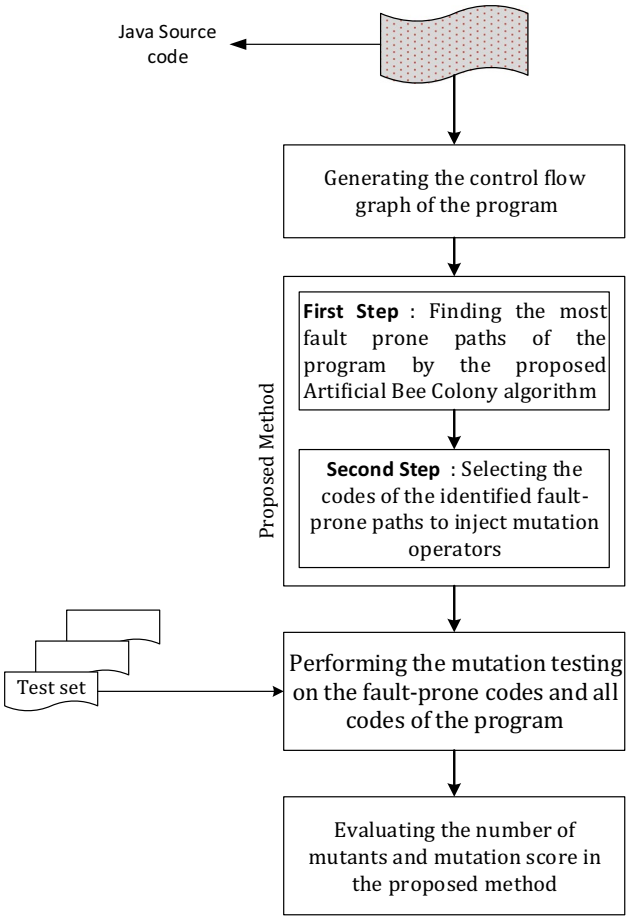


Fig. 1 Steps of the proposed method

respective node is divided on the total number of operators of the nodes.

$$W'(N_i) = \frac{W(N_i)}{\sum_{i=1}^{|p_i|} W(N_i)} \tag{2}$$

For normalizing the weight of operands, we used Formula (3) in which the number of available operands in the respective node is divided on the total number of operands of the nodes.

$$W'(M_i) = \frac{W(M_i)}{\sum_{i=1}^{|p_i|} W(M_i)} \tag{3}$$

3.3 Branch Weight

A branch's reachability is determined by its weight. The algorithm should strive harder to obtain the branch with a greater branch weight. The branch weight is affected by the complexity of the propositions of the decision node. Proposition weight was calculated using Formula (4) and Table 2. The following two states are created by this formula:

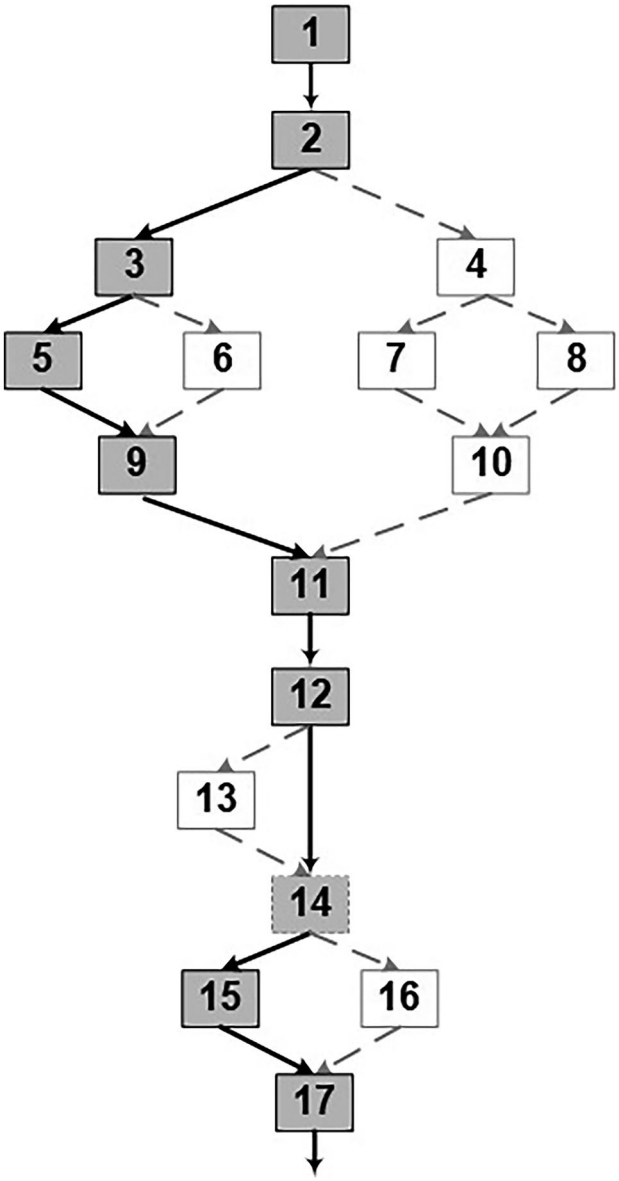


Fig. 2 A control flow graph of a program

- If the respective decision node includes h conditions which have combined with each other through AND operator, the square root of the total weight of the propositions will be considered to be the selection condition.

Table 2 Weights of the operators for computing proposition weight

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

- If the respective decision node includes h conditions which have been combined with each other through OR operator, the lowest weight of the propositions' weight will be regarded as the selection condition.

$$W\left(BCH_j\right) = \begin{cases} \sqrt{\sum_{g=1}^{h} W_r^2\left(C_g\right)}. & if\ conjuction\ is\ AND \\ min\left\{ \begin{array}{c} h \\ g=1 \end{array} W_r\left(C_g\right)\right\}. & other\ wise \end{cases} \tag{4}$$

In this formula, Bch_j variable stands for the j^{th} decision node which is determined as ($1 \le j \le p_j$). In computing the proposition weight of the j^{th} node, h variable refers to the available conditions within the decision node. C_g denotes the g^{th} condition which is determined as ($1 \le g \le h$). W_r variable refers to condition weight which was specified by Table 2. We used Formula (5) for normalizing the weight of propositions in which the proposition weight of the respective branch is divided on the total weight of the propositions.

$$W'\left(BCH_j\right) = \frac{W\left(BCH_j\right)}{\sum_{j=1}^{|p_j|} W\left(BCH_j\right)} \tag{5}$$

Also, Table 2 gives all the operators which may be involved in the condition. Indeed, they determine the weight of the propositions. Finally, Formula (6) indicates objective (fitness) function.

$$Fitness\left(Path_i\right) = \sum_{i=1}^{|p_i|} W'\left(BB_i\right) \times \alpha + \sum_{j=1}^{|p_j|} W'\left(BCH_j\right) \times (1-\alpha) \tag{6}$$

In this formula, the expression $\sum_{i=1}^{|p_i|} BB_i$ denotes the total complexity of the nodes into $path_i$ in which p_i indicates the number of nodes; also, the expression $\sum_{j=1}^{|p_j|} Bch_j$ refers to the total complexity of decision nodes in which p_j stands for the number of available decision nodes in the program. In this formula, α was used as an impact factor; it was applied as the efficacy degree of complexity criteria. In this paper, the value of α was assumed to be 0.5.

3.4 Artificial Bee Colony Algorithm (ABC)

Karaboga and Basturk [18] proposed the Artificial Bee Colony (ABC) method. Its goal was to achieve true parameter optimization. This technique was developed as an optimization algorithm that replicates the unrestricted exploring behavior of a bee colony in terms of optimization difficulties. One limitation-handling approach is paired with this algorithm to solve optimization issues with limitations. There are responsibilities carried out by specialist personnel in a real honeybee colony. That is, the specialized bees use labor division and effective self-organization to strive to optimize the amount of stored nectar in the hive. The program has adapted the minimal food selection search strategy used by smart bee groups in the honeybee colony, which contains three categories of bees: worker bees, observer bees, and precursor bees. The worker bees make up half of the colony, while the observer bees make up the other half. Worker bees are in charge of collecting nectar from previously discovered sources. They should also advise other bees (waiting observation bees in the hive) about the quality and location of the nectar supply being harvested. Observer bees remain in the hive and make food-related decisions based on the information provided by worker bees. To identify new food sources, precursor bees explore the surroundings instinctively, randomly, or based on other external indications.

3.4.1 Steps of the algorithm

The main steps of the ABC algorithm are as follows:

1. Giving an initial value to food source locations.
2. Each worker bee produces a new food source in its own food source location and extracts the better source.
3. Each precursor bee selects a source depending on its solution quality. Then, it produces a new food source in

Fig. 3 Steps of the ABC algorithm

Step 1: producing initial solutions and computing their quality by precursor bees
Step 2: optimizing the presented solutions and re-computing quality
Step3: computing the probability of solution selection and greedy selection of them
Step 4: optimizing selected solutions and re-computing their quality
Step 5: saving the selected solutions
Step 6: presenting a new solution by precursor bee in case the old solution is abandoned
Step 7: return to the 2nd step if the algorithm is not ended
Step 8: showing the selected solutions

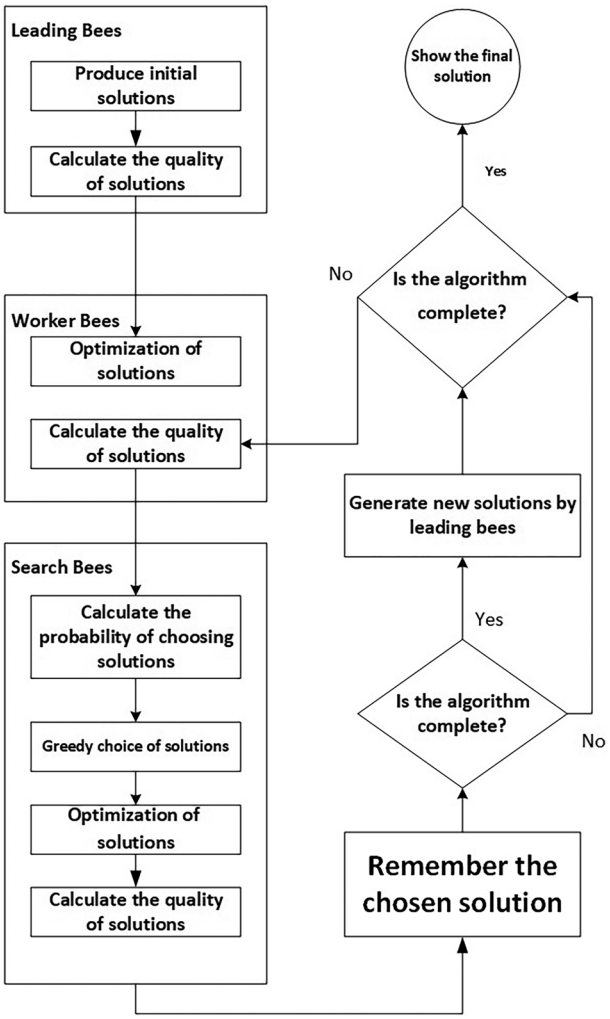


Fig. 4 Stages of the ABC algorithm

the location of the selected food source and extracts the better source.

4. The food source which should be abandoned is determined and some worker bees are allocated as precursor bees that should search for new food sources.
5. The best food source which has been discovered up to now is remembered.
6. The steps 2–5 are repeated until the stop criterion becomes appropriate.

In the first step of the algorithm, $\left(i = 1. \ldots .S_N\right)Xi$, solutions are randomly produced. S_N refers to the number of food sources. In the second step of the algorithm, a new food source is produced by Formula (7) for each worker bee, whose population is equal to the half of the total number of food sources.

$$v_{ij} = x_{ij} + \varphi_{ij}\left(x_{ij} - x_{kj}\right)\varphi_{ij} \tag{7}$$

A random number was uniformly distributed within [-1, 1] interval which controls location production of neighboring food sources around *xij*. *K* is the solution index which was randomly selected from the colony. $j = 1. \ldots .D = int(rand * SN) + 1$ and *D* are problem dimensions. After V_i is produced, this new solution is compared with X_i and the worker bee extracts the better source. In the third stage of the algorithm, an observer bee selects a food source with (2) probability and produces a new source in the location of the food source selected by (1). In the same way as the worker bee, decision is made about extracting a better source. Fit_i refers to the fitness degree of *Xi* solution. After all the observer bees are distributed in the sources, sources are examined to find out whether or not they should be abandoned. If the number of the cycles of a source which are not optimized is greater than the predetermined limit, that source will be regarded as a terminated source. The worker bee related to the terminated source becomes a precursor bee which starts a random search within the problem area by using Formula (8). The operating procedure of the ABC algorithm is depicted in Figs. 3 and 4.

$$x_{ij} = x_j\min + \left(x_j\max - x_j\min\right) * \text{rand} \tag{8}$$

The paths of the CFG are regarded as the input of the ABC algorithm. Each graph path illustrates a honey bee which is specified by an array. The length of the graph stands for the path length. Each member of the array indicates an available node on the path. Based on honey bee structure shown in Fig. 5, each honeybee has 2 characteristics:

- Honeybee position is specified by array.
- Nectar quality in the food source is the amount of the respective target function.

After initial population is randomly produced, the values of the target function defined in Formula (6) are computed for each honey bee. The target function for the proposed algorithm is of the maximization type. Then, by capitalizing on ABC algorithm, high-complexity (most fault prone) paths are selected from the control flow graph. This subset of paths is applied for injecting mutation on the mutation test. These paths are the most fault prone

Fig. 5 Honey bee structure

BB_1	BB_2	.	.	.	.	.	BB_n	Cost

Table 3 Benchmark programs used in the study

Program name	Input parameters	Code lines (LOC)	Program objective
Triangle Type	3	31	Specifying triangle type
CallDay	3	72	Specifying weekday
isValidDate	3	41	Checking the validity of the inserted date
Cal	6	26	Calculating the number of days between two dates
Reminder	2	17	Calculating the remainder integer number

locations of the program. the remaining paths are executive paths which have no impact or minimum impact on the output. As a result, injecting mutation operators on such routes is fruitless and ineffective. Optimization algorithms, random search and evolutionary methods are regarded as modern and efficient methods which are particularly applied for finding global optimal responses for the problems. The randomness feature of these algorithms prevents them from being trapped in local optimal points. Most algorithms have been inspired from biological systems. Honey bee colony algorithm is considered to be an example of such biological systems. It models the behavior of honeybees and assigns a value in accordance to the fitness of the location of each bee for the quality of bees; in this way, by updates bees' locations in consecutive iterations of the algorithm, the algorithm seeks optimal response for the problem. The output of ABC algorithm for each input algorithm is the most fault-prone (most complex) executive paths. The input of the proposed algorithm is the control flow graph and its output is a subset of executive routes with maximum complexity.

4 The Experiments of the Proposed ABC Algorithm

On a 64-bit Win7 operating system, the suggested method was evaluated and implemented in Matlab 2018. The experiments were carried out on a machine with an Intel Core i7 CPU and 4 GB of RAM. The suggested approach was implemented in Matlab, and mutations were injected using the MUJAVA tool. This problem has been solved using a variety of evolutionary techniques. We utilized the ABC technique to discover the program's fault-prone places in this article. As previously stated, an effort was made to identify the program's most complicated (fault-prone) routes. It should be noted that benchmark programs are essential for evaluating the suggested method's efficiency and efficacy. The elements of the benchmark software are listed in Table 3.

One of the shortcomings of evolutionary algorithms is the value specification of the parameters of each method. In many cases, parameters play an essential role in bringing the algorithm closer to the optimal response. ABC algorithm has 2 parameters. Appropriate values for these two parameters within the approximate interval were defined by Karaboga and Basturk [18]. Like other algorithms, the parameters of ABC algorithm are calibrated experimentally through trial and error. Table 4 gives the adjusted parameters of ABC algorithm in this study. The features of the benchmark programs are described in Table 4. All of the programming structures that may be used in real-world software are included in these programs, including:

- if-else structure
- for structure
- while structure
- switch structure
- I/O structure
- Operators for arithmetic and logic

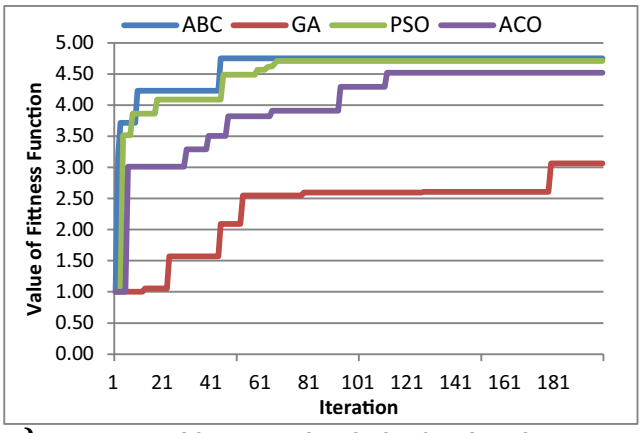
The source code of huge real-world applications (with millions of lines of code), which is made up of modules and

Table 4 Adjusted parameters for ABC algorithm with regard to benchmark programs

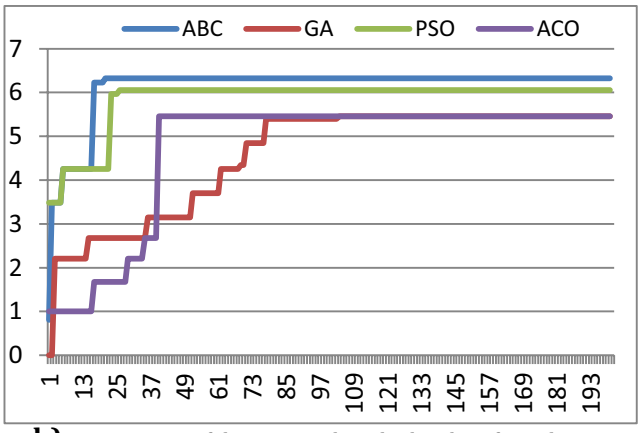
Benchmark Program	Population size	Pm
cal	3	0.03
CallDay	3	0.03
isValidDate	6	0.02
reminder	6	0.006
triangle	80	0.03

Table 5 Success rate comparisons of ABC and GA algorithms in 5 benchmark programs

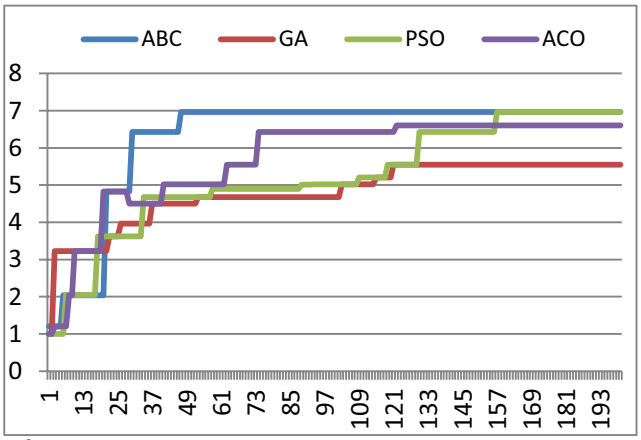
Program name	ABC	GA
Cal	70%	10%
calDay	60%	30%
isValidDate	70%	60%
Reminder	90%	40%
Triangle	90%	10%



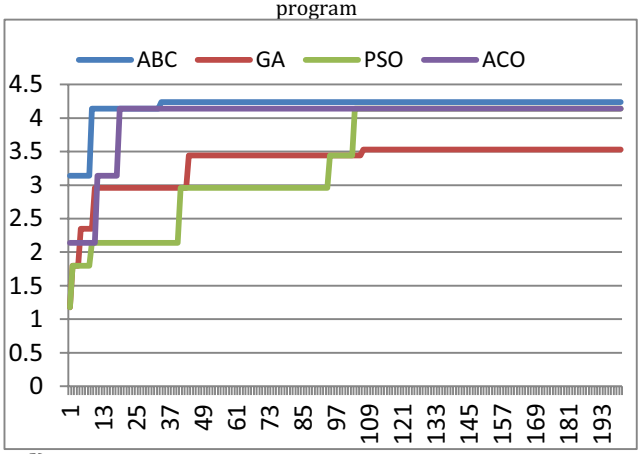
a) Convergence of the proposed method and GA for *cal* program



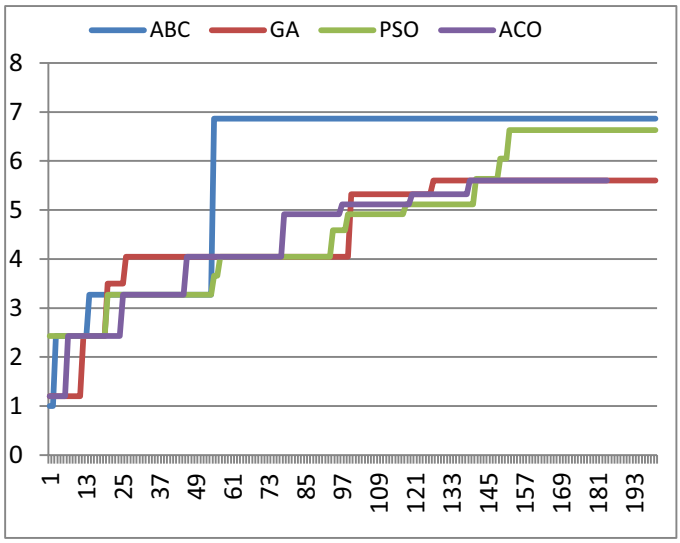
b) Convergence of the proposed method and GA for *calDay* program



c) Convergence of the proposed method and GA for *isValidDate* program



d) Convergence of the proposed method and GA for *Reminder* program



e) Convergence of the proposed method and GA for *Triangle* program

Fig. 6 Comparing the convergence of FA and GA algorithms for 5 benchmark programs

routines. Functions in real-world applications are typically between 10 and 60 lines of code in length. Hundreds of lines of code are not conventional or intelligible, and should be split down into smaller functions.

4.1 Results and Discussion

4.1.1 Success Rate

The collected findings for the performances of ABC and genetic algorithm (GA) algorithms are detailed in the next section in relation to the assessed criteria in five benchmark programs. The optimal value for the fitness function of each program is determined for computing success rate in 10 executions of the method. Table 5 shows ABC and GA's success rate in determining the most error prone routes of each benchmark program in ten executions; this table shows how many executions out of ten have attained the best fitness function value.

4.1.2 Convergence

Figure 6 shows the convergence of the ABC, Genetic Algorithm (GA), Particle Swarm Optimization Algorithm (PSO), and Ant Colony Optimization Algorithm (ACO) algorithms in five benchmark applications. The same benchmark programs were used to run all of the algorithms. The convergence of ABC and GA algorithms in five benchmark programs is shown in Fig. 6. The suggested technique surpasses the previous algorithms in terms of discovering the most error prone routes of the input program and also in terms of convergence speed, as shown in Fig. 6. Table 6 illustrates the average fitness of the algorithms' produced outputs in 10 executions (ABS, GA, PSO, and ACO). The fitness (complexity) of the final generated output (program executing path) created by the ABC algorithm after 44 iterations is around 4.740 in the *cal* benchmark. After 21 rounds on the *calDay* benchmark, the ABC algorithm earned the best result of 6.320. In compared to GA, PSO, and ACO, the ABC algorithm was able to generate a better answer in a shorter time when it came to discovering the fault prone (complex) routes of a program. The ABC algorithm determined the most failure prone routes with 4.230 complexity weight in roughly 35 iterations for the Reminder application, yielding similar results. After 107 rounds, GA was only able to discover the path with complexity weight 3.520. In other words, the suggested ABC algorithm is more successful and efficient than GA, PSO, and ACO in identifying the most complicated paths of programs.

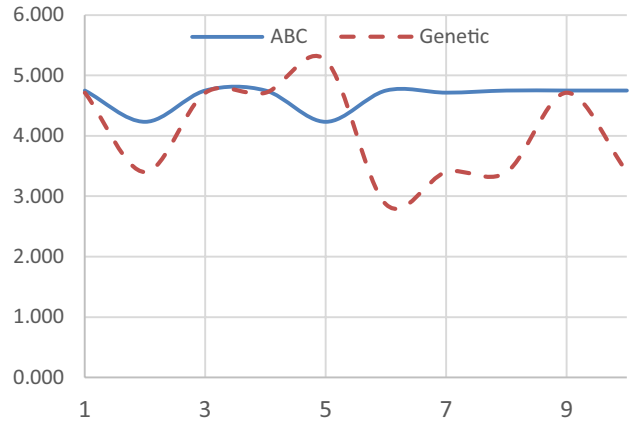
Table 6 The average finesses of the outputs of ABC, GA, PSO and ACO in 10 executions

Program name	ABC	GA	PSO	ACO
Cal	4.740	3.066	4.710	4.514
calDay	6.320	5.450	6.048	5.450
isValidDate	6.961	5.509	6.960	6.600
Reminder	4.230	3.526	4.130	4.131
Triangle	6.869	5.590	6.625	5.558

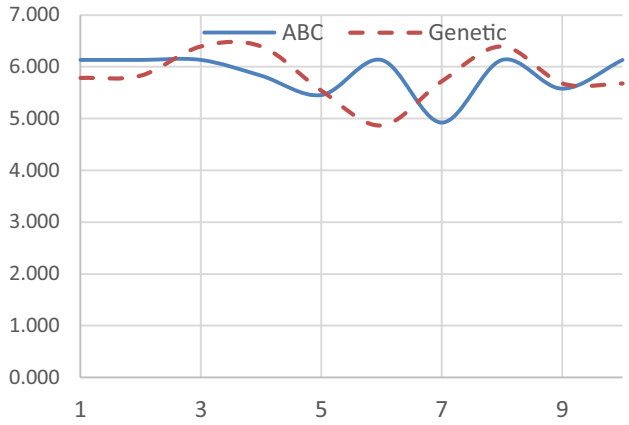
Based on the obtained results (shown in Fig. 6 and Table 6), it can be maintained that the proposed ABC algorithm has better convergence than GA, PSO and ACO. As shown in these figures, although GA has converged earlier than ABC in some benchmark programs, ABC algorithm was able to achieve more optimal responses. It was observed that after the ABC, the PSO based method produces better results.

The achieved outcomes for various executions of each algorithm will be different since the starting population in evolutionary algorithms is formed randomly and the obtained answers in the execution process of each algorithm are random. As a result, we cannot analyze and evaluate the algorithm's performance based on simply one good or negative outcome for a run of the algorithm. As a result, after determining the best parameters of the algorithm, 10 distinct executions of the algorithm with 200 iterations for each different execution were considered in this study. The objective function of the ABC algorithm gave better replies than the GA method, according to findings obtained from 10 executions for 5 benchmark programs.

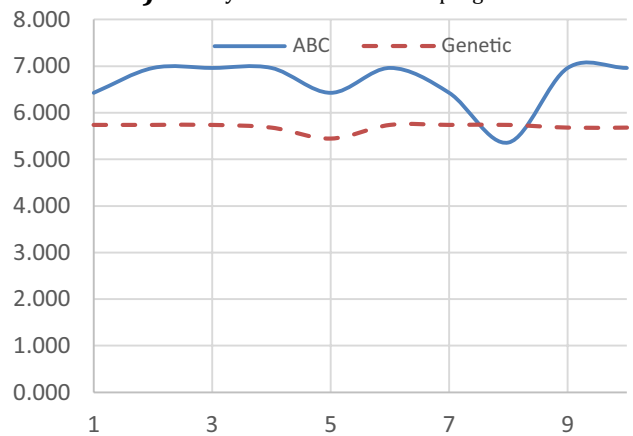
As shown in Fig. 7, ABC algorithm has fewer fluctuations in different iterations. That is, the ABC heuristic algorithm is more stable than GA. The standard deviations (SD) for ABC and GA in different benchmark programs were computed which are given in Table 7. It was observed that the proposed ABC algorithm has fewer standard deviations than GA. In fact, fewer SD in the obtained results is regarded as another evidence for the stability of the ABC algorithm in detecting and identifying the most complex paths of a given program. In other words, it can be argued that the proposed ABC algorithm is more suitable for identifying the most complex (fault prone) paths and injecting mutation.



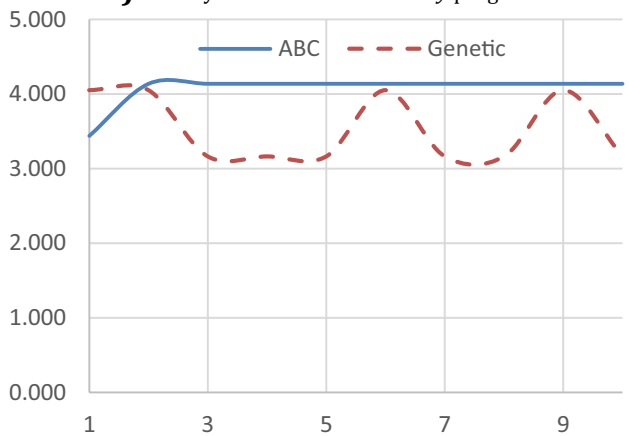
a) Stability of ABC and GA in *cal* program



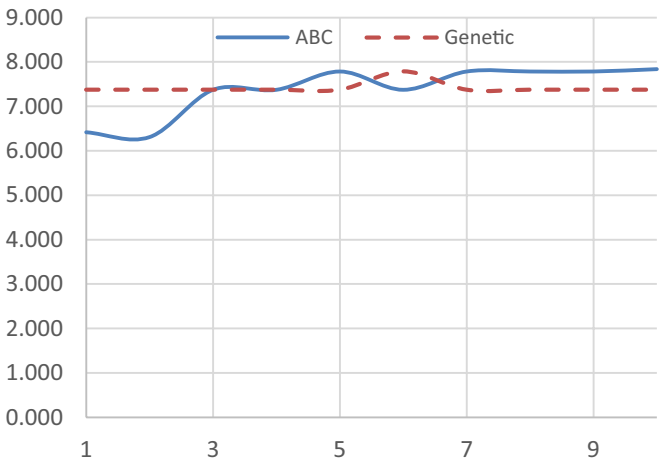
b) Stability of ABC and GA in *calDay* program



c) Stability of ABC and GA in *isValidDate* program



d) Stability of ABC and GA in *Reminder* program



e) Stability of ABC and GA in *Triangle* program

Fig. 7 Stability comparison of ABC and GA in 10 iterations of different benchmark programs

Table 7 Comparison of average results and standard deviation among the generated results for ABC and GA regarding 10 iterations of 5 benchmark programs

Program name	ABC		GA	
	AVG. Fitness	Standard Deviation	AVG. Fitness	Standard Deviation
Cal	643/4	217/0	059/4	836/0
calDay	044/6	418/0	828/5	474/0
isValidDate	561/6	515/0	693/5	090/0
Reminder	069/4	221/0	522/3	459/0
Triangle	382/7	572/0	414/7	130/0

4.1.3 Mutant Reduction

This criterion is used to determine how successful a test set is in detecting errors. Mutation testing is one of the most important procedures performed on test cases in order to validate and acknowledge them. The mutation score is a useful metric for assessing the quality of test cases. It is achieved by the operators' frequent executions of the created altered programs. Tables 8 and 9 show the total mutants created by the proposed technique for all pathways of each benchmark program, as well as the total mutants generated for solely fault-prone paths. The proposed strategy, as indicated in Table 8, lowers the number of mutations. The suggested technique uses the ABC algorithm to identify the source code's fault-prone routes, and then only runs mutation operators on those paths. As a result, the suggested solution minimizes the number of mutations by preventing mutant injection in the program's non-fault prone routes. According on the findings of studies conducted on typical benchmark programs, the suggested technique eliminates 28.10 percent of the produced mutants when compared to existing methods. The cost of mutation testing will be reduced if the number of created mutants is reduced. The proposed technique has a considerable impact on software mutation testing cost reduction. Using the approach given in this paper, classic mutation testing tools (Mujava, Muclipse, Jester, Jumble) may do mutation testing at a lesser cost.

Table 9 The mutant reduction by the proposed method in the benchmark programs

Program name	ABC
Cal	56.12%
calDay	24.55%
isValidDate	04.50%
Reminder	23.22%
Triangle	32.13%
AVG	28.10%

Preventing the mutation of non-error propagating codes of a program decreases the number of generated mutants and increase the performance of the **mutation testing techniques**. The goal of mutation test is to evaluate the effectiveness of a test suite and not to evaluate the program. The main demerits of the mutation test methods and tools is to inject brute force and unreal mutants. Some of the created mutants (faults) does not occur by any programmers in the real-world programs. Regarding the competent programmer hypothesis, the programmer is competent which means the programmer will code programs close to perfection. Hence, the fault (bug) occurring probability in the simple part of a program source code is very low (negligible). The results of the proposed method tires to make real-world faults in the program by avoiding the simple code mutation.

Table 8 The results of mutation testing on all codes and fault-prone codes of 5 benchmark programs in 10 executions

Programs		Total Mutants	Killed Mutants	Live Mutants	Mutation Score
Cal	Mutation of all codes	98	78	20	79.59%
	Mutation of fault-prone codes	43	26	17	60.46%
calDay	Mutation of all codes	167	109	58	65.28%
	Mutation of fault-prone codes	126	73	53	57.93%
isValidDate	Mutation of all codes	111	78	33	70.27%
	Mutation of fault-prone codes	106	61	45	57.54%
Reminder	Mutation of all codes	155	40	115	25.80%
	Mutation of fault-prone codes	119	24	95	20.16%
Triangle	Mutation of all codes	445	304	141	68.31%
	Mutation of fault-prone codes	302	213	89	70.52%

5 Conclusion and Directions for Further Research

The rationale behind this study was to investigate the efficacy of the proposed method in reducing the mutation test by detecting redundant mutations. The proposed method was compared with the previous methods in terms of convergence speed and stability. As discussed above, the obtained results of the proposed ABC algorithm were tabulated with respect to the evaluation parameters. The comparison results revealed that the proposed method has better results than the previous methods. Furthermore, the proposed method can be used into the mutation testing tool such as MuJava to perform mutation test with lower cost. As a direction for further research, other evolutionary algorithms, developed for reducing software mutation testing, can be used for achieving optimal results. Indeed, future studies may focus on overall comparison of evolutionary algorithms for developing an efficient and effective method with regard to mutation testing.

Author Contribution All authors contributed to the study conception and design. Thesis statement, data collection and analysis were performed by Bahman Arasteh. Experiments have been performed by Bahman Arasteh, Parisa Imanzadeh, Keyvan Arasteh, Farhad Soleimanian Gharehchopogh and Bagher Zarei and also, the first draft of the manuscript was written by Parisa Imanzadeh. All authors commented on previous versions of the manuscript. Bahman Arasteh read and approved the final manuscript.

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 the following link: https://drive.google.com/drive/folders/1XFId09ZM88thDHCRWNoTkJWNBiokk-S_?usp=sharing.

Declarations

Conflicts of Interests The authors have no relevant financial or non-financial interests to disclose.

References

1. Acree A, Budd T, DeMillo R, Lipton R, Sayward F (1980) Mutation Analysis. School of Information and Computer Science, Georgia Institute of Technology
2. 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):2017
3. Arasteh B (2019) ReDup: A software-based method for detecting soft-error using data analysis. Comput Electr Eng 78:89–107
4. Arasteh B, Hosseini SMJ (2022) Traxtor: An Automatic Software Test Suit Generation Method Inspired by Imperialist Competitive Optimization Algorithms. J Electron Test. https://doi.org/10.1007/s10836-022-05999-9
5. Barbosa EF, Maldonado JC, Vincenzi AMR (2001) Toward the determination of sufficient mutant operators for C. Softw Test Verification Reliab 11(2):113–136
6. Budd TA (1980) Mutation Analysis of Program Test Data. Yale University
7. Cutigi F, Viola Pizzoleto A, Offutt J (2018) A Systematic Review of Cost Reduction Techniques for Mutation Testing: Preliminary Results. In: Proc. IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp 1–10. https://doi.org/10.1109/ICSTW.2018.00021
8. Delgado-Pérez P, Medina-Bulo I (2018) Search-based mutant selection for efficient test suite improvement: Evaluation and results. Inf Softw Technol 104(2018):130–143
9. Deng L, Offutt J, Ammann P, Mirzaei N (2017) Mutation operators for testing Android apps. Inf Softw Technol 81(2017):154–168
10. Dominguez-Jimenez JJ, Estero-Botaro A, Garcia-Dominguez A, Medina-Bulo I (2011) Evolutionary mutation testing. Inf Softw Technol 53(10):1108–1123
11. Fenton NE, Ohlsson N (2000) Quantitative analysis of faults and failures in a complex software system. IEEE Trans Softw Eng 26(8):797–814
12. Ghaemi A, Arasteh B (2020) SFLA-based heuristic method to generate software structural test data. J Softw Evol Proc 32:e2228. https://doi.org/10.1002/smr.2228
13. Gharehpasha S, Masdari M, Jafarian A (2021) Power efficient virtual machine placement in cloud data centers with a discrete and chaotic hybrid optimization algorithm. Cluster Comput 24:1293–1315. https://doi.org/10.1007/s10586-020-03187-y
14. Gheyi R, Ribeiro M, Souza B, Guimarães M, Fernandes L, d'Amorim M, Alves V, Teixeira L, Fonseca B (2021) (2021), Identifying method-level mutation subsumption relations using Z3. Inf Softw Technol 132:106496
15. Hosseini S, Arasteh B, Isazadeh A, Mohsenzadeh M, Mirzarezaee M (2021) 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
16. Howden WE (1982) Weak mutation testing and completeness of test sets. IEEE Trans Softw Eng 8(4):371–379
17. Jafarian T, Masdari M, Ghaffari A et al (2021) A survey and classification of the security anomaly detection mechanisms in software defined networks. Cluster Comput 24:1235–1253. https://doi.org/10.1007/s10586-020-03184-1
18. Karaboga D, Basturk B (2007) Artificial bee colony (ABC) optimization algorithm for solving constrained optimization problems. Advances in Soft Computing: Foundations of Fuzzy Logic and Soft Computing, vol 4529. Springer, Berlin, pp 789–798
19. King KN, Offutt AJ (1991) A Fortran language system for mutation-based software testing. Softw Pract Exper 21(7):685–718
20. Kintis M, Papadakis M, Malevris N (2010) Evaluating mutation testing alternatives: a collateral experiment. In: Proceedings of the 17th Asia-Pacific Software Engineering Conference (APSEC)
21. Kurtz B, Ammann P, Delamaro M, Offutt J, Deng L (2014) Mutant subsumption graphs. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
22. Kurtz B, Ammann P, Offutt J (2015) Static analysis of mutant subsumption. In: IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
23. Ma YS, Offutt J, Kwon YR (2006) MuJava: A Mutation System for Java. In: 28th International Conference on Software Engineering (ICSE '06)
24. Malevris N, Yates D (2006) The collateral coverage of data flow criteria when branch testing. Inf Softw Technol 48(8):676–686

25. Masdari M, Khezri H (2020) Efficient VM migrations using forecasting techniques in cloud computing: a comprehensive review. Cluster Comput 23:2629–2658. https://doi.org/10.1007/s10586-019-03032-x
26. Offutt AJ, Lee A, Rothermel G, Untch RH, Zapf C (1996) An experimental determination of sufficient mutant operators. ACM Trans Softw Eng Methodol 5(2):99–118
27. Offutt AJ, Rothermel G, Zapf C (1993) An experimental evaluation of selective mutation. In: Proceedings of the 15th International Conference on Software Engineering, ICSE '93. IEEE Computer Society Press, Los Alamitos, CA
28. Papadakis M, Malevris N (2010) An empirical evaluation of the first and second order mutation testing strategies. In: 2010 Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)
29. Wei C, Yao X, Gong D, Liu H (2021) Spectral clustering based mutant reduction for mutation testing. Inf Softw Technol 132:106502
30. Wong WE (1993) On mutation and data flow. Purdue University (Ph.D. dissertation)
31. Woodward M, Halewood K (1998) From weak to strong, dead or alive? An analysis of some mutation testing issues. In: Proceedings of the Second Workshop on Software Testing, Verification, and Analysis
32. Zhang L, Gligoric M, Marinov D, Khurshid S (2013) Operator-based and random mutant selection: better together. In: Proc. IEEE/ACM 28th International Conference on Automated Software Engineering (ASE)
33. Zhang L, Hou S-S, Hu J-J, Xie T, Mei H (2010) Is operator-based mutant selection superior to random mutant selection? In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering

Bahman Arasteh was born in Tabriz. Currently, he is associated professor in Istinye University in Turkey. His research interests include software engineering, software testing, software fault tolerance and software-implemented fault injection.

Parisa Imanzadeh received master's degree in Software engineering from Azad University of Tabriz. His research interests include Software development, maintenance and testing.

Keyvan Arasteh is instructor in Istinye University. He received master's degree in software engineering from Azad University of Tabriz. His research interests include software engineering, web Application security, full-stack programing.

Farhad Soleimanian Gharehchopogh is associated professor in Urmia Azad University in Iran. His research interest includes search-based computer engineering, complex networks, optimization problems and meta heuristic algorithms.

Bagher Zarei is assistant professor in Shabestar branch of Islamic Azad university in Iran. His research interest includes complex networks, evolutionary algorithms and their function in computer engineering.