Journal of Electronic Testing (2023) 39:347–370
https://doi.org/10.1007/s10836-023-06070-x

A Novel Metaheuristic Based Method for Software Mutation Test Using the Discretized and Modified Forrest Optimization Algorithm

Bahman Arasteh[1] · **Farhad Soleimanian Gharehchopogh**[2] · **Peri Gunes**[3] · **Farzad Kiani**[4] · **Mahsa Torkamanian-Afshar**[5]

Received: 23 November 2022 / Accepted: 20 May 2023 / Published online: 20 June 2023
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2023

Abstract

The number of detected bugs by software test data determines the efficacy of the test data. One of the most important topics in software engineering is software mutation testing, which is used to evaluate the efficiency of software test methods. The syntactical modifications are made to the program source code to make buggy (mutated) programs, and then the resulting mutants (buggy programs) along with the original programs are executed with the test data. Mutation testing has several drawbacks, one of which is its high computational cost. Higher execution time of mutation tests is a challenging problem in the software engineering field. The major goal of this work is to reduce the time and cost of mutation testing. Mutants are inserted in each instruction of a program using typical mutation procedures and tools. Meanwhile, in a real-world program, the likelihood of a bug occurrence in the simple and non-bug-prone sections of a program is quite low. According to the 80–20 rule, 80 percent of a program's bugs are discovered in 20% of its fault-prone code. The first stage of the suggested solution uses a discretized and modified version of the Forrest optimization algorithm to identify the program's most bug-prone paths; the second stage injects mutants just in the identified bug-prone instructions and data. In the second step, the mutation operators are only injected into the identified instructions and data that are bug-prone. Studies on standard benchmark programs have shown that the proposed method reduces about 27.63% of the created mutants when compared to existing techniques. If the number of produced mutants is decreased, the cost of mutation testing will also decrease. The proposed method is independent of the platform and testing tool. The results of the experiments confirm that the use of the proposed method in each testing tool such as Mujava, Muclipse, Jester, and Jumble makes a considerable mutant reduction.

Keywords Software mutation test · Bug-prone codes · Forest optimization algorithm · Mutation score

Responsible Editor: Y. K. Malaiya

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

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

2 Department of Computer, Urmia Branch, Islamic Azad University, Urmia, Iran

3 Computer Education and Instructional Technology, Yıldız Technical University, Istanbul, Turkey

4 Computer Engineering Department, Faculty of Engineering, Fatih Sultan Mehmet Vakif University, 34445 Istanbul, Turkey

5 Computer Engineering Department, Faculty of Engineering and Architecture, Nisantasi University, 34398 Istanbul, Turkey

1 Introduction

Testing is an important method to find and remove bugs in a software product [20]. Software engineers employ testing techniques to improve the quality of software. Finding effective test data is the main role of software testers. The percentage of identified bugs by the selected test data indicates its effectiveness. One of the most challenging study fields in software testing is evaluating the effectiveness of test data [2, 9, 14, 31, 32]. Mutation testing is the main technique to evaluate the effectiveness of test data. In this technique, the effectiveness of test data is indicated by mutation score [12]. In the mutation test, programming bugs are made by the mutation operators and injected into the source code of the original program. The injected bugs (mutants) are made by syntactic modification using mutation operators. A set of buggy (mutated) programs are created in the mutation

testing in such a way none of them have compiler error. The mutants (buggy programs) are executed along with the original program to measure the mutation score of the selected test data. A mutant is killed by the test data when the outputs of the original program and the mutated (buggy) program are not the same. The mutation score of a test set is 100% when it kills all of the created mutants. Test data with a 100% score is ideal test data.

Each syntactical modification made by the mutation operator simulates a bug in the program source code. The number of produced mutants (buggy programs) is a function of the lines of code and mutation operators; in the real-world program, a large number of mutated versions are generated, and these mutants should be executed with the test set. Indeed, the computational cost and time of mutation testing are one of the most significant problems in mutation testing. The major goal of this work is to reduce the number of generated mutants, and the time and cost of mutation testing. Mutants are inserted in each instruction of a program using typical mutation procedures and tools. But, in a real-world program, the likelihood of a failure bug occurring in the program's simple areas (instructions and data) is quite low. According to the 80–20 rule, 80 percent of a program's bugs are discovered in 20% of its bug-prone code [5, 13].

A suitable selection of mutant operators and cod sections leads to reducing the cost of the mutation test. As a result, injecting mutants into the bug-prone codes of a program results in a limited number of mutations. Moreover, injecting mutants into the simple codes (codes with low complexity) results in bugs that are found (killed) by poor test data. According to the expert programmer hypothesis, the probability of programming bugs in the non-bug-prone parts of the program is very low. The method proposed in this study makes a static source code analysis to find out the program's bug-prone parts. This method avoids injecting mutants in non-bug-prone codes and makes a large reduction in the number of mutations. In a program having *n* branch instructions, there are 2^n execution paths (test paths). Finding the most bug-prone (most complex) test paths in a program source code is an NP-hard problem. Nowadays, different artificial intelligence and machine learning algorithms have been used to sort out different NP = complete problems in computer science [7, 18]. In the first stage of the proposed technique, the Forrest optimization algorithm (FOA) is used to find out the most bug-prone paths of the program; in the second stage, the mutant operators are injected primarily into the bug-prone sections. MuJava was utilized to achieve program code alteration [25]. The main contributions of this study are as follows:

- A novel heuristic-based method using discretized forest optimization algorithm was developed to find the bug-prone codes of a program source code. The method quantifies the complexity weight of the identified bug-prone codes.
- The mutation operators were applied only to the most bug-prone instructions identified by the method, and the non-bug-prone instructions were eliminated in the mutation test.
- Avoiding the mutation of the non-bug-prone instructions is the main technical advantage of the proposed method, which leads to about a 27% reduction in the number of generated mutants.
- The proposed method is independent of the platform and testing tool. Results of experiments confirm that the use of the proposed method in each testing tool such as Mujava, Muclipse, Jester, and Jumble makes a considerable mutant reduction.
- An open-source tool to analyze a program's source code and find the bug-prone codes of the program was implemented in this study.

Section 2 examines relevant studies. The proposed method is shown in Section 3. The suggested method's simulation, experiments, and evaluation criteria are discussed in Section 4. This part also examines and analyzes the experimental data, as well as compares and contrasts the suggested method with alternative approaches. Section 5 wraps up the study's findings and suggests future research areas.

2 Related Studies

Researchers have proposed many strategies for lowering the cost of mutation testing. Here's a review of several key techniques. It is regarded as one of the simplest methods for reducing the number of mutations [8]. Techniques for sampling mutants aim to take a representative sample of the generated mutants. Researchers have examined the percentage of several samples ranging from 10 to 40% [34]. The effect of the 10% sample percentage was only 16% smaller than the entire set of generated mutants, according to the experimental results. As a result, techniques for assessing mutations with a 10% sampling percentage can still be a good option. This is consistent with King's research results [21]. Papadakis and Malevris [30] investigated the effectiveness of several mutation sampling techniques (from 10 to 60 percent in 10 percent steps). The researchers found that the registered test's effectiveness loss varied from 6 to 26%.

Another approximate strategy for reducing the number of mutants is selective mutation. Proposed random selection mutation as a way to reduce the number of mutations. Only a small percentage of the mutations are randomly examined. Limited mutation [28] is an approach that evaluates only a small number of mutations while ignoring the rest. One disadvantage of this strategy is the way operators are

chosen; also, they are unable to construct multiple excellent sets for different reasons. Offutt et al. [28, 29] investigated the effectiveness of several mutation operator sets to build on this approach. According to the results, measuring the success of mutation testing requires just 5 operators out of 22. Barbosa et al. [6] proposed six operators for estimating the number of suitable mutation operators. These operators were combined to create a set of 10 operators that removed 65 percent of the mutations while retaining test effectiveness. Other studies looked at how successful it was to use just one or two mutation operators. Wong [34] evaluated the efficacy of using mutation with one or two assignment mutation operators in contrast to the dependent mutation operator. According to the experiment results, the number of matching mutations can be decreased by up to 67%, while only 5% of the test efficacy is lost. In addition, multiple studies have demonstrated that inserting these mutations does not affect the quality of the test cases generated. Zhang et al. [36] looked at the differences between sampling mutation and selective mutation. Two sample methodologies were compared to three selection procedures. The selective mutation was shown to be less effective than the sampling mutation. Finally, Zhang et al. [37] proposed that selecting and sampling mutations be used in combination to generate promising results.

The results indicate that by concentrating on other mutants, a considerable proportion of mutations might be eliminated [26]. Researchers have made an effort to determine the fewest mutations necessary to fully cover their set, which would be adequate for determining the minimum set's ability. The experiment that used the fewest changes to program source code was originally carried out by Kintis et al. [22]. The gathered data demonstrates that even for mutations that are scarcely destroyed, just a little portion of the produced mutations (9%) is required to cover the whole set (35 percent). Investigated this issue both theoretically and experimentally. Dynamic sharing was used to lower the number of mutations. Given a test set, the x mutation is dynamically translated into the y mutation; test cases that kill x also kill y. Testing the dynamic subset in the C programming language revealed that just 12% of the generated mutations were necessary to cover the whole set. Last but not least, Kurtz et al. [23, 24] investigated if establishing the association between frequent mutations might be done using dynamic and static analytic approaches. They found that for better results, static and dynamic analysis techniques should be used.

Researchers have developed a variety of strategies for reducing the price of the mutation test's application, in addition to limiting the number of pertinent mutations to reduce mutation costs. The weak mutation strategy is one of Howden's strategies [17]. By omitting the whole implementation of the main program and its mutations, weak mutation aims to lower the processing cost associated with mutation avoidance. To achieve this, the weak mutation lays forth the requirements that a mutation must meet to be labeled as a dead mutation. To compare the final output of the main program and the modified program, the internal states of the programs are compared immediately after applying the mutation or altered components. It should be noted that when compared to a "weak mutation," the average mutation is referred to as a "strong mutation." As a middle ground between strong and weak mutations, Jackson and Woodward [18] proposed the strong mutation. They claimed that we could compare the internal states of the main program and its mutation at any point between the mutation's first execution and the program's conclusion. Weak mutations are useful in many investigations. Offutt and Lee [22] built a fragile structure for the FORTRAN77 software and then assessed its effectiveness and usefulness. The results showed that weak mutations reduced manual effectiveness because fewer related mutations were evaluated. Offutt and Lee [22, 30] used a range of techniques to examine the efficacy of weak mutations. They concluded that this approach provides stronger mutations at a lower cost. Researchers suggested comparing the internal states of the main program and its mutations following the first execution of the altered expression or the main block that contains it based on the results of the experiments. Table 1 displays the salient characteristics of the previously proposed strategies.

3 Identifying and Mutating the Bug Prone Paths

Software mutation testing is time- and money-consuming, therefore recent research efforts have focused on finding a solution. The fundamental objective of this kind of study is to reduce mutations while retaining efficacy. In this study, we provide a method for mutation testing that makes use of a forest optimization algorithm (FOA). In this process, mutation operators were only applied to the codes of the identified fault-prone regions of the program source code. The recommended approach stops mutation of the program's non-bug-prone sections, which significantly lowers the number of mutants. Figure 1 shows the proposed approach. The developed FOA takes a subset of CFG's paths as input (initial population). The size of the initial population (number of selected paths) depends on the number of total paths in the CFGs of the input program. Figure 3 shows the structure of the created CFG and the structure of a test path (solution) in the proposed method. Each solution (test path), which represents a tree in FOA, is implemented by an array. The suggested FOA is used to find out the most bug-prone paths (a subset of paths) of the program. The FOA begins with the initial population of paths (trees). The bug-proneness of the

Table 1 The related works proposed to reduce the number of mutants

The methods	Procedure	Merits	Demerits
Mutation sampling: (Budd [8]; Acree et al. [1]; Wong [34]; King and Offutt [21]; Wei et al. [33]; Arasteh et al. [4])	A subset of the generated mutations is selected.	The simplicity of conducting the test	Reduced test efficacy
Selective mutation, limited mutation: (Papadakis and Malevris [30]; Kintis et al. [22]; Wong [34]; Offutt et al. [28]; Jia et al. [19]; Offutt et al. [29]; Barbosa et al. [6]; Zhang et al. [36, 37]; Delgado et al. [10])	Selection of a small set of mutation operators	Maintenance of test effectiveness by reducing 65% of mutations	Poor performance and the requirement to combine it with mutation sampling
Minimum mutation sets: (Kintis et al. [22]; Malevris and Yates [26]; Kurtz et al. [23, 24]; Deng et al. [11]; Gheyi et al. [15])	Elimination of mutations	Small section of produced mutations is selected for covering the entire set	impreciseness
Strong, weak and hard mutations: (Kintis et al. [22]; Jackson and Woodward [18]; Howden [17]; Hosseini et al. [16], Yao et al. [35])	Weak mutation: By bypassing the full execution of the program, it lowers the number of mutations. Strong mutation: it reduces the number of mutations by contrasting the output of the original program with the output of the modified program. A mixture of strong and weak mutation is referred to as hard mutation.	Weak mutation is less expensive and uses fewer computing resources.	Weak mutations require evaluation and might be inaccurate if the entire program is not run.

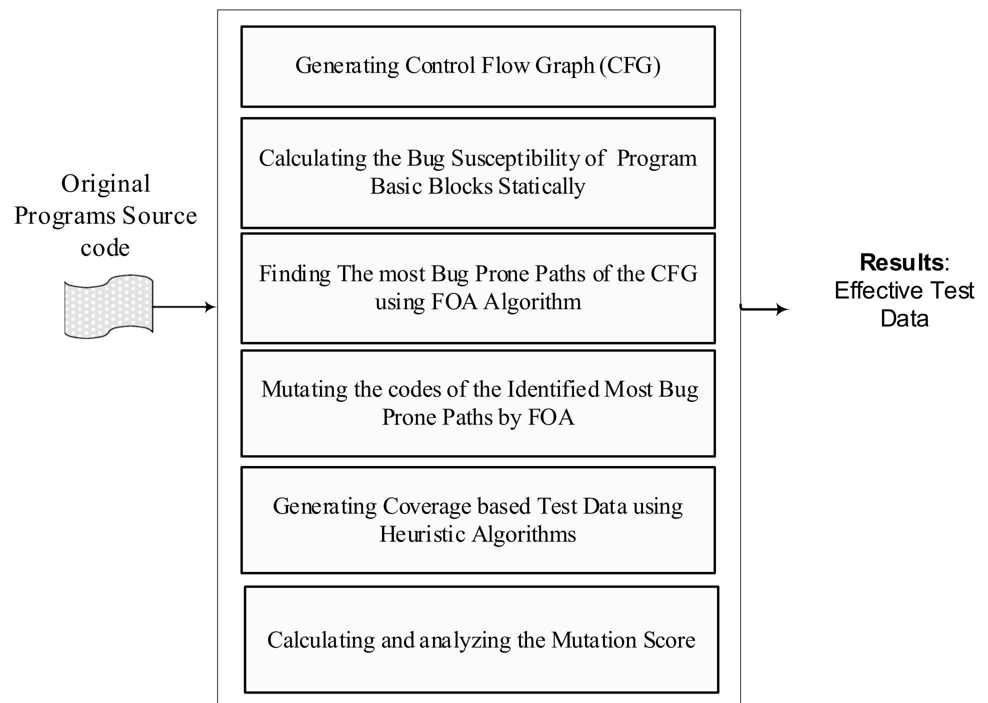


Fig. 1 An overview of the proposed method

paths in the populations is calculated by Eq. 7. The population's fitness value increases during iterations. The first population is the worst, and the final population includes the optimal (the most bug-prone) paths. The final population is the output of the suggested method. Indeed, the final population, as the most bug-prone path subset, is considered for performing mutation tests.

3.1 Control Flow Graph

While injecting mutation operators, the suggested technique identifies the program's most bug-prone regions. The related control flow graph (CFG) of the program source code should be constructed initially, as illustrated in Fig. 1. A CFG is a diagram that shows all of a program's potential pathways and branches. There are nodes and edges in the graph. Each node is described as a block that contains a collection of constantly performed operators and operands. In reality, if a single instruction in the block is performed, the entire block is executed. The presence of a directed edge between nodes suggests a probable graph execution path. The term "branch" refers to a node with more than one output edge. Figure 2 depicts the CFG of a program.

In this study, program complexity criteria were used to calculate the bug susceptibility of program blocks. In a CFG, a path's bug susceptibility (complexity) is determined by the complexity of its nodes. As a result, in the CFG, determining the weight of nodes (bug susceptibility measure of nodes) is necessary. The bug susceptibility of a path is a function of the complexity of the nodes and the sum of the complexity of the branches in the paths. Finding the bug-prone test path of a program is the main research problem of this study. Regarding the control flow graph of a program, finding the optimal test paths (test paths with the highest bug proneness) is an NP-complete optimization problem (Page 4, First Paragraph). Equation 1 was proposed to calculate the bug proneness of a test path. In a program with n branch instruction, there are 2^n test paths. The proposed method evaluates the error detection power of a software test method by injecting bugs only into error-prone codes. The modified and discretized version of FOA was developed to find out the most bug-prone paths of a program. At the first stage of the proposed method, the CFG of the input source code was automatically generated by visustin tool. The adjacency matrix of the CFG was generated automatically by the second module of the method. In the third stage, the FOA gets the adjacency matrix and generates a subset of paths (final population) as the most bug-prone test paths. Finally, Mujava was used to make the heuristic mutation test only on the selected bug-prone codes of the program.

Equation 1 represents the bug susceptibility of a path in a CFG. In this equation, α and β are fixed values whose values are considered 0.5. Measuring the bug susceptibility of blocks and branches in a CFG is explained in subsections 3.2 and 3.3. BB_j refers to normal basic block (BB) and BCH_k refers to branch type node (BB) in $path_i$. In Eq. 1, b_j specifies the number of non-branch nodes (BB) in the $path_i$. Similarly, r_k indicates the number of branch nodes in the $path_i$. Table 2 describes the variables used in this study.

```
public float function(int a, int b)
{
int x=0;
float sum=0;
int z=-1;
read(x);
read(z);
if(x>0)
  if(z%2==0)
       sum=a+(b*3);
  else
       sum=(a-b)/2;
else
   if(z%7==0)
       sum=a-1;
   else
       sum=(2*a)/b;
if (sum % 2==0)
    sum=sum + sqrt (a + b);
if ((sum % 5 ==0) and (b% 2 ==1)
    sum = sum-x;
else
    sum=sum-z;
return sum;
}
```

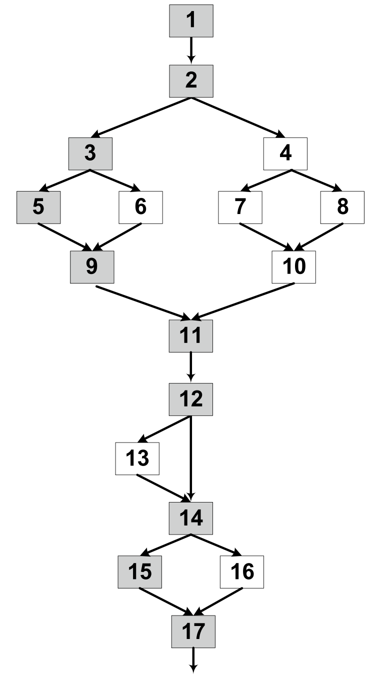


Fig. 2 A program source code and its generated CFG

$$Susceptability\ of\ Pathi = W(Path_i) = \sum_{j=1}^{|bj|} W(BB_j) \times \alpha + \sum_{k=1}^{|rk|} W(BCH_k) \times \beta \tag{1}$$

3.2 Block Susceptibility

This study uses basic block (CFG node) weight to evaluate the bug susceptibility (complexity) of a code basic block (BB). The larger the weight of a block, the more bug-prone it is. BB weight is measured by Eq. 2. The number of operators and operands in a BB influences its bug susceptibility.

$$W'\left(BB_i\right) = W'\left(N_i\right) + W\left(M_i\right) + \lambda \begin{cases} \lambda = 0.5, \text{ Node have } \textit{if} \text{ instruction.} \\ \lambda = 1, \text{ Node have not } \textit{if} \text{ instruction.} \end{cases} \tag{2}$$

The weight of operators in BBi is N_i; where N_i is the total number of operators in that node. The weight of operands in BBi is shown by M_i, which provides the total number of accessible operands in that node. The total weight ($W'\left(BB_i\right)$) is then calculated using each of their normalized weights. Operator weights were normalized

Table 2 Variable description

Variable Name	Description
CFG	Control flow graph of the program under test
$\mathbf{Path_i}$	The execution path i in the program control flow graph
$\mathbf{W(Path_i)}$	The bug proneness of a *pathi*
BBi	The normal basic block i in the program control flow graph which does not include the jump command
BCHi	The branch basic block i in the program control flow graph which include the jump command
W(BBi)	Weight of normal BBi
W(BCHi)	Weight of branch BCBi
b_i	The number of non-branch nodes (BB) in *pathi*
r_i	The number of branch nodes (BB) in *pathi*
α and β	Constant coefficients
$\mathbf{W'\left(BB_i\right)}$	Normalized $W\left(BB_i\right)$
h	Number of expressions in the branch statement

using Eq. 3, which divides the number of available operators in each node by the total number of operators in the corresponding path. We utilized Eq. 4, which divides the number of operands accessible at each node by the total number of operands in the path.

$$W'(N_i) = \frac{N_i}{\sum_{j=1}^{|b_j|} N_j} \tag{3}$$

$$W'\left(M_i\right) = \frac{M_i}{\sum_{j=1}^{|b_j|} M_j} \tag{4}$$

3.3 Branch Susceptibility

A branch block in a program's source code includes conditional statements and expressions. An expression in a branch block is a sequence of data and operators that evaluates a single value. An executable path of a program is selected based on the value of the respective expressions at run time. A branch's susceptibility is determined by the complexity weight of the respective expressions. A program path with a higher complexity weight is hard to comprehend by programmers and is bug-prone. In real-world programs, the hard-to-comprehend and hard-to-reach parts of a program may include more bugs (bug-prone parts). Hence, to simulate the real-world bugs in a program, the hard-to-reach codes of a program should be considered with higher priority in the mutation test. Expression weight (complexity) is calculated using Eq. 5 and Table 3 for each branch statement in the CFG. The following two stages are created by Eq. 5:

- If the relevant decision node contains h expressions that have been combined using the AND operator, the square root of the overall weight of the expressions is considered.
- The selection criteria will be the lowest weight of the expression weight if the associated decision node includes h expressions that have been merged using the OR operator.

Table 3 Operators' weight in terms of bug-prone used to compute the weight of expression

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

$$W\left(BCH_j\right) = \begin{cases} \sqrt{\sum_{g=1}^{h} W_r^2\left(C_g\right)}. & \textit{if conjuction is AND} \\ min\{W_r\left(C_g\right), 1 \le g \le h\}. & \textit{other wise} \end{cases} \tag{5}$$

In Eq. 5, C_g stands for the gth expression in the branch statement ($1 \le g \le h$). h shows the number of expressions in the branch statement. The Wr variable is the condition weight set by Table 3. For normalizing the expression weight of branch statements, we employed Eq. 6, which divides the expression weight of each branch by the overall weight of the branches. Table 3 also lists all the operators that might be included in the condition expressions. Equation 7 represents the objective (fitness) function.

$$W'\left(BCH_i\right) = \frac{W\left(BCH_j\right)}{\sum_{j=1}^{|r_i|} W\left(BCH_j\right)} \tag{6}$$

$$Fitness\left(Path_{pi}\right) = \sum_{i=1}^{|b_i|} W'(BB_i) \times \alpha + \sum_{j=1}^{|r_j|} W'(BCH_j) \times (1-\alpha) \tag{7}$$

The phrase $\sum_{i=1}^{|p_i|} W'(BB_i)$ describes the total complexity of nodes in a program path ($path_i$), where b_i specifies the number of non-branch nodes in the path_i. Similarly, the phrase $\sum_{j=1}^{|r_j|} BCH_j$ defines the total complexity of branch (decision) nodes, where r_i is the number of branch nodes (decision node) in the program path. In Eq. 7, α was employed as an impact factor and as a criterion for the efficacy degree of complexity. The value of α is 0.5 in this study.

3.4 Finding Bug-Susceptible Paths Using Forrest Optimization Algorithm (FOA)

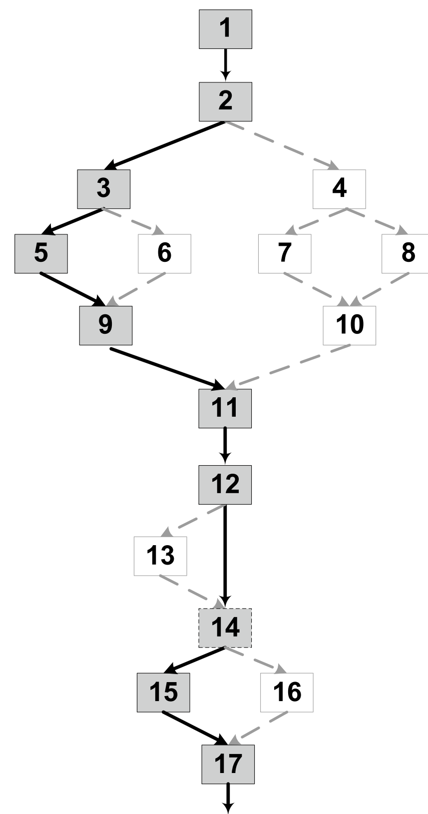
3.4.1 Modeling the Research Problem in the FOA

The third stage of the proposed method, as shown in Fig. 1, is a search optimization problem; also, finding the most susceptible paths in a control flow graph is an NP-complete problem. In this research, the forest optimization algorithm (FOA) is suggested to identify the most susceptible paths in a program source code. As explained in subsection 3.3, the bug-susceptibility of each block should be calculated using Eqs. 2–6 before executing the FOA. FOA takes the calculated weights of each block and then finds the most bug-susceptible paths of a program source code. To reduce

the cost of software mutation testing, only the identified susceptible paths of the program are considered for mutation. The base form of FOA was proposed by Manizheh and Mohammad-Reza [27]. The basic forest optimization algorithm is a continuous heuristic algorithm that can be used to solve continuous NP-hard problems. On the other hand, finding the bug-prone test paths of a program source code is an NP-compete discrete problem. The basic form of the FOA cannot be adapted directly to the software testing problem. The other challenge of the basic FOA is its local optima problem in the test optimization problem. The lower success rate and low stability are the main problems of the basic FOA. In this study, the basic FOA was modified and discretized. As shown in Fig. 3, each individual (tree) is implemented by an integer array that shows a test path in a graph. Furthermore, to increase the diversity among the individuals and to avoid the local optimum, a version crossover has been used between the best and worst trees in a specific iteration. The results of experiments indicate that the modified and discretized FOA has higher performance in the software test problem.

The customized and modified form of FOA is used in this study. Similar to other heuristic algorithms, the FOA takes a subset of CFG's paths as input (initial population). The FOA process begins with the initial population of trees; each tree denotes a potential solution to the problem (a path in the CFG). Each CFG path, which represents a tree, is implemented by an array. The array's length represents the path length. Figure 3 shows a tree structure in the proposed FOA. In the tree array, Nvar is equal to the length of the path in the CFG and Age is equal to the age of the tree, and Cost is the bug-susceptibility of the tree (path) that is calculated via Eq. 7. The initial population is represented as a matrix where each row is a tree array (Fig. 3). The proposed FOA includes three basic steps: local tree seeding, population constraint, and global tree seeding. Figure 4 shows the general form of FOA.

The age of the tree is regarded as zero in the FOA when it begins to work. The local seed operator grows new trees from the young trees in the forest. Then, with the exception of newly generated trees, all trees rise in age from 0 to 1. After that, control over the forest's tree population eliminates part of the trees. One percent of the candidate population is chosen to travel around the forest during the global seeding stage. Now the forest trees are ordered by their bug-susceptibility values (calculated by Eq. 7); the tree with the greatest bug-susceptibility (most fitted) value is chosen as the best tree, and its age is set to zero to prevent re-selection. These steps continue until the stop condition is obtained.



Age	1, 2, 3, 5, 9, 11, 12, 14, 15, 17	Cost

A Randomly Generated Tree as a Execution Path

Fig. 3 Tree structure in FOA

3.4.2 Forest Initialization and Local Seeding

The effect of tree age is that if a tree was optimal in terms of bug-susceptibility, the local seed operator would add trees similar to the optimal tree to the population, and optimal trees would remain in the population. But if the tree is not optimal, the trees produced from this tree will be removed from the population after several repetitions due to old age or limited forest width. A predetermined parameter called the lifetime determines how old a tree may be. At the start of the algorithm, this parameter must be changed. When a tree reaches the end of its life, it is cut down and added to the candidate population. If this option is set to a large value, each iteration of the algorithm will only grow this tree, resulting in a forest full of elderly trees that don't participate in local seeding. If we set this parameter to a low value, the trees will quickly age and be eliminated from the

```
1.  Discretized Forest Optimization Algorithm (FOA)
2.  {
3.  //Input: subset of test path from CFG
4.  //Output: The most bug-prone test path as final population
5.  //Each tree in FOA represents a test path
6.  Initializing the ForestPop with 0 aged trees;
7.  iter=1;
8.  While (iter < max-iter) do
9.   {
10.  i=1;
11.    while (i<PopSize)
12.     {
13.         if(age(Treei > lifeLimit)
14.            Append (CandidateTree, Treei);
15.         i=i+1;
16.     }
17.    ForestPop=sort(ForestPop);
18.    If(PopSize > AreaLimit)
19.         Append(CandidateTree, AreaTree);
20.    ForestPop=globalseeding(ForestPop);
21.    BestTree.age=0;
22.    iter=iter+1;
23.   }
24. Return ForestPop; //final pop as the most bug-prone test
    paths
25. }
```

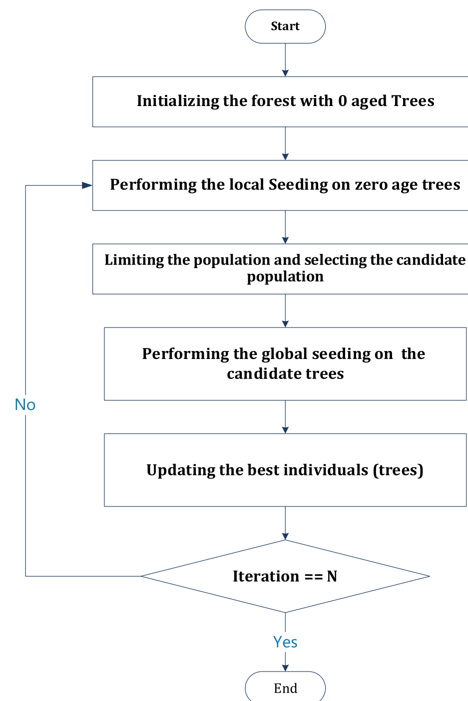


Fig. 4 The pseudo-code and flowchart of the developed FOA

forest population. As a result, depending on the application, this value should be changed.

Local seeding is done on trees that are less than a year old and adds some trees from the surrounding area to the forest. The figure shows two repetitions of this operator (Fig. 5). Except for newly generated trees, the age of all trees grows by one unit after local sowing in zero-age trees. The algorithm approach is implemented utilizing the local seeding step if a tree is promising (increased bug susceptibility). Otherwise, the hopeless trees (lower bug susceptibility trees) age and die naturally after a few cycles. The number of seeds that fall near a tree (CFG's

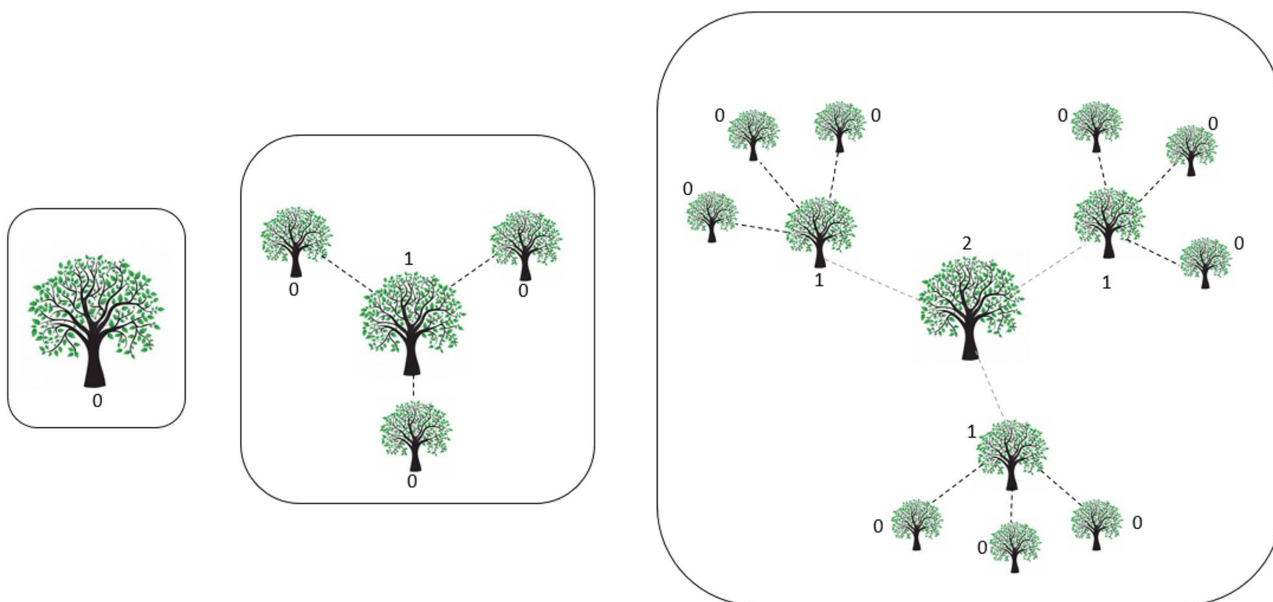


Fig. 5 Local seeding operation in FOA

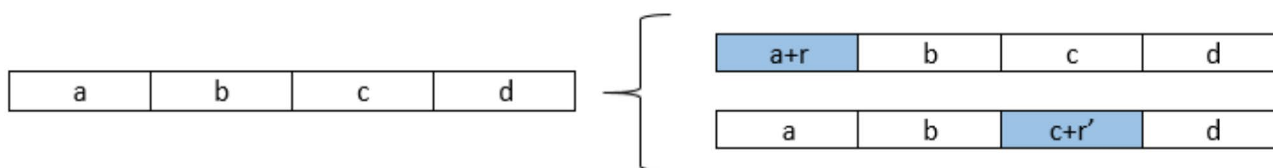


Fig. 6 Local seeding example for a tree when LSC = 2

path) and then form a tree is a FOA calibration parameter, as illustrated in Fig. 5. A local seeding change (LSC) is a parameter. In Fig. 5, the value of this option is set to 3. The application scope dimension should be used to determine this parameter. We experiment to determine the ideal LSC parameter values (explained in Section 4). As a result, three trees have been put into the forest for each tree with an age of 0. Figure 6 illustrates an example of a local seeding operator for a tree (CFG's path) with an LSC value of 2. The values of *r* and *r*' (values generated randomly) are in the interval $[-\Delta X.\Delta X]$.

3.4.3 Population Limitation and Global Seeding

The number of trees in the forest needs to be managed to prevent unrestrained forest installation. Two factors, life span, and geographical restriction, limit the number of trees. The trees that go through the tree's life cycle must first be eliminated from the forest since they will eventually represent the population. The second restriction, the area constraint, is determined by fitness (bug-susceptibility); this constraint is handled in such a way that if the number of trees exceeds the forest constraint, many trees are destroyed. Another parameter called area constraint is forest constraint. In the experiments of this study, the constraint values of the area were considered the same number of primary trees. Therefore, after doing this, the number of trees in the forest is equal to the number of primary trees.

The global seeding operator seeks to imitate tree seed dispersion in the forest. In the search space, this operator does a global search. In the previous stage, the global seeding operator was specified as a percentage of the candidate trees. Figure 7 shows an example of a tree's global seeding procedure (program path). The tree is chosen at random from the candidate population; after that, certain elements in each tree are chosen at random. At this point, each element's value is created using a random output value within a reasonable range. Another FOA parameter is the number of these elements that vary, which is described as global seeding variation (GSC). The GSC parameter is assumed to have a value of 2 in Fig. 7. As a result, two variables are chosen at random and their values are swapped with other randomly produced values in the relevant variables' range, such as r and r'. The trees should be classified depending on fitness once they have been seeded globally (bug susceptibility). The best tree is chosen by sorting the trees and selecting the one with the highest fitness. To re-select in the local seeding stage, the age of the best tree is set to zero. The finest tree in this scenario might be selected from a local seed stage. Because, as previously said, local seeding is performed on trees that are less than ten years old.

4 Evaluation

4.1 Experiment Platform and Benchmark Programs

An extensive series of experiments have been implemented to evaluate the method. The proposed method has been implemented in the MATLAB 2020 programming language. The tests were run on a PC with an Intel Core i7 CPU, 8 GB of RAM, and Windows 10 operating system. After selecting the most bug-prone paths of the benchmark programs by the proposed FOA, the mutants were injected by the MuJava tool (https://cs.gmu.edu/~offutt/mujava/). A set of standard and most frequently used programs have been selected as benchmarks. The selected programs have been used in previous studies as evaluation benchmarks. Table 4 explains the characteristics of the benchmark programs. These programs include the structures that are used in real-world programs, such as loop and conditional structures, I/O instructions, and different arithmetic, and logical operators. Also, the selected programs include nested programming structures. Today's real-world programs with millions of lines of code are written in modular form. Indeed, all the codes are not written in one function (unit) but are divided into functions

Fig. 7 Global seeding example for a tree (program path) when GSC = 2

Table 4 Characteristics of the benchmark programs

Program name	Num. Of Input	Code lines (LOC)	Cyclomatic Complexity	Time Complexity	Program Output
Triangle Type	3	31	20	O(c)	Specifying triangle type
Binary Search	3	75	8	O (log n)	Searching from an integer list
Quadratic Equation	3	30	5	O(c)	Calculating the roots of quadratic equations
Number of Digits	1	39	5	O(n)	Calculating the digits of integer number
Largest Num	3	32	7	O(n)	Finding the largest integer number

and modules. Regarding the programming standard styles, the code written into a function should be between 5 and 60 lines of code. Indeed, these functions are considered the test unit. The functions with hundreds of lines of code are not standard and understandable and should be broken to smaller-size functions. Therefore, in a real project, we face a set of functions (units) that must be tested. To evaluate the proposed method, a set of standard functions was selected. Similar to real-world programs, the selected benchmark programs include 30 to 75 lines of code. The selected benchmark programs include different arithmetic and logical operators, branch instructions (if-else instructions), loop structures, and different types of data. Concerning the Halstead and Cyclomatic metrics, the selected programs are hard to test. The number of branch instructions and operators in the selected benchmark programs is higher than in real-world application programs. For example, the *triangle type* benchmark includes 31 lines of code, but each line includes an if structure. As a result, testing the selected benchmark programs with 100% coverage is hard and time-consuming.

Performance Criteria that were used to evaluate the proposed method are as follows:

- **Success Rate**: Heuristic algorithms work based on probability, so the results of one execution are not enough to assess their success rate. The success rate shows the probability of success of the proposed method in producing the optimal solution. In this study, the success rate of the proposed FOA was evaluated during 10 times. The percentage of times that the heuristic algorithm can produce the optimal solution indicates the success rate of the algorithm. The success rate of the proposed method was explained in subsection 4.2.3.
- **Mutant Reduction**: Mutation test is used to evaluate the effectiveness of a test technique or tool. The mutation score indicates the effectiveness of the test data. The mutation score is the percentage of the detected bugs that were injected by the mutation test tool. The cost and time of the mutation test methods depend on the number of injected mutants (bugs) by the mutation test method. The proposed method injects mutants only in the bug-prone codes of the program. The rate of reduction of ineffective mutants is one of the important criteria that was discussed in subsection 4.2.4.
- **Stability**: Stability shows the closeness of the generated results to each other. The stability of an algorithm means that the results are not subject to specific conditions or are not obtained by chance. The algorithm will be stable when the difference in the fitness of the final results generated in different executions is not noticeable. The stability is evaluated by calculating the standard deviation among the results obtained during 10 executions. The stability of the proposed mutation test method was discussed in subsection 4.2.2.
- **Convergence Speed**: One of the evaluation criteria of heuristic algorithms is the convergence to optimal response. During the execution of the heuristic algorithms, the results should be gradually improved in terms of fitness and converged to the optimal solution. The convergence shows how well FOA does at identifying bug-prone codes of the program's source code. The convergence speed of the proposed method was discussed in subsection 4.2.1.

As shown in Fig. 1, the source code of the input program (function) source code should be converted to the CFG. This stage of the method is performed automatically using existing tools such as Visustin (https://www.aivosto.com/visustin.html). Figure 8 shows the generated CFG for the *triangle* benchmark program. After extracting the CFG by the software tool, the CFG is also automatically converted to an adjacent matrix by the implemented code in MATLAB. Table 5 shows the generated adjacent matrix for the *triangle* benchmark. At the next stage, the implemented FOA as a module in the MATLAB is invoked. The invoked FOA finds out the bug-prone paths of the program during the determined iterations. Overall, all stages of the proposed method are performed automatically, and the method was implemented as a software package to perform automatically. Table 6 shows the calculated node weight for all nodes of the triangle CFG (shown in Fig. 8).

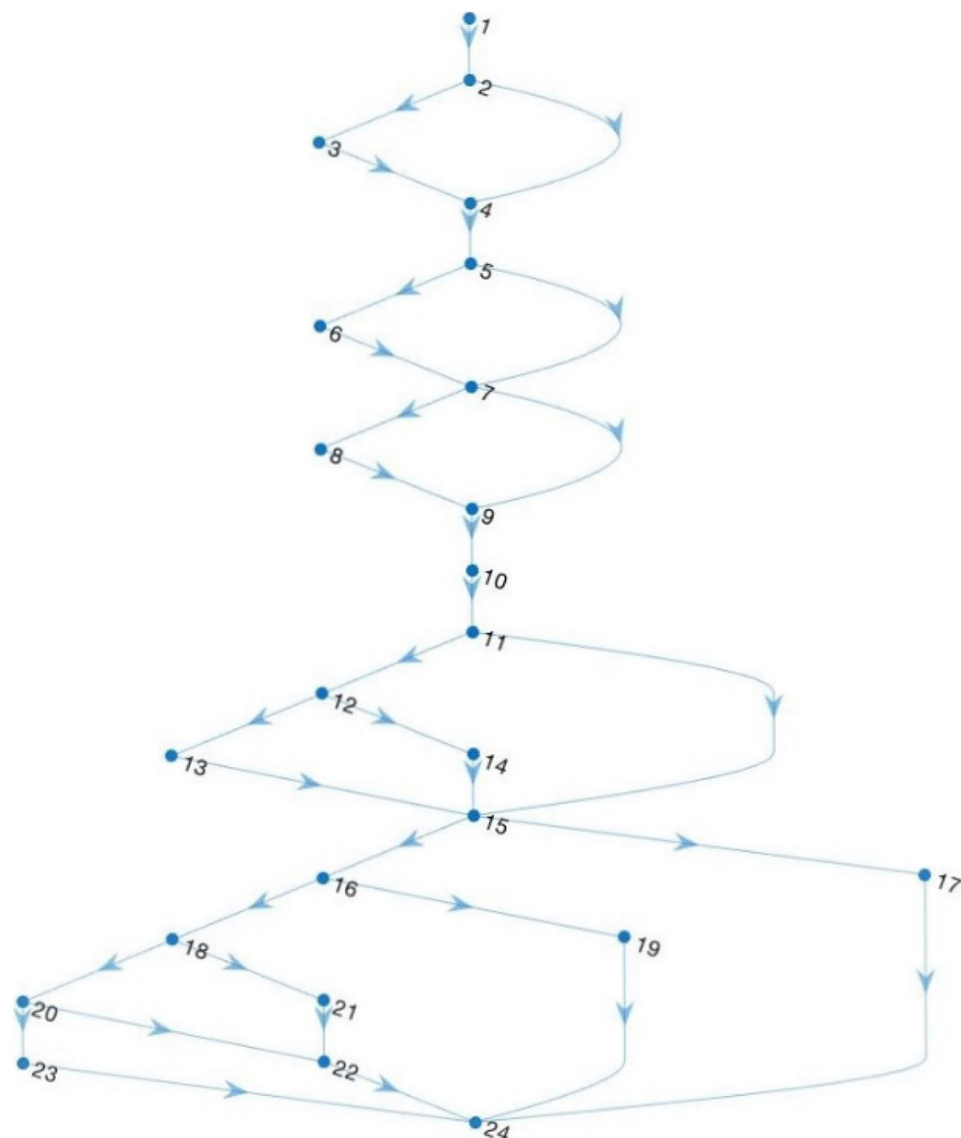


Fig. 8 The generated CFG for the triangle benchmark by the *visustin* tool

After identifying the bug-prone paths of the benchmark programs by the FOA, the mutation stage is performed on the selected paths. To this end, MuJava is used to make the function-level mutants (buggy versions of the original programs). Table 7 depicts the function-level mutation operators in MuJava that are used to simulate the programming bugs in the bug-prone codes of the programs (selected by FOA). In stage 5 of the proposed method, a set of test suits are required to evaluate the proposed method in terms of mutation score. Indeed, the generated mutants by the MuJava should be executed using a test suit. The branch coverage criterion is used to generate effective test data for the benchmarks. Traxtor [3], was used for the test-generation stage of this study. This method (Traxtor) is invoked as the other MATLAB module to automatically generate the coverage-based test data. Finally, the calibration parameters of the FOA and genetic algorithm have been calibrated experimentally and are described in Table 8.

4.2 Results and Discussion

4.2.1 Convergence Criterion

One of the evaluation criteria of heuristic algorithms is the convergence to optimal response. Therefore, during the implementation of the heuristic algorithms, the results should be gradually improved and converged to the optimal response. The convergence depicts the performance of the heuristic algorithms. The convergence in this study shows how well a heuristic algorithm does when it comes to identifying the most bug-prone regions in a program's source code. The convergence diagram depicts the fitness (bug-susceptibility) function's optimal value across iterations for the proposed approach. The population's fitness value increases during iterations in the best-fit convergence diagram. The ideal solution is finally found using

Table 5 The generated adjacent matric for the *triangle* benchmark program

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

this convergence diagram (most bug-prone paths). The reason the chart is staggered during different performances is that the improved individuals (trees) always replace the worse individuals. Figure 9 shows the convergence (performance) of the FOA and GA in finding the most bug-prone paths in the binary-search benchmark program. The genetic algorithm converged to 3.94 at 111 iterations. While the forest algorithm has converged to 4.06 with a maximum of 200 iterations. FOA finds the optimal solution (a path with higher bug susceptibility). The fitness of the best path found by FOA is 4.09 which is higher than the path found by GA. As a result, FOA is more capable of finding bug-prone paths in this benchmark. Also, as shown in Fig. 9, the convergence speed of the FOA is higher than the GA.

As shown in Fig. 10, although the GA had a good start compared to the FOA; then, the proposed algorithm (FOA) has converged significantly faster than the GA, although they are both identical in value. Overall, the FOA has better performance than GA in terms of convergence. Figure 11, like the previous convergence diagrams, has an evolutionary trend. It can be seen that the FOA has converged in lower iterations than the genetic algorithm. Hence, the FOA has better performance than the GA in several *digits* benchmark in terms of convergence. The other experiments have been performed on the *quadratic equation* benchmark; FOA and GA have been executed to find out the bug-prone paths of this benchmark program. Figure 12 shows the results of this experiment. The results show the superiority of the FOA over the GA because it converged faster than GA. FOA attains the maximum value of fitness (4.48) in iteration 26 while the GA achieved the same value in replication 192.

Finally, in Fig. 13, which is related to the triangle benchmark, it is observed that the FOA is more powerful than the GA in terms of convergence. Also, FOA outperforms GA concerning convergence speed. Overall, concerning the convergence criteria, FOA has a higher performance.

Figure 14 shows the average results obtained by FOA and GA. Each algorithm (FOA and GA) has been executed 10 times on each of the benchmark programs. Each execution includes 200 iterations. The fitness (bug-susceptibility) of final results (CFG's paths) obtained in each execution has been used to calculate the average value. As shown in Fig. 14, the FOA has better performance in terms of average fitness, specifically in the *triangle* benchmark. The *triangle* benchmark has higher cyclomatic complexity than the other benchmarks. Indeed, FOA has a higher bug-prone finding capability than GA, specifically in complex programs.

Table 6 The weight of each node in the CFG of the *triangle* benchmark calculated by Eq. 2

Node	N1	N2	N1(Normalized)	N2(Normalized)	N1 + N2(Weight)	N1 + N2 + λ	BCH Weight
1	0	3	0.00	0.05	0.05	1.05	0
2	0	1	0.00	0.02	0.02	1.02	0
3	6	6	0.13	0.10	0.23	0.73	0.1
4	0	1	0.00	0.02	0.02	1.02	0
5	0	1	0.00	0.02	0.02	1.02	0
6	2	2	0.04	0.03	0.08	0.58	0.11
7	2	3	0.04	0.05	0.09	1.09	0
8	2	2	0.04	0.03	0.08	0.58	0.13
9	2	3	0.04	0.05	0.09	1.09	0
10	2	2	0.04	0.03	0.08	0.58	0.15
11	2	3	0.04	0.05	0.09	1.09	0
12	2	2	0.04	0.03	0.08	0.58	0.16
13	9	9	0.20	0.15	0.34	0.84	0.2
14	2	2	0.04	0.03	0.08	0.58	0.2
15	0	1	0.00	0.02	0.02	1.02	0
16	0	1	0.00	0.02	0.02	1.02	0
17	0	1	0.00	0.02	0.02	1.02	0
18	5	5	0.11	0.08	0.19	0.66	0.3
19	0	1	0.00	0.02	0.02	1.02	0
20	5	5	0.11	0.08	0.19	0.69	0.32
21	0	1	0.00	0.02	0.02	1.02	0
22	5	5	0.11	0.08	0.19	0.69	0.34
23	0	1	0.00	0.02	0.02	1.02	0
24	0	1	0.00	0.02	0.02	1.02	0
25 (Final instruction)	0	0	0.00	0.00	0.00	1	0

Table 7 MuJava mutation operators used in the mutation step of the proposed method

Operator	Description
AOR	Replacing the arithmetic operator in the code
AOI	Inserting an arithmetic operator in the code
AOD	Deleting an arithmetic operator from the code
ROR	Replacing a relational operator in the code
COR	Replacing a conditional operator in the code
COI	Inserting a conditional operator in the code
COD	Deleting a conditional operator from the code
SOR	Replacing a shift operator in the code
LOR	Replacing a logical operator in the code
LOI	Inserting a logical operator in the code
LOD	Deleting a logical operator from the code
ASR	Replacing an assignment operator in the code
SDL	Deleting a statement from the code
VDL	Deleting a variable from the code
CDL	Deleting a constant from the code
ODL	Operator Deletion

4.2.2 Stability Criterion

Because the initial population is produced randomly and because these algorithms use random operators, the outcomes achieved from each heuristic method may change in

Table 8 Calibration parameters of FOA and GA

Method	Parameters	Values
FOA	25% of all paths in the CFG	Initial Population
	Lifetime	10
	Area limit	100
	LSC	8
	GSC	8
	Transfer rate	%3
	Iteration	200
	Dimension	15
GA	Initial population	25% of all paths in the CFG
	Pc	0.8
	Pm	0.3
	Iteration	200

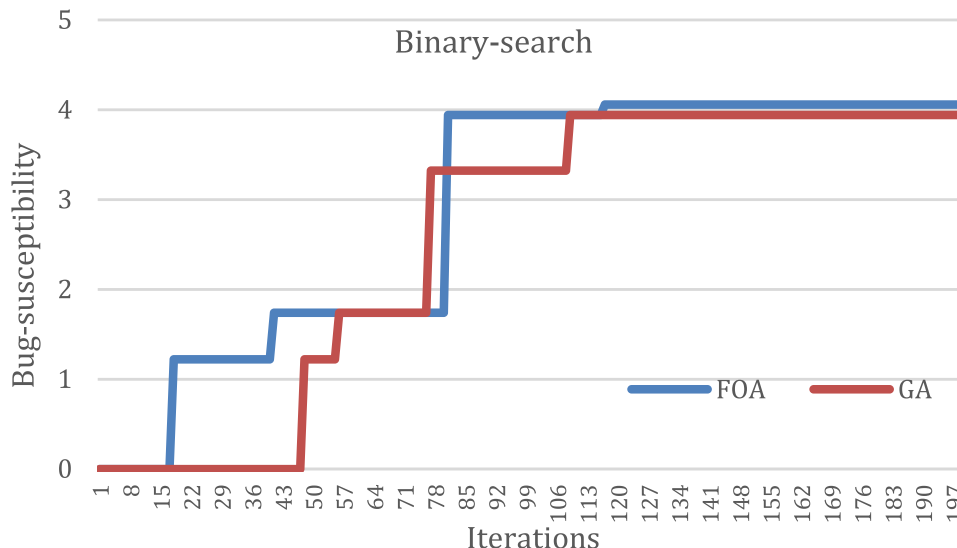


Fig. 9 Convergence of the FOA and GA in finding the bug-prone paths of *binary-search* benchmark

different executions. As a result, one of the most significant characteristics to consider while assessing a heuristic algorithm is its stability. The stability indicates the closeness of the different results (obtained from different executions of a heuristic algorithm) to each other. The stability of an algorithm means that the results are not subject to specific conditions or are not obtained by chance. The algorithm will be stable when the difference between the final values of the fitness function in different executions is not noticeable. This will be confirmed by calculating the standard deviation of different values of the fitness function. The amount of standard deviation is inversely related to the stability of the algorithm. Therefore, the smaller the standard deviation, the more stable the algorithm is. To this end, each algorithm has been executed 10 times, and each execution includes 200 iterations. Figure 15 depicts the bug-susceptibility of obtained results (CFG's path) by FOA and GA in 10 executions. Each execution includes 200 iterations. The lower the variance among the obtained 10 results, the higher the stability of the algorithm in different executions. The standard deviation of these results was calculated and shown in Fig. 16. Figure 16 shows the standard deviations among the obtained results. Based on the results of these 10 executions, for each benchmark program, the standard deviation of FOA is less than the GA. This is one of the reasons for the stability of FOA in finding the bug-prone path of a program.

4.2.3 Success Rate Criterion

Another performance criterion for heuristic algorithms to consider is the success rate. The stability revealed the similarity of distinct fitness result values from different

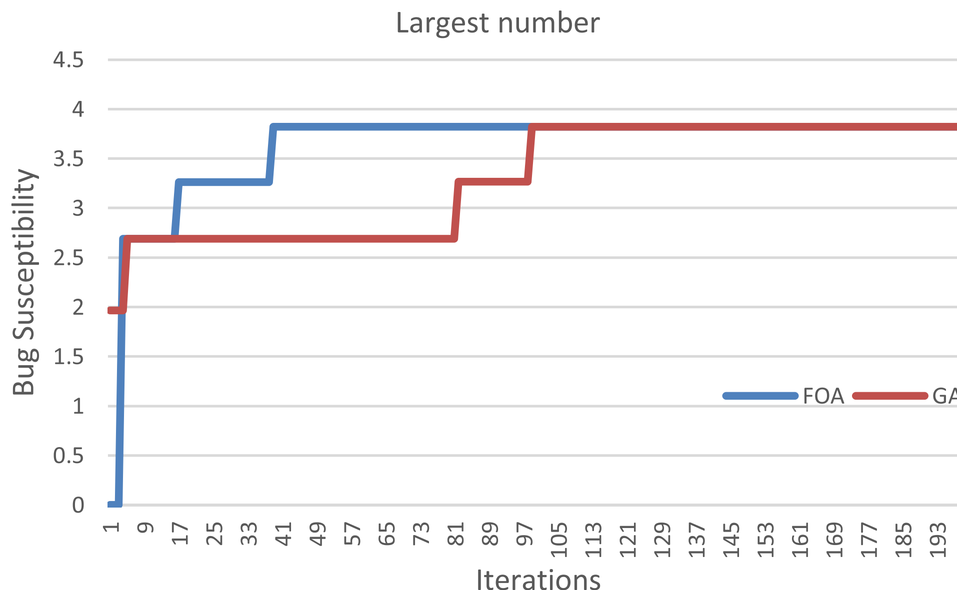


Fig. 10 Convergence of the FOA and GA in finding the bug-prone paths of *largest-number* benchmark

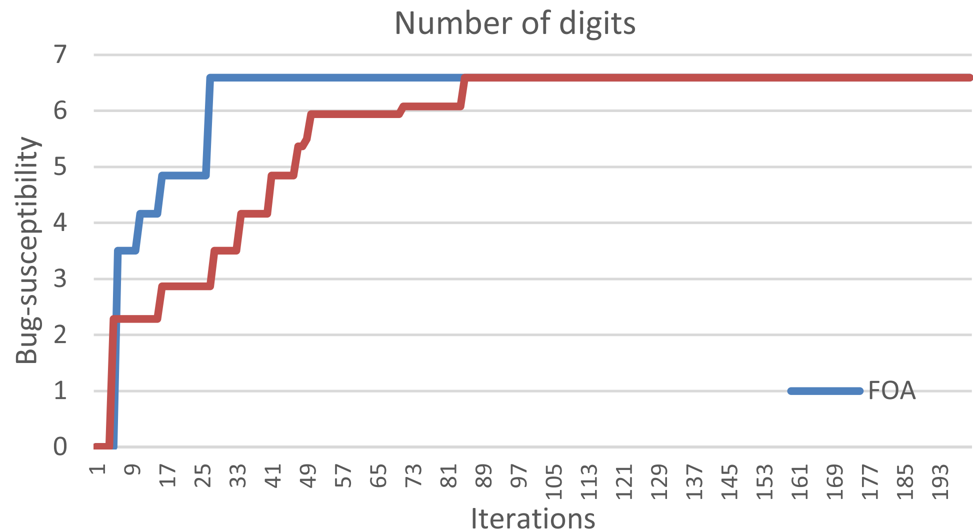


Fig. 11 Convergence of the FOA and GA in finding the bug-prone paths of *number of digits* benchmark

executions. The success rate of a heuristic algorithm, on the other hand, is the capacity of the algorithm to determine the optimal (best) value. In other words, it's the degree to which the acquired fitness value is similar to an ideal value. This criterion is determined by dividing the total number of times the algorithm has been run using the suitable benchmark program by the number of times the fitness function value has reached its maximum value. With 10 runs, the results displayed in Fig. 17 confirmed that the FOA outperformed GA in terms of success rate. In almost 80% of cases, the algorithm can find the bug-prone paths of programs. Overall, the probability of the FOA finding the best results (most bug-prone) paths of a program source code is close to 80%.

4.2.4 Mutant Reduction

The mutation score is a useful metric for evaluating the effectiveness of a test suite. Tables 9 and 10 display the total number of mutants produced by the suggested method for all paths in each benchmark program. The recommended approach reduces the number of mutations. The suggested method uses the FOA algorithm to locate the bug-prone paths in the source code before performing mutation operations. As a result, the suggested method decreases the number of mutants by removing mutant injection in the program's non-bug-prone paths. According to studies done on common benchmark programs, the proposed method reduces 27.63% of the created mutants when compared to existing methodologies. The cost of mutation testing will go down if the quantity of created mutants is decreased. The method outlined in this paper may allow the use of common mutation testing tools (Mujava, Muclipse, Jester, and Jumble) to do mutation testing at a lower cost.

Concerning the results of experiments shown in Fig. 8, the bug-prone aware mutation test requires a lower number of mutants. The lower the number of generated mutants,

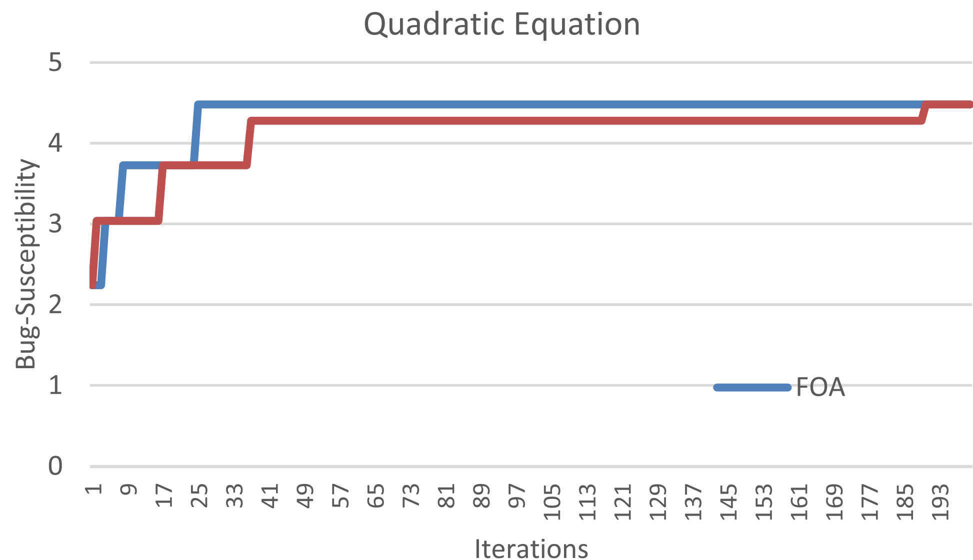


Fig. 12 Convergence of the FOA and GA in identifying the bug-prone paths of *Quadratic equation* benchmark

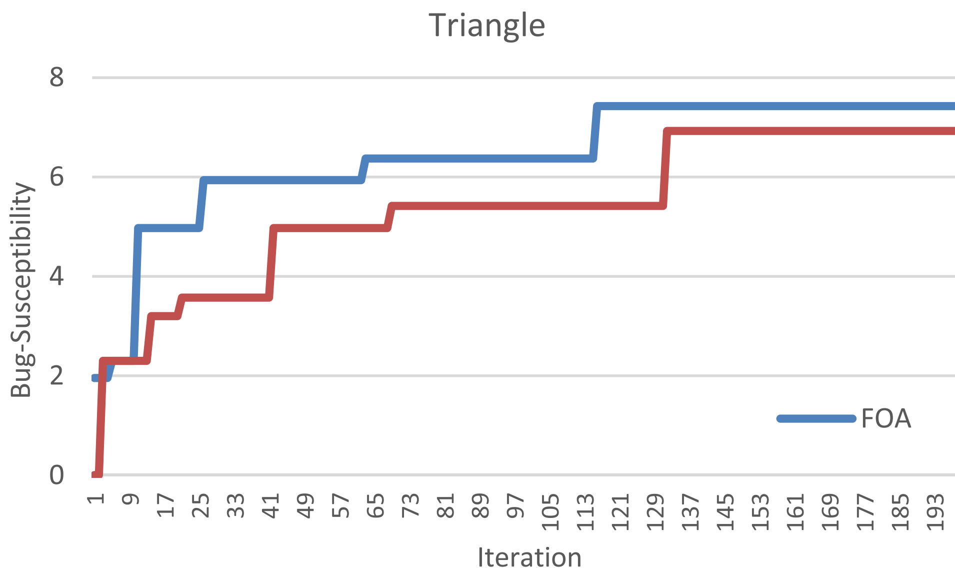


Fig. 13 Convergence of the FOA and GA in identifying the bug-prone paths of *triangle* program

the less time and cost. Performing the mutation test with a limited number of generated mutants is more efficient in terms of time and cost.

4.2.5 Calibrating the FOA Parameter

The behavior of the heuristic algorithms depends on different parameters. The calibration parameters of the FOA should be adapted regarding the problem features and its applications. The authors adapt the FOA parameters regarding the benchmark programs during the experiments. LSC, GSC, and transfer rate are the main parameters of the FOA algorithm that should be calibrated experimentally. To this end, the experiments have been repeated with different values of GSC (8, 6, 4, and 2). Figure 18 shows the effect of GSC on the performance of the FOA in the field. The value of GCS influences the fitness of the obtained results by the FOA. As shown in Fig. 18, the optimal results (the paths with maximum bug susceptibility) have been identified when the GSC = 8. Figure 19 shows the effects of the LSC, as the other calibration parameter, on the performance of the FOA in finding the most bug-prone paths of a program. The best results were obtained when LSC = 8.

Figure 10 shows the most bug-prone path of each benchmark program identified by the proposed method. The best (most bug-prone) path of the final population that is generated by the method is shown in Table 11. This path, along with the other paths of the final population, is considered for the mutation test.

The proposed method is independent of the platform and tools used for mutation tests. To this end, the proposed method was used with different mutation test tools. Pitest, Muclipse, MuJava, Jester, Jumble, and JavaLancer are the most frequently used tools for Java programs. In the

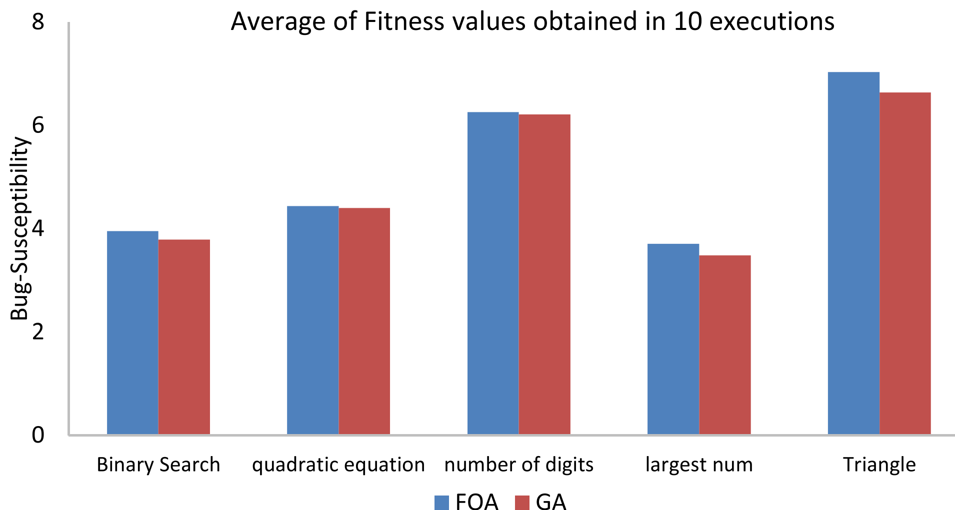
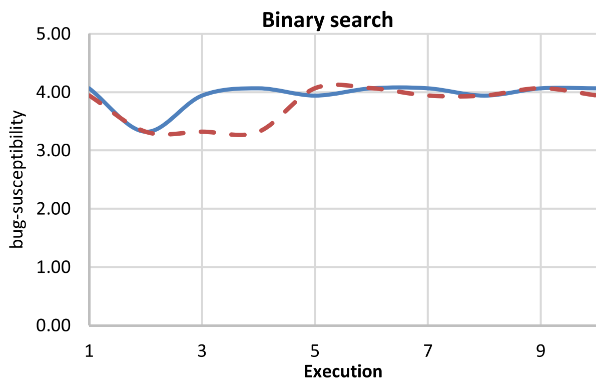
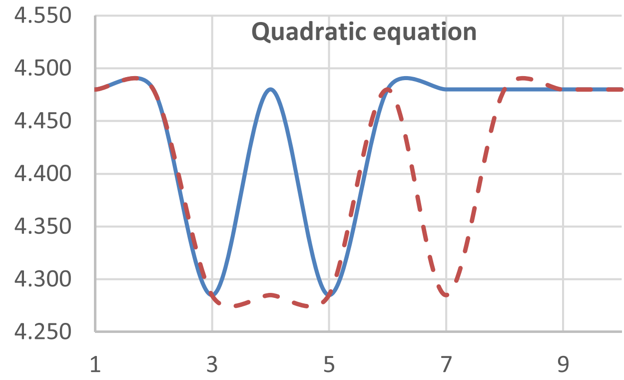


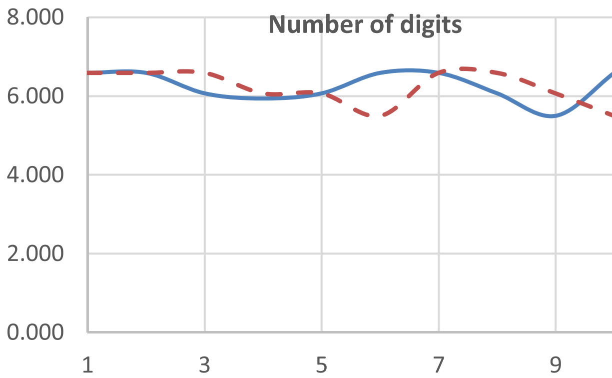
Fig. 14 The average of obtained fitness values in 10 executions of FOA and GA



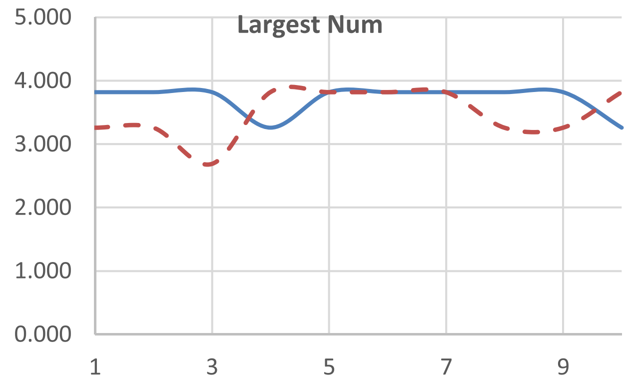
a) Obtained results by FOA and GA for *Binary search* program



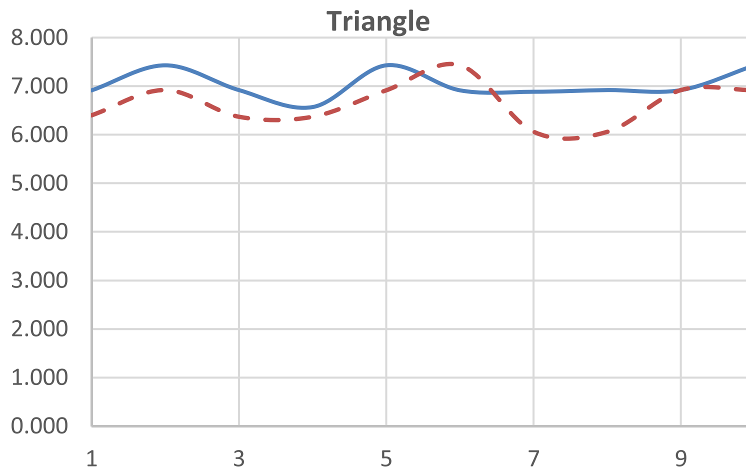
b) Obtained results by FOA and GA for *Quadratic equation* program



c) Obtained results by FOA and GA for *Number of digits* program



d) Obtained results by FOA and GA for *Largest num* program



e) Obtained results by FOA and GA for *Triangle* program

FOA Genetic

Fig. 15 Standard deviation among the fitness (bug-susceptibility) values of the obtained results from 10 executions of each algorithm

final series of experiments, the benchmark programs were mutated by different mutation-test tools. The number of generated mutants for each benchmark program with and without the proposed method is shown in Figs. 20, 21, and 22. The results confirm that the proposed method makes a considerable reduction in the number of generated mutants by all testing tools. In the *triangle* benchmark program, the average number of generated mutants in all tools is about 160; meanwhile, the average number of generated mutants for *triangle* program with the proposed method is about 107.

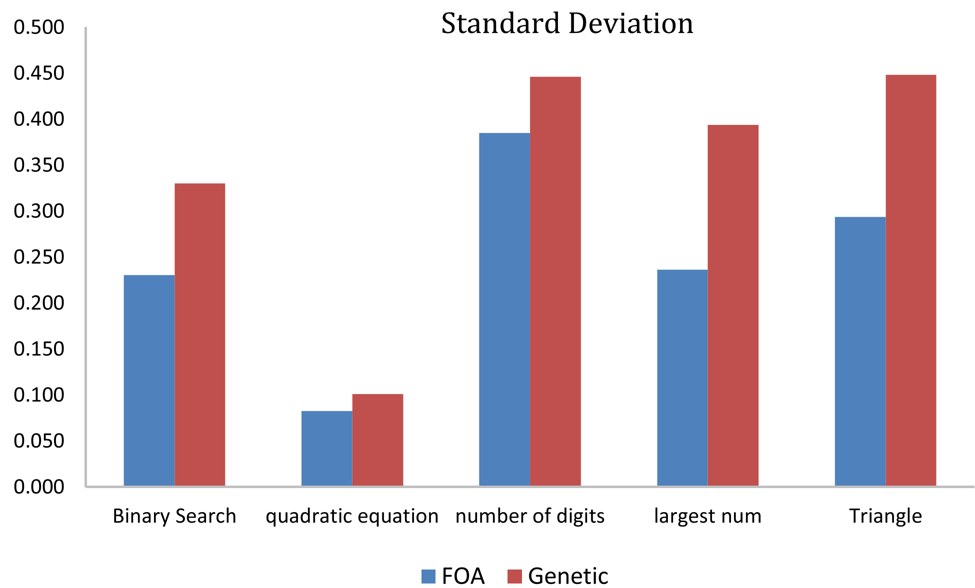


Fig. 16 Standard deviation among the fitness (bug-susceptibility) values of the obtained results from 10 executions of each algorithm

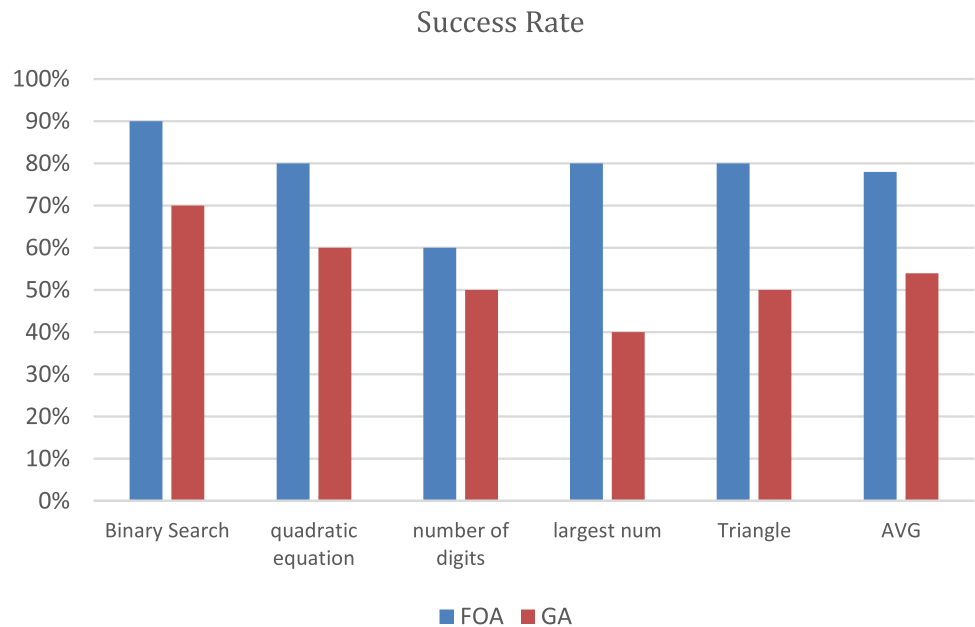


Fig. 17 The probability of FOA and GA algorithms in finding the most bug-prone paths in different benchmarks

Table 9 The average number of generated mutants with and without proposed method

Programs		Total Mutants	Killed Mutants	Live Mutants	Mutation Score
Largest Number	Mutation of all codes	242	196	46	81.19%
	Mutation of bug-prone codes	161	127	34	79.30%
Quadratic Equation	Mutation of all codes	114	100	14	88.00%
	Mutation of bug-prone codes	82	70	12	86.00%
Number of Digits	Mutation of all codes	77	73	4	95.10%
	Mutation of bug-prone codes	66	60	60	92.00%
Binary search	Mutation of all codes	155	136	19	88.00%
	Mutation of bug-prone codes	112	88	24	78.60%
Triangle	Mutation of all codes	445	304	141	68.31%
	Mutation of bug-prone codes	291	206	85	71.58%

Table 10 The effect of proposed method on the mutant reduction

Program name	Mutant Reduction Rate
Largest Number	33.47%
Quadratic Equation	28.07%
Number of digits	14.28%
Binary search	27.74%
Triangle	34.60%
AVG	27.63%

Similar results are produced for the *binary search* program. Figure 21 shows the average number of the generated mutants by different tools. The number of generated mutants for this benchmark are respectively 56 and 42. The sort programs are one of the most used program units in real-world programs. The proposed method makes a considerable reduction in the number of mutants. Indeed, the proposed method is independent of the test platform and tools. Figure 22 shows the generated mutants by different methods for the largest number program. The average number of mutants generated by different tools is about 70, while the number

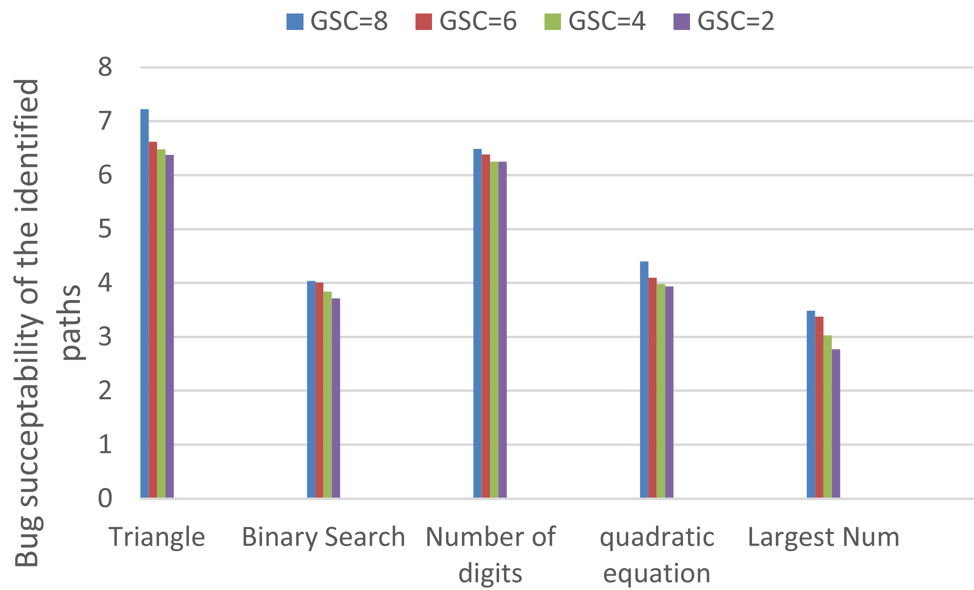


Fig. 18 The effects of GSC on the fitness of the obtained results by the FOA in different benchmarks

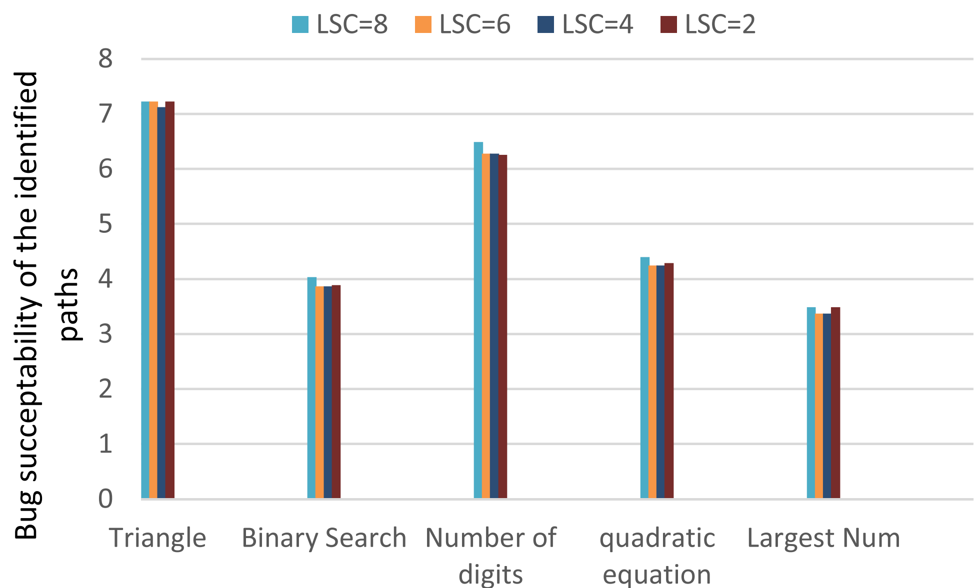


Fig. 19 The effects of LSC on the fitness of the obtained results by the FOA in different benchmarks

Table 11 The most bug-prone path of each benchmark is identified by the FOA

Bench. App	Most bug-prone path
Triangle	Best solution: [1 2 3 5 6 7 8 9 10 11 12 14 15 16 18 20 22 23 24]
Binary Search	Best solution: [1 2 4 5 7 8 10]
Number of digits	Best solution: [1 2 3 4 5 6 7 8 9 10 11 12]
Quadratic Eq.	Best solution: [1 2 3 5 6 8 10 11]
Largest Number	Best solution: [1 2 3 5 9 10 11]

of mutants generated by the same tools using the proposed method is about 52. All in all, the proposed method is platform independent method that can be used along with the different mutation tools.

The suggested FOA takes a subset of CFG`s paths as input (initial population). The initial population (a subset of testing paths) is selected randomly from the created control flow graph (CFG). The CFG of the input source code is created automatically by different tools in polynomial time complexity. In this study, the CFG was generated by Visustin tool. This tool takes the source code of the program under test and automatically generates the CFG. Each test path is implemented by an array (shown in Fig. 3). The suggested FOA is used to find out the most bug-prone paths of the program under test. The bug-proneness of each selected path is calculated by Eq. 7. The final population, as the most bug-prone paths of the input program, was considered for performing the mutation operators instead of all paths of the program. The proposed method reduces about

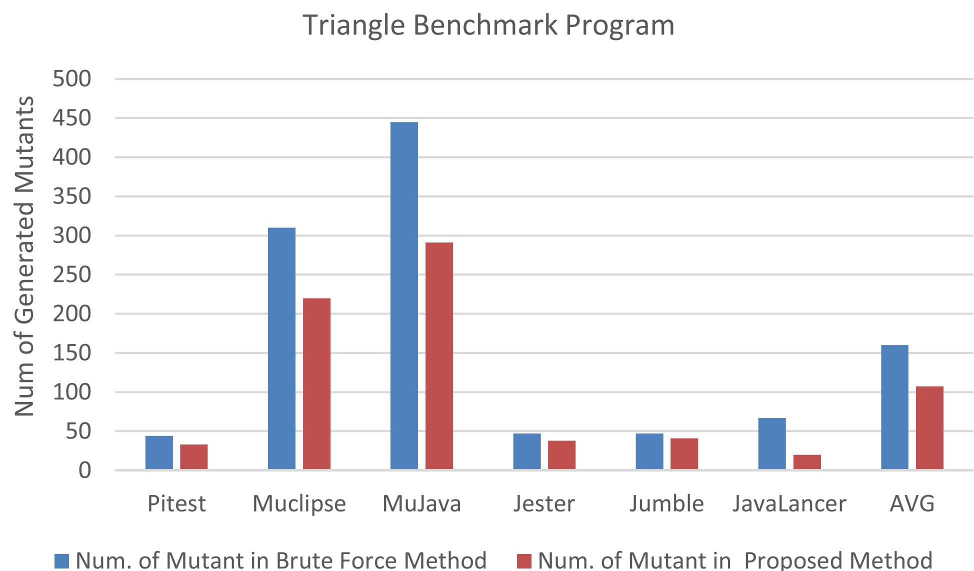


Fig. 20 Number of generated mutants in *triangle* benchmark by different mutation test tools with and without proposed method

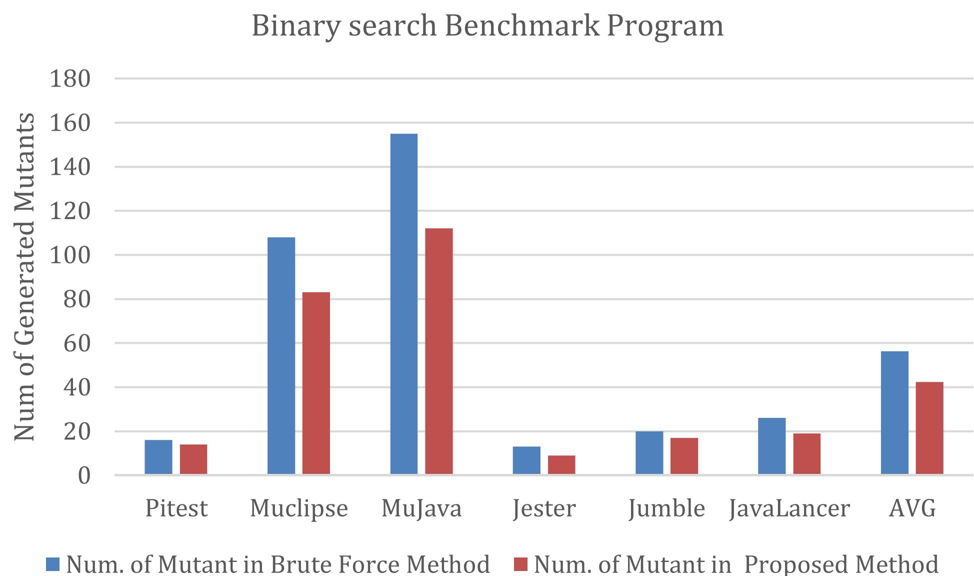


Fig. 21 Number of generated mutants in *binary search* benchmark by different mutation test tools with and without proposed method

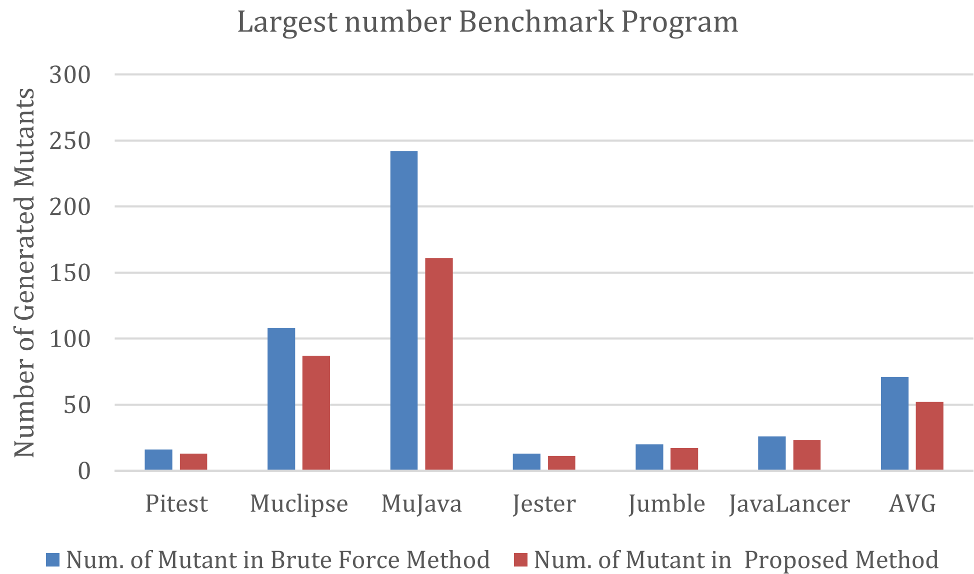


Fig. 22 Number of generated mutants in *largest number* benchmark by different mutation test tools with and without proposed method

27.63% of the created mutants when compared to existing techniques. The proposed method can be used to evaluate the effectiveness of the test data generating methods and tools with a limited time and cost consumption. The proposed method can be used in each testing tool such as Mujava, Muclipse, Jester, and Jumble.

ANOVA, as a statistical test, has been performed on the results obtained by the GA and proposed FOA. ANOVA is used to prove the significant effects of the FOA on the results of the experiments. The success rate of the two methods on the benchmarks has been used in the ANOVA analysis. Table 12 shows the statistical analysis of the results obtained by the GA and FOA. The *mean*, *variance,* and *standard deviation* of the obtained results during the 10 executions of each method are shown in Table 12. Table 13 shows the results of the ANOVA test. The values of p and f indicate the significance of the results. The f-ratio value is 11.52 and the p-value is 0.009442.

Table 12 The summary data in the ANOVA test on the success rate of the GA and proposed FOA

	FOA	GA	Total
$\sum X$	3.9	2.7	6.6
Mean	0.78	0.54	0.66
$\sum X^2$	3.09	1.51	4.6

Table 13 The results of the ANOVA test on the success rate of the GA and proposed FOA

Source	*SS*	*df*	*MS*	
Between-treatments	0.144	1	0.144	$F = 11.28$
Within-treatments	0.1	8	0.0125	
Total	0.244	9		

Hence, the result is significant at $p < .01$ and the proposed FOA has significant effects on the number of mutants and hence on the time and cost of software mutation testing.

5 Conclusion and Future Studies

Reducing the number of generated mutants is the main goal of this research. The proposed method identifies the most bug-prone paths at the first stage; then, the identified bug-prone paths are considered in mutation tasting at the second stage of the method. The modified version of the FOA, as a heuristic algorithm, was used at the first stage of the proposed method. MuJava is used as the code mutation tool. The original program and also the sliced program by the proposed method were mutated by MuJava. The mutants are executed with the test data generated by the Traxtor tool. This tool automatically generates the coverage-based test data at the unit level (function level). Indeed, the generated mutants have been executed by the generated test data via Traxtor. The results of conducted experiments confirm that the proposed method avoids mutating the non-bug-prone codes of the program. All in all, the method performs the mutation test with a lower number of mutants and lower cost. The method has higher performance and stability than the other methods. Bug-susceptibility of a line of source code is a function of different parameters, such as complexity parameters. Some of these effective parameters are now unknown. These parameters depend on the programming language features and programming styles. Also, the people (programmers) features may affect the probability distribution of bug occurrences. Analyzing and identifying the parameters effective on the bug-susceptibility are suggested as one of future study. Other evolutionary algorithms can be employed to get optimal results.

Author Contribution All authors contributed to the study's conception and design. The data collection, and analysis were performed by Bahman Arasteh. Experiments have been performed by Bahman Arasteh. All authors read and approved the final manuscript.

Data Availability The datasets generated during and the implemented code during the current study are available in the google.drive and can be freely accessed by the following link: https://drive.google.com/drive/folders/1eHkLdF2b-of6LqgJQaEpcyLjcfejz0Az?usp=sharing.

Declarations

Conflict of Interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Acree AT, Budd TA, DeMillo RA, Lipton RJ, Sayward FG (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 Software Eng Knowl Eng 27(6):2017
3. 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
4. Arasteh B, Imanzadeh P, Arasteh K et al (2022) A Source-code Aware Method for Software Mutation Testing Using Artificial Bee Colony Algorithm. J Electron Test 38:289–302. https://doi.org/10.1007/s10836-022-06008-9
5. Arasteh B (2019) ReDup: A software-based method for detecting soft-error using data analysis. Comp Electrical Eng 78(September 2019):89–107
6. Barbosa EF, Maldonado JC, Vincenzi AMR (2001) Toward the determination of sufficient mutant operators for C. Software Testing, Verification and Reliability 11(2):113–136
7. Bouyer A, Arasteh B, Movaghar A (2007) A New Hybrid Model Using Case-Based Reasoning and Decision Tree Methods for Improving Speedup and Accuracy. IADIS International Conference of Applied Computing
8. Budd TA (1980) Mutation Analysis of Program Test Data. Yale University
9. Chandra SSV, Sankar SS, Anand HS (2022) Smell Detection Agent Optimization Approach to Path Generation in Automated Software Testing. J Electron Test. https://doi.org/10.1007/s10836-022-06033-8
10. 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
11. Deng L, Offutt J, Ammann P, Mirzaei N (2017) Mutation operators for testing Android apps. Inf Softw Technol 81(2017):154–168
12. Dominguez-Jimenez JJ, Estero-Botaro A, Garcia-Dominguez A, Medina-Bulo I (2011) Evolutionary mutation testing. Inf Softw Technol 53(10):1108–1123
13. Fenton NE, Ohlsson N (2000) Quantitative analysis of faults and failures in a complex software system. IEEE Transact Software Eng 26(8):797–814
14. 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
15. Gheyi R, Ribeiro M, Souza B, Guimarães M, Fernandes L, d'Amorim M, Alves V, Teixeira L, Fonseca B (2021) Identifying method-level mutation subsumption relations using Z3. Inf Softw Technol 132:106496
16. 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 Technologies and Applications 55(1):118–148. https://doi.org/10.1108/DTA-03-2020-0073
17. Howden WE (1982) Weak mutation testing and completeness of test sets. IEEE Trans Software Eng 8(4):371–379
18. Jackson D, Woodward MR (2000) Parallel firm mutation of Java programs. Proc. First Workshop on Mutation Analysis, pp 55–61
19. Jia Y, Harman M (2011) An Analysis and Survey of the Development of Mutation Testing. IEEE Trans Software Eng 37(5):649–678. https://doi.org/10.1109/tse.2010.62
20. Keshtgar A, Arasteh B (2017) Enhancing Software Reliability against Soft-Error using Minimum Redundancy on Critical Data. https://doi.org/10.5815/ijcnis.2017.05.03
21. King KN, Offutt AJ (1991) A Fortran language system for mutation-based software testing. Software: Practice and Experience 21(7): 685–718
22. Kintis M, Papadakis M, Malevris N (2010) Evaluating mutation testing alternatives: a collateral experiment. Proceedings of the 17th Asia-Pacific Software Engineering Conference (APSEC)
23. Kurtz B, Ammann P, Delamaro ME, Offutt J, Deng L (2014) Mutant subsumption graphs. 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
24. Kurtz B, Ammann P, Offutt J (2015) Static analysis of mutant subsumption. IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
25. Ma YS, Offutt J, Kwon YR (2006) MuJava: A Mutation System for Java. In 28th International Conference on Software Engineering (ICSE '06)
26. Malevris N, Yates D (2006) The collateral coverage of data flow criteria when branch testing. Inf Softw Technol 48(8):676–686
27. Manizheh G, Mohammad-Reza F (2014) Forest Optimization Algorithm. Expert Syst Appl 41(15):6676–6687
28. 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
29. Offutt AJ, Rothermel G, Zapf C (1993) An experimental evaluation of selective mutation. Proceedings of the 15th International Conference on Software Engineering, ICSE '93. IEEE Computer Society Press, Los Alamitos, CA
30. Papadakis M, Malevris N (2010) An empirical evaluation of the first and second order mutation testing strategies. 2010 Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)
31. Sankar SS, Chandra SS (2020a) A Structural Testing Model Using SDA Algorithm. Lect Notes Comput Sci 405–412. https://doi.org/10.1007/978-3-030-53956-6_36
32. Sankar SS, Chandra SS (2020b) An Ant Colony Optimization Algorithm Based Automated Generation of Software Test Cases. Lect Notes Comput Sci 231–239. https://doi.org/10.1007/978-3-030-53956-6_21
33. Wei C, Yao X, Gong D, Liu H (2021) Spectral clustering based mutant reduction for mutation testing. Inf Softw Technol 132(2021):106502
34. Wong WE (1993) On mutation and data flow. Ph.D. dissertation, Purdue University
35. Yao X, Zhang G, Pan F, Gong D, Wei C (2020) Orderly Generation of Test Data via Sorting Mutant Branches Based on Their Dominance Degrees for Weak Mutation Testing. IEEE Trans Software Eng 48(4):1169–1184. https://doi.org/10.1109/tse.2020.3014960
36. Zhang L, Hou SS, Hu JJ, Xie T, Mei H (2010) Is operator-based mutant selection superior to random mutant selection?

Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering

37. Zhang L, Gligoric M, Marinov D, Khurshid S (2013) Operator-based and random mutant selection: better together. Automated Software Engineering (ASE). IEEE/ACM 28th International Conference

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Bahman Arasteh was born in Tabriz. He received the master's degree in software engineering from Azad University of Arak, and the Ph.D degree in software engineering from Islamic Azad University, Tehran Science and Research Branch, respectively. Currently, he is an associate professor at Istinye University, Istanbul, Turkiye. He has published more than 50 papers in refereed international journals and conferences. He is the coordinating editor in the springer journal of electronic test, and he is the reviewer of different international journals in Elsevier, Springer, Wiley and Hindawi. His research interests include search-based software engineering, Software testing, optimization algorithms, software fault tolerance, and software security.

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

Peri Gunes is an assistant professor in Dogus university in Turkiye. Her research interest includes computer networks, evolutionary algorithms and their function in dependability engineering.

Farzad Kiani is associate professor at Fatih Sultan Mehmet Vakif University in Turkiye. His research interest includes software testing, evolutionary algorithms and network security.

Mahsa Torkamanian-Afshar is an assistant professor in Nisantasi University in Turkiye. Her research interest includes the dependability of computer networks, and evolutionary and optimization algorithms.