Abstract—Online judgment (OJ) systems can automatically evaluate program results by executing corresponding test cases. Among them, automatic fault localization techniques can assist students in localizing the root cause of program faults. However, artificially constructing many valid test cases is a very time-consuming and challenging task. Therefore, we propose an approach for automatically generating test data to solve this problem. Specifically, we utilize the real faulty novice programs from the OJ system as training data, and then we leverage the search algorithm to evolve the test data according to the fitness function. To evaluate the effectiveness of our approach, we conduct experiments on 1,136 faulty novice programs collected from the OJ system. The experimental results show that the test data generated by our approach can achieve the same accuracy of fault localization as the artificially designed test data while has better fault localization accuracy for most fault localization techniques.

Index Terms—Programming Education, Fault Localization, Test Cases Generation, Genetic Algorithm

I. INTRODUCTION

In recent years, Massive Open Online Course (MOOC) has received widespread attention [1]. However, it is difficult for teachers to provide targeted guidance for each student. For program design courses, teachers manually debug and give corrective plans, which results in a considerable workload and low efficiency [2]. After receiving the guidance, students still need to review the idea of the question, analyze the root cause of the fault, and constantly try to fix it. Repeating this process often dissipates their enthusiasm for further analysis of the program. Therefore, the automated debugging and evaluation feedback of student programs has become an active research topic.

Online Judge (OJ) system is a system designed for reliability evaluation of the code submitted by users [3]. The OJ system tests the programs submitted by users under strict execution restrictions and determines whether the logic constraints are met. Reasonable use of the OJ system in teaching can help teachers save a lot of labor costs and more accurately verify the accuracy of student codes [4]–[6]. However, the traditional OJ platform can only inform students that the program is correct or incorrect and does not inform which statements in the program may contain faults. It is difficult for novice programmers to associate such simple platform interaction information with the root cause of the fault in the program. Therefore, accurately and effectively providing personalized guidance information to assist students with localizing program faults is still a challenging task [7], [8].

Personalized feedback aims to assist students in localizing their faults quickly. The automatic fault localization technique is an effective way to guide programmers to focus on specific parts of the program. Through the fault localization technique, the user can obtain the localization of faulty statements in the programs. By focusing more effort on the most suspicious statements, programmers can save a lot of time during debugging [9]. At present, researchers have conducted much research in the field of software fault localization, and the existing techniques can achieve promising results in the program with artificial faults, but the performance is not satisfactory in the case of real faults [10]. Araujo et al. [11] utilized the Spectrum-Based Fault Localization (SBFL) technique to localize faults in novice programs and found that nearly 40% of programs could not even pass a test case, which fails to meet the traditional SBFL conditions.

As an essential part of software testing, test data quality directly affects the effectiveness of fault localization [12]. However, we found that the artificially constructed test data in the traditional OJ system is not suitable for fault localization because the test data in the platform is maintained manually or semi-manually, which original aims to detect the faults contained in the user code, but not to help users localize code faults. Moreover, for the platform administrator, managing and maintaining the platform question bank itself is a time-consuming and laborious task. With the continuous increase of platform questions, it is not practical to rely on the artificial construction of test data. Therefore, automatically obtaining test data suitable for fault localization has become an urgent problem to be solved.

To improve the fault localization performance of existing techniques in novice programs, we propose a fault localization-guided test data generation approach based on the search algorithm method. In particular, our approach firstly uses the real faulty programs existing in the OJ system as training data, and then we design a fault localization-guided fitness function to evaluate the test data quality. Finally, this approach can automatically evolve the high-quality test data to improve the accuracy of fault localization.

In order to evaluate the effectiveness of the test data generated by our approach on fault localization, we conduct experiments on 1,136 faulty novice programs. The result
shows that, in all cases, the test data generated by our approach can identify 100% of the faulty programs as well as the original artificially constructed test data. Taking Dstar [13] on SBFL as an example, compared with the fault localization base on artificially designed test data, our approach can improve 11.6% in terms of the EXAM metric and almost 30 times improvement in terms of the TOP-1 metric. In other words, using the test data generated by our approach to conduct fault localization techniques in novice programs, students can use less effort to localize more faults.

To our best knowledge, the main contributions of our study can be summarized as follows.

- We propose a novel fault localization-guided test data generation approach, which leverages the search-based algorithm to generate test data to achieve better fault localization accuracy in novice programs.
- We design an effective fitness function within our approach, which can qualify the fault localization ability and fault detection ability of test data.
- We apply our approach to real novice programs from seven OJ questions. The experimental results show that our approach can significantly improve the accuracy of fault localization compared with the original artificially constructed test data without losing the fault detection ability.

The rest of this paper is structured as follows. Section II introduces the background of the OJ system and related techniques. Section III explains how our approach works in detail. Section IV discusses the experimental setup and the corresponding results. Section V presents the threats to validity. Finally Section VI concludes this study.

II. BACKGROUND

A. Online Judge System

Online Judge (OJ) system was originally used to judge programs submitted to solve questions and rank contestants automatically in programming competitions like ACM-ICPC (International Collegiate Programming Competition) and IOI (Olympiad in Informatics) [14]. Nowadays, it has been widely used in practicing programming skills for students, the training and selection of contestants, and the automatic submission and judgment for programming courses. For instance, the National University of Singapore has already successfully applied the OJ system in the teaching of Algorithm and Data Structure courses [3].

Typically, OJ system managers should prepare multiple test cases for each question, and each test case contains a pair of standard inputs and outputs. After receiving a program submitted by users, OJ system will execute the source code and compare its every output with the expected output. Finally, OJ system will return the judgment result of each program. Fig. 1 gives an overview of the framework of OJ system.

Normally, a student submits a program that already conforms to the sample input and expected outputs, most unaccepted programs fail in complex test cases that are not shown to users, and the only feedback information for students is their programs are accepted or not. Thus, although the OJ system is wildly used in education, it lacks other useful feedback to help students localize and fix their faults quickly.

In recent years, some researchers paid attention to OJ program related topics. Ahmed et al. [15] attempted to fix compile errors in student programs by deep learning techniques. Drummond et al. [6] use KNN regression model to grade student programs automatically. And there were other studies that tried to design a new OJ system that can provide more various and specific feedback information rather than just right or wrong [2], [4], [5], [16].

In this paper, we aim to enhance the function of the OJ system by providing fault-localization-related feedback, and we focus on the utilization of fault localization techniques on OJ programs. The background and related work about fault localization are introduced in the next section.

B. Fault Localization

Finding the exact localization of faults in programs is very time-consuming and costs a lot of human effort. To alleviate the difficulty of debugging, a lot of fault localization techniques were proposed, such as Spectrum-Based Fault Localization (SBFL) [17], [18], Mutation-Based Fault Localization (MBFL) [19]–[21], information retrieval-based fault localization [22], predicate switching [23], dynamic program slicing [24], stack trace analysis [25], history-based fault localization [26] and so on. Such fault localization techniques have been widely studied in commercial or open-source programs.

Since the novice programs investigated in this paper are quite different from other subject programs used in previous fault localization studies, not all kinds of fault localization techniques are suitable for novice programs. In this paper, we consider three widely studied fault localization techniques, SBFL, MBFL and Slicing-Based Fault Localization, which are introduced as follows.

1) Spectrum-Based Fault Localization: Spectrum-Based Fault Localization (SBFL) is a widely studied technique that tries to find the localization of faults in programs by employing the spectrum information and execution results of test cases [27].

Fig. 2 shows the framework of SBFL. It first collects coverage information and results (failed/passed) of test cases...
Program Code
print("error")
if a+b
print("regular triangle")
elif a==b and b==c:
else:
print("right triangle")
else:
print("equilateral triangle")
elif a*a+b*b==c*c or a*a+c*c==b*b or b*b+c*c==a*a:
while True:
print("isosceles triangle")
a, b, c = map(int, input().split())

T = [t
parameter ‘a state-of-the-art technique proposed by Wong et al., and the
Table II. Among the five formulas shown in Table I, DStar is
formation and execution results of the test cases, as shown in
are shown in Table I. These formulas calculate suspiciousness
formulas were proposed, and five popularly studied formulas
fault localization accuracy of SBFL, a lot of suspiciousness
accuracy of SBFL techniques is often low to localize faults
in real-world programs [9].
Table IV illustrates how SBFL works in the novice program
shown in Table III. There are two test cases in the OJ system
that validate the novice program shown in Table III. As can be
seen in the middle part of Table IV, the rank of the exact faulty
statement (S4) is only slightly higher than the rank of three
other statements, indicating that the fault localization accuracy
is insufficient to assist students in fixing their faults.
2) Mutation-Based Fault Localization: Mutation-Based
Fault Localization (MBFL) [19]–[21], [32] employs mutation
analysis to find the localization of faults in software. Fig. 3
shows the framework of MBFL. Different from SBFL, after
collecting coverage information and execution results of test
cases, MBFL will use a lot of mutation operators to inject
artificial faults into the program under test. The program
with artificially injected faults is called a mutant. During the
MBFL process, a large number of mutants will be generated
and executed on all test cases. Finally, the suspiciousness of

during the execution process. Next, SBFL uses suspiciousness
formulas to calculate the probability that each statement
tains faults. Finally, a rank list will be generated by descen
dering ordering all statements according to their suspiciousness
values. The rank list is used to assist developers in finding
faults quickly by checking the rank list from top to bottom.
The higher rank of the exact faulty statement is, the better the
corresponding fault localization technique is. To improve the
fault localization accuracy of SBFL, a lot of suspiciousness
formulas were proposed, and five popularly studied formulas
are shown in Table I. These formulas calculate suspiciousness
based on four parameters collected from the coverage informa
tion and execution results of the test cases, as shown in
Table II. Among the five formulas shown in Table I, DStar
is a state-of-the-art technique proposed by Wong et al., and the
parameter ‘a’ of the DStar is set to 3 in this paper according to
the recommendation of Wong et al. [13].

TABLE I
SUSPICIOUSNESS FORMULAS

<table>
<thead>
<tr>
<th>Name</th>
<th>SBFL formula</th>
<th>MBFL formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jaccard [28]</td>
<td>( S_{us}(s) = \frac{\text{fail}(s)}{\text{totalpass}+\text{pass}(s)} )</td>
<td>( S_{us}(s) = \frac{a_{kf}}{a_{kf}+a_{nf}+a_{kp}} )</td>
</tr>
<tr>
<td>Tarantula [29]</td>
<td>( S_{us}(s) = \frac{\text{fail}(s)}{\text{totalpass}+\text{pass}(s)} )</td>
<td>( S_{us}(s) = \frac{a_{kf}}{a_{kf}+a_{nf}+a_{kp}} )</td>
</tr>
<tr>
<td>Ochiai [30]</td>
<td>( S_{us}(s) = \frac{\text{fail}(s)}{\sqrt{\text{totalfail}(\text{fail}(s)+\text{pass}(s))}} )</td>
<td>( S_{us}(s) = \frac{a_{kf}}{\sqrt{(a_{kf}+a_{nf})(a_{kf}+a_{kp})}} )</td>
</tr>
<tr>
<td>OP2 [31]</td>
<td>( S_{us}(s) = \text{fail}(s) - \frac{\text{pass}(s)}{\text{totalpass}+1} )</td>
<td>( S_{us}(s) = a_{kf} - \frac{a_{kp}}{a_{kp}+a_{nf}+1} )</td>
</tr>
<tr>
<td>Dstar* [13]</td>
<td>( S_{us}(s) = \frac{\text{fail}(s)^2}{\text{pass}(s)+(\text{totalfail}-\text{fail}(s))} )</td>
<td>( S_{us}(s) = \frac{a_{kf}^2}{a_{kp}+a_{nf}} )</td>
</tr>
</tbody>
</table>

TABLE II
PARAMETERS FOR SBFL

<table>
<thead>
<tr>
<th>Term</th>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{fail}(s)</td>
<td>Number of failed tests that execute statement s</td>
</tr>
<tr>
<td>\text{pass}(s)</td>
<td>Number of passed tests that execute statement s</td>
</tr>
<tr>
<td>\text{total fail}</td>
<td>Total number of failed test cases</td>
</tr>
<tr>
<td>\text{total pass}</td>
<td>Total number of passed test cases</td>
</tr>
</tbody>
</table>

SBFL is a light-weight technique because it calculates
the suspiciousness of each statement by using test cases’
exection results and coverage information. Such a process
avoids complex semantic and code analysis. However, the
accuracy of SBFL techniques is often low to localize faults
in real-world programs [9].

Fig. 2. Framework of SBFL
The suspiciousness value of each statement is defined as:

\[ Sus(s) = \max_{m \in \text{mut}(s)} \{M(m)\} \]  

The suspiciousness value of a statement \( s \) is defined as:

\[ Sus(s) = \frac{1}{\text{mut}(s)} \sum_{m \in \text{mut}(s)} Score(s) \]  

\[ Score(s) = \frac{|f_p(s) \cap p_m|}{|f_p|} - \alpha \frac{|p_p(s) \cap f_m|}{|p_p|} \]

where \( |\text{mut}(s)| \) is the number of mutants generated from mutating on statement \( s \), \( f_p(s) \) represents the failed test cases which cover \( s \). Similarly, \( p_p(s) \) represents the passed test cases which cover \( s \), and \( f_m \) and \( p_m \) indicate the number of failed and passed test cases respectively when executing all test cases on the corresponding mutant \( m \). The first term, \( \frac{|f_p(s) \cap p_m|}{|f_p|} \), reflects the first kind of test cases which failed on \( P \) but now passed on mutant \( m \). The second term, \( \frac{|p_p(s) \cap f_m|}{|p_p|} \), reflects the second kind of test cases which passed on \( P \) but now failed on mutant \( m \). Besides, the weight \( \alpha \) are used for adjusting the average values of the two terms to be the same [32].

Recent researches demonstrated that the two kinds of MBFL techniques, Metallaxis and MUSE, showed a significant advantage over state-of-the-art SBFL techniques [20], [32]. Moreover, Metallaxis are twice as much accurate as MUSE with only half of the cost [9]. In the rest of this paper, we chose Metallaxis as MBFL technique to conduct our experiments.

Table IV also lists the illustration example of MBFL when localizing faults in the same novice program mentioned above. It can be seen from Table IV that the suspiciousness value of the exact faulty statement (S4) is the most with other three correct statements.

**3) Slicing-Based Fault Localization: **Slicing-Based Fault Localization (abbreviated as Slice hereafter) is base on program slicing technique. Program slicing is a program decomposition method first proposed by Weiser in 1979 [35].

Based on the original definition of Weiser, A slicing criterion contains several variables at some point of interest. Program slicing refers to a set of program statements that may affect slicing criterion [36].

Program slicing was introduced as a debugging tool to reduce a program to a minimum while still maintaining a given behavior [35]. Static slicing just searches for all possible statements that affect any variables in slicing criterion. Makes use of information about a particular execution of a program.
A dynamic slice contains all statements that actually affect the value of a variable at a program point for a particular execution of the program rather than all statements that may have affected the value of a variable at a program point for any arbitrary execution of the program.

During the experiment, if there is only a single failed test, we use the statements covered by the test as the program slice. When multiple test cases fail, we can take the following three strategies from a previous study [37] to utilize multiple slices: union, intersection, and frequency. The first two strategies will take the union and intersection of the failing test cases coverage statements as a result. The frequency strategy will count the number of times these statements are covered, and sort these statements according to the number of covered times and return a ranked list.

III. Approach

Fig. 4 shows the fault localization-guided test data generation approach framework based on the search algorithm, which contains three modules (i.e., Data Initialization, Dynamic Evolution, and Fault Localization). First of all, we should determine the data coding strategy for a specific OJ question to initialize the input data of the search algorithm, and the selected coding strategy will be used for the subsequent data generation process.

Next, we choose the appropriate search algorithm and fitness function according to the coding strategy in the given generation scenario. In the dynamic evolution module, the test data generated by the search algorithm continuously drives the novice program under test to generate fault localization feedback information through the fault localization module. After that, the fitness function will be used to evaluate test case quality for further evolution to generate better test cases.

With the evolution of the population, the search process will be finally terminated when the fault localization accuracy reaches the except or meet the maximum iteration number. After that, we use the best individual of the population as the final test data, and then we utilize the existing fault localization techniques to complete the fault localization process.

Specifically, In this paper, we apply the Genetic Algorithm (GA) as the search algorithm to generate fault localization-guided test data, because the GA is a widely used algorithm in other traditional test data generation research [38], [39]. Note that traditional test data generation research mainly considers the adequacy and completeness of software testing, aiming to detect program faults as much as possible rather than fault localization.

Besides, combined with the characteristics of the subject program, the integer coding method is adopted. In the evolution process, we tack a small number of subject programs as evaluation sets and use average EXAM (introduced in Section IV-D) as the fitness function. Besides, we take measures for guidance control to ensure test data quality and optimize generation efficiency.

A. Test Data Coding and Initialization

Gene coding is the first step of the whole framework. Due to all the OJ questions used in this paper taking integer numbers as inputs, we use the integer coding method, and we initialize the population by filling in random legal values.

Suppose that the test data population size is \( M \), each individual is a test data set, and the individual contains \( N \) groups of integer test data. Fig. 5 is an example of a certain generation in the triangle judgment question.

\[
\begin{bmatrix}
3 & 4 & 5 \\
235 & 299 & 32 \\
... & ... & ... \\
1 & 2 & 9
\end{bmatrix}
\begin{bmatrix}
2 & 10 & 5 \\
8 & 27 & 8 \\
... & ... & ... \\
9 & 87 & 286
\end{bmatrix}
\begin{bmatrix}
9 & 33 & 7 \\
177 & 139 & 48 \\
... & ... & ... \\
31 & 60 & 279
\end{bmatrix}
\]

Fig. 5. Example of Population

As shown in Fig. 5, there are \( M \) matrices in this population. Each matrix composed of several rows of test data represents an individual. Each line contains three integers as the sides of the triangle, and each integer was generated randomly within the range of inputs defined by this question.

B. Fitness Function

Fitness function is the key interface of the genetic algorithm to solve practical problems. A genetic algorithm uses the fitness function to evaluate the quality of individuals. As for fault localization, \( \text{EXAM Score} \) is commonly used to evaluate the accuracy of fault localization techniques [40], [41], and a lower \( \text{EXAM} \) means it needs to check fewer program statements to find the exact faulty statement, then the corresponding technique has a better fault localization accuracy. In this paper, we take the reciprocal of the average \( \text{EXAM} \) as the fitness function, which can be calculated as follow:

\[
\text{Fitness}(T) = \frac{1}{\text{EXAM}(\text{trainingData}, T)} \quad (5)
\]

where \( T \) is the target individual under evaluation, and \( \text{trainingData} \) is a randomly selected training set of origin dataset of our experiments, in which 30 programs are selected for each question to be used as the training set. \( \text{EXAM}(\text{trainingData}, T) \) indicates the average of the \( \text{EXAM} \) value of all programs within \( \text{trainingData} \) under the test cases from \( T \). Finally, a higher \( \text{Fitness}(T) \) value indicates that individual \( T \) can achieve a better fault localization accuracy, and this individual should be remained.

C. Dynamic Evolution

Utilizing adaptive genetic operators to prevent premature convergence phenomena and ensure population diversity is a common optimization method in genetic algorithms. The adaptive genetic operator can dynamically adjust the probability of crossover and mutation with the change of population fitness in the genetic process, ensure that the good individuals of the population are protected, and quickly eliminate or
mutate the low-quality individuals. The calculation formulas of crossover probability $P_C$ and mutation probability $P_M$ used in the genetic algorithm are as follows:

$$P_C = \begin{cases} k_1 \frac{F_{\text{max}} - F}{F_{\text{avg}} - F_{\text{avg}}} & F \geq F_{\text{avg}} \\ k_2 & F < F_{\text{avg}} \end{cases}$$

$$P_M = \begin{cases} k_3 \frac{F_{\text{max}} - F'}{F_{\text{avg}} - F_{\text{avg}}} & F' \geq F_{\text{avg}} \\ k_4 & F' < F_{\text{avg}} \end{cases}$$

where $F_{\text{max}}$ is the maximum fitness value of the population; $F_{\text{avg}}$ is the average fitness of the population; $F$ is the larger fitness value of the two individuals that need to cross; $F'$ is the individual fitness value that needs mutation. $k_1$, $k_2$, $k_3$ and $k_4$ are constants.

Moreover, to improve generation efficiency and ensure test data quality, certain genetic-oriented control needs to be carried out according to the actual generation problems.

As we all know, the purpose of the fault localization technique is to distinguish fault statements from correct statements as far as possible. Therefore, the more similar coverage information of different test data is, the more closely the performance in fault localization is, which leads to more statements have the same suspiciousness value.

In addition, as for the test case generation problem, the most basic goal is that the generated test data should cover as many statements as possible. Consequently, we use a greedy strategy to control the genetic process from the statement coverage and coverage diversity. Algorithm 1 shows the genetic guidance control, which describes the process of two individuals crossing to generate a new individual.

According to Algorithm 1, we first initialize the new individual, and two empty sets are prepared to record the coverage path and the coverage statement, respectively (Line 1). Then we gradually add the test data that contribute to the overall statement coverage and coverage path diversity to the new individuals through the greedy strategy, but the individuals that do not contribute are put into $Gene2$ for random selection (Lines 3-21). Among them, we determine whether the current test data contributes to the overall coverage path diversity (Lines 4-9) and consider the statement coverage diversity (Lines 11-21). Finally, we fill in the vacant part through random selection and finally return the new individual (Line 24-26).

### IV. EXPERIMENT

#### A. Research Questions

To evaluate the fault localization performance on the test data generated by our approach, we conduct several empirical studies, and the following research questions are defined and investigated:

- **RQ1**: Compared with baselines, whether using test data generated by our approach for fault localization can achieve better performance?

In this paper, to evaluate whether the test case generated by our approach can improve the fault localization accuracy, we compare test data generate by our approach with the original manually designed test data on three fault localization techniques: SBFL, MBFL, and Slice. Besides, we use $EXAM$ and $TOP$-$N$ as evaluation metrics to evaluate the experiments.

- **RQ2**: What is the fault detection ability of test data generated by our approach?

When designing test data for a question, the most challenging and essential requirement is to detect whether the program meets OJ questions’ requirements. If the test data cannot find all the faulty programs, the OJ system will lose its use value. Therefore, we need to use the test data generated by our method to check out all the faulty programs.

- **RQ3**: How does the number of iterations of genetic algorithm affect the effectiveness of fault localization?

The test cases may not cover all the statement coverage and execution path in the phase with fewer iterations because the constructed data is too simple. As a result, it makes it impossible for us to localize the faults in the programs. In contrast, as the number of iterations increases, it will take much time to generate test data. Therefore, we study the impact of the number of iterations on the test data.

#### B. Experimental Setup

In our experiments, we collected most of the common mutation operators [20], [32] and adjusted them for python
Algorithm 1 Genetically Direction Control

Require: Gene1, Gene2: Population individual;
TCSIZE: Number of test cases within each individual;

Ensure: NewGene: Cross-generating new individuals;
1: newGene ← [], statementDic ← ∅, covpathDic ← ∅;
2: for aCase in Gene1 do
3: \( \text{Determine whether the current Case contributes to the diversity of coverage paths} \)
4: if aCase.cov not in covpathDic then
5: newGene.append(aCase);
6: covpathDic.record(aCase.cov);
7: statementDic.record(each statement in aCase.cov);
8: continue;
9: end if
10: \( \text{Determine whether the current Case contributes to statement coverage} \)
11: Flag = True;
12: for aStatement in aCase.cov do
13: if aStatement not in statementDic then
14: statementDic.record(aStatement);
15: Flag = True;
16: end if
17: end for
18: if Flag == True then
19: newGene.append(aCase);
20: continue;
21: end if
22: \( \text{The current Case has no contribution to sentence coverage and path diversity, put it in Gene2 to be filled randomly} \)
23: Gene2.append(aCase)
24: while newGene.length<TCSIZE do
25: newGene.append(Gene2[random.randomIndex])
26: end while
27: end for
28: return newGene

programs. Program coverage information is collected by coverage 1 and gcov 2 tool.

Besides, in the genetic algorithm, the population is set to 100; The maximum iteration number is 1000. It is difficult to judge whether the current generation reaches the optimal solution of fault localization, so we do not take any interrupt measures so that the genetic algorithm iterates to the final round and selects the optimal individual. The parameters \( k_1, k_2, k_3 \) and \( k_4 \) of the adaptive genetic operator are 0.5, 0.7, 0.02 and 0.09 respectively, which are mainly determined through the optimization of the process of the experiments.

C. Subject Programs

In the experimental phase, we choose the code submitted in an examination of the introductory programming course as the experimental program. The selected examination contains seven questions, which are described as follows:

- **Input Format-1**: This is an introductory input/output exercise, each input contains an integer \( N \) in the first line, and then \( N \) lines follow. Each line consists of a pair of integers \( a \) and \( b \), separated by a space, one pair of integers per line. Users should output the sum of \( a \) and \( b \) in one line, and with one line of output for each line in input.
- **Input Format-2**: Compared with the previous question, this is an advanced input/output exercise, each input contains an integer \( N \) in the first line, and then \( N \) lines follow. The first of each line is a positive integer \( m \) followed by \( m \) integers. For each line, Users need to calculate the sum of \( m \) integers, output one result in one line.
- **Who is the turn**: This is a simple ping-pong competition simulation problem, users play a role in referee and receive two integers as the current score to judge whether the game is over. If not, users need to judge who is the turn to serve according to the rule.
- **Judge triangle**: It is an easy problem which takes three integers as input, and the students need to write programs to judge what kind of triangle can the three integers form. The corresponding example programs can be seen in Table III.
- **Division**: It is the easiest problem for this task. Students should output the division result of two integers, \( a \) and \( b \). The calculation result needs to be rounded, and if \( b \) is equal to 0, then the output should be “error” since it’s illegal to divide by 0.
- **Piecewise function** This problem defined a common piecewise function. According to the function scope given by the background, users need to output the corresponding \( y \) value according to the relationship between the input \( x \) value and the corresponding function.
- **Poker game** This problem requires the user to determine who is the winner of the poker game. In the poker game, each gambler holds two cards. According to the rule, paired cards are greater than singular. Besides, if neither side holds paired cards, then compares the last digits of the sum of two cards.

Students provided multiple answers to these seven questions. After the judging of the OJ system, we selected 1,136 faulty programs for empirical research and excluded some defective programs for the following reasons: (1) Some programs have syntax errors and cannot be executed, which are not suitable for fault localization; (2) some programs will crash while it is running, and they cannot execute a complete test case; (3) Some programs are too incorrect, and the entire code should be rewritten (current program under test contains up to three faults). Finally, we select 1,136 programs as the subject programs. The detailed information is listed in Table V, including the title of the question, the number of faulty programs, the number of statements, the total number

---

1https://pypi.org/project/coverage/
2https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
of faulty statements, and the number of original test cases.

D. Evaluation Metrics

1) EXAM Score: EXAM Score (EXAM) [33], [42] gives the percentage of statements that need to be examined until the faulty statement is localized. EXAM Score is commonly used to evaluate the accuracy of fault localization techniques [40], [41], and a lower EXAM means it needs to check fewer program statements to find the exact faulty statement, then the corresponding technique has a better fault localization accuracy. The EXAM formula is defined as:

\[
EXAM = \frac{\text{rank of the faulty statement}}{\text{number of the executable statements}}
\]

where the numerator is the rank of faulty statements in the suspiciousness ranking list, and the denominator is the total number of executable statements that need to be checked.

However, there may be some statements which share the same suspiciousness value and cause the tie issues. Therefore, we use the average rank to present their final ranks, and such a strategy to handle the tie issues is widely adopted by existing fault localization techniques [9], [10], [43]. Taking a certain faulty statement S as an example, the number of correct statements with a higher suspiciousness value than S is A, and the number of statements shares the same suspiciousness value as S is B, then the final rank of the exact faulty statement S is \((A+1) + \frac{B}{2}\).

Besides, this equation would be disturbed when multiple faulty statements exist. For example, multiple faulty statements will result in multiple EXAM Scores. To overcome this problem, we would only evaluate the cost of localizing the first faulty statement in the rank list when there are multiple faulty statements in a program. This strategy is widely used to evaluate the ability to localize the first faulty statement for a fault [44]–[46].

2) TOP-N: This metric reports the number of faults, whose faulty statements can be discovered by examining the top N (N=1,2,3,...) statements of the returned suspiciousness ranking list of code entities [47], [48]. The higher the value of the TOP-N metric, the fewer efforts required for developers to localize the faults, and thus the corresponding fault localization technique will have a better performance. Note that if statements in the same program block have the same suspicious score, we would assign the average value of the highest and lowest rankings to those statements. Previous research [49] suggested that programmers check only the first few statements in the ranking list, and TOP-N metric reflects this.

E. Wilcoxon Signed-Rank Test

Wilcoxon Signed-Rank Test is an alternative hypothesis test approach when the test data cannot be assumed to be normally distributed [50]. Therefore, it can provide a reliable statistical basis for comparing the effectiveness of different approaches.

Therefore, to compare the fault localization techniques in different situations from the statistical view, we employ the Wilcoxon Signed-Rank Test to evaluate the confidence level of fault localization technique with test cases generated by our approach outperforms that with the original test cases.

In this paper, all the experiments were performed on Linux system (version 3.10.0-957.el7.x86-64) with 18 cores CPU (Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz). We collected most of the common mutation operators [20], [32]
and adjusted them for python programs. Program coverage information is collected by coverage \(^3\) and gcov \(^4\) tool.

**F. Result Analysis**

1) RQ1: The fault localization effectiveness of the test data generated by our approach: To answer RQ1, we use three fault localization techniques (i.e., SBFL \([27]\), MBFL \([34]\), and Slice \([35]\)) to conduct experiments. We compare the test case generated by our approach with the original test case by EXAM and TOP-N metrics, where the original test cases refer to the artificially designed test data that are actually applied to our OJ system. Besides, to make the experimental results more convincing, we use Wilcoxon Signed-Rank Test to prove the credibility of the conclusion further.

Fig. 6 shows the intuitive experimental results in terms of the EXAM metrics. It can be seen that the test data generated by our approach has a lower EXAM Score on most fault localization techniques. Table VI presents the specific values of Fig. 6 in detail, and the highlighted cell indicates that our test case can achieve a better fault localization effect on the corresponding technique. In Table VI, the first two columns show the different fault localization techniques, and the following three columns show the experimental results of two kinds of test data and their differences in terms of the EXAM metric. We can find that among the six representative methods for MBFL and the five most common formulas in the SBFL, the fault localization accuracy of the test data generated by our approach achieved significant improvements compared to the original artificially constructed test data. In particular, when we use the Dstar formula on SBFL, the test data generated by our approach with an 11.6% improvement in fault localization accuracy, and SBFL can achieve its best performance with the Ochiai formula which EXAM value is 0.179. However, for Slice, using the test data generated by our approach has only a slight improvement in terms of the EXAM metric, and the difference between the two kinds of test data is less than 4%. In other words, the performance of these two kinds of test data is relatively close on Slice.

Furthermore, we conduct Wilcoxon Signed-Rank Test to verify the fault localization competitiveness of our test data generation approach. The used hypothesis in our study is set as follows, \(H_0\): The fault localization performance with test cases generated by our approach does not significantly outperform that with the original test cases in terms of the EXAM metric. The significance level of this test is set as 0.05, and the sixth column in Table VI shows the results of the above hypothesis testing. Obviously, the \(p\)-values are lower than 0.05 in most fault techniques, and then the statistical results led to the rejection of the null hypothesis. Therefore, these results imply that among the majority of fault localization techniques, the fault localization performance with test cases generated by our approach significantly outperforms that with the original test cases in terms of the EXAM metric.

In addition, Table VII represents the results of the comparison in terms of the TOP-N metric. The first two columns of Table VII display all fault localization techniques, and the following columns show the TOP-N \((N = 1, 2, 4, 5)\) results. Overall, the results are similar to those of the EXAM metric, with significant improvements in both MBFL and SBFL technologies and a similar performance in Slice too. Especially when we use Dast on SBFL to conduct experiments, the fault localization effect of test data generated by our approach is most obvious on TOP-1, TOP-2, with 30 times and 25 times improvement, respectively.

In summary, the experimental results show that the test data generated by our approach can effectively improve the fault localization accuracy of both SBFL and MBFL on novice programs compared with the original artificial test data. Therefore using our approach to generate test data can effectively improve the debugging effectiveness of students.

2) RQ2: The fault detection ability of test data generated by our approach: To answer RQ2, we compared the number of wrong answers detected in subject programs with two kinds of test data. A wrong answer refers to the submitted program whose execution result is not accepted by the corresponding OJ system. The fault detection ability of an OJ system relies on the test data designed for each question. Since the origin artificial test cases are carefully constructed by questioners, we believe that it can take into account all the special test cases and detect all the faulty novice programs.

Table VIII presents an overview of two kinds of test data’s performance on fault detection ability. It can be seen that both test cases can detect all 1,136 faults in this paper. This proves that our approach has the same fault detection ability as the artificial design.

3) RQ3: The influence of the different iterations: To answer RQ3, we use our approach to generate the test data for the seven questions mentioned in Section IV-C. We choose the Ochiai formula on SBFL which achieved the best performance of SBFL in RQ1 in terms of the EXAM metric. The specific
TABLE VII
PERFORMANCE COMPARISON BETWEEN THE ARTIFICIAL TEST DATA AND THE TEST DATA GENERATED BY OUR APPROACH IN TERMS OF TOP-N

<table>
<thead>
<tr>
<th>Approach</th>
<th>Formula</th>
<th>TOP-1</th>
<th>TOP-2</th>
<th>TOP-3</th>
<th>TOP-5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GA</td>
<td>Origin</td>
<td>GA</td>
<td>Origin</td>
</tr>
<tr>
<td>SBFL</td>
<td>Jaccard</td>
<td>377</td>
<td>52</td>
<td>660</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>Tarantula</td>
<td>310</td>
<td>61</td>
<td>585</td>
<td>129</td>
</tr>
<tr>
<td></td>
<td>Ochiai</td>
<td>386</td>
<td>52</td>
<td>663</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>OP2</td>
<td>359</td>
<td>52</td>
<td>609</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>Dstar</td>
<td>275</td>
<td>9</td>
<td>502</td>
<td>21</td>
</tr>
<tr>
<td>MBFL</td>
<td>Jaccard</td>
<td>414</td>
<td>214</td>
<td>744</td>
<td>488</td>
</tr>
<tr>
<td></td>
<td>Tarantula</td>
<td>342</td>
<td>204</td>
<td>533</td>
<td>369</td>
</tr>
<tr>
<td></td>
<td>Ochiai</td>
<td>425</td>
<td>214</td>
<td>746</td>
<td>488</td>
</tr>
<tr>
<td></td>
<td>OP2</td>
<td>424</td>
<td>214</td>
<td>725</td>
<td>488</td>
</tr>
<tr>
<td></td>
<td>Dstar</td>
<td>428</td>
<td>214</td>
<td>747</td>
<td>488</td>
</tr>
<tr>
<td></td>
<td>MUSE</td>
<td>5</td>
<td>5</td>
<td>173</td>
<td>173</td>
</tr>
<tr>
<td>Slice</td>
<td>Frequency</td>
<td>6</td>
<td>6</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Union</td>
<td>6</td>
<td>6</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>Intersection</td>
<td>6</td>
<td>6</td>
<td>21</td>
<td>21</td>
</tr>
</tbody>
</table>

TABLE VIII
OJ SYSTEM’S FAULTS DETECTION CAPABILITY WITH DIFFERENT TEST CASES

<table>
<thead>
<tr>
<th>Approach</th>
<th>#Detected Wrong Answers</th>
<th>Failed/Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Origin</td>
<td>1,136</td>
<td>100%</td>
</tr>
<tr>
<td>GA</td>
<td>1,136</td>
<td>100%</td>
</tr>
</tbody>
</table>

results for each question are shown in Fig. 7. In particular, in each line chart, the horizontal axis represents the number of iterations of the genetic algorithm, and the vertical axis represents the EXAM Score. Besides, to demonstrate the fault localization ability of test cases generated by our approach, we use the black line to represent our average EXAM value for each iteration and the green line to show the EXAM Score of our best individual for each iteration.

As shown in Fig. 7, for both questions A and B, the average EXAM and the best EXAM Score remain essentially unchanged throughout the iterative process, which reason is that the programs for both questions A and B contain only one statement block, and no matter how we modify the combined test data, SBFL cannot distinguish the statements in that statement block. For other questions, there is a very similar trend. The optimal individual’s EXAM Score suddenly decreases at some points, and we found that the new cases generated at these points contain special cases with increased fault localization capability. For example, in question D, its two main inflection points are the generation = 12 and the generation = 252. We compare generation = 1 with generation = 12 and find that the number of statements covered by the test data is the same as the number of path types covered, but the test data is more balanced among the types of test data, so the fault localization accuracy is optimized. In the second point, equilateral triangles are constructed here to enhance the diversity of the data. Therefore, when the number of iterations is at a smaller level, there may be only a local optimal solution and get a poor result, and when the number of iterations grows to a large stage, our approach could get a satisfactory result conveniently.

V. THREATS TO VALIDITY

Threats to Internal Validity. One threat to internal validity relates to experimental errors and biases. During the localizing phase, we have selected three fault localization techniques (i.e., SBFL, MBFL and Slice) and five suspiciousness calculation formulas, which strategies contain widely studied programs with real faults [24], [51]–[55], which makes these threats limited.

Besides, the second threat to internal validity is related to the parameters we selected in our genetic algorithm. The
Fig. 7. Evolution Trend of Fitness Value of Genetic Algorithm

parameter setting of our proposed method mainly comes from the optimization of the process of the experiments. We will explore automatically adapt approaches in our future work.

**Threats to External Validity.** The threat of external validity is related to the scalability of our proposed method. We use 1,136 real-world novice programs as the experimental dataset, but our conclusions may be limited by choice of language (C/C++, Python). To alleviate this threat, We have selected diverse faulty programs as our experimental programs. Specifically, both single-fault and multiple-fault programs are considered. Besides, our experimental programs include simple programs that contain only one block and complex programs that include judgment or loop statements.

**Threats to Construct Validity.** Threats to construct validity include how well the measurements we take are actually correlated to what they claim to measure. We use EXAM and TOP-N to evaluate our approach. They are widely used in evaluating fault localization [56] and ranking techniques [57].

**VI. CONCLUSION**

In this paper, we propose a fault localization-guided test data generation approach. We employ genetic algorithms to construct a search algorithm for OJ questions to generate test data. It takes the faulty programs existing in the OJ system as training data, designs the adaptation function for fault localization, and automatically evolves to generate test data to increase fault localization accuracy. To validate our approach, we generated corresponding data for seven OJ questions to test the fault detection and fault localization capabilities. The experimental results show that the test data generated by our approach can achieve the same fault detection ability as the artificially designed test data. In addition, test data generated by our approach with significantly better fault localization accuracy than the original test cases for both SBFL and MBFL.

In the future, we plan to employ more approaches to improve the fault localization accuracy of novice programs. (1) Optimize the generation of test data for multiple metrics. (2) Use learning algorithms to optimize fault localization accuracy. (3) Extending our approach to various industrial and open-source programs.

**ACKNOWLEDGMENT**

The work is supported by the National Natural Science Foundation of China (Grant nos. 61902015, 61872026 and 61672085).

**REFERENCES**


