Using Fine-grained Test Cases for Improving Novice Program Fault Localization

Abstract—**Online Judge (OJ) system, which can automatically evaluate the results (right or wrong) of programs by executing them on standard test cases, is widely used in programming education. The test cases in OJ system are designed for fault detection, which aims to check whether the user-submitted programs meet the problem requirements or not. While an OJ system with personalized feedback can not only give execution results, but also provide information to assist students in locating their problems quickly. Automatically fault localization techniques are designed to find the exact faults in programs automatically, experimental results showed their effect on locating artificial faults, but their effectiveness on novice programs needs to be investigated. In this paper, we first evaluate the effectiveness of several widely-studied fault localization techniques on novice programs, and then we use fine-grained test cases to improve the fault localization accuracy. Empirical studies are conducted on 77 real student programs and the results show that, compared with original test cases in OJ system, the fault localization accuracy can be improved obviously when using fine-grained test cases. More specifically, in terms of TOP-1, TOP-3 and TOP-5 metrics, the maximum results can be improved from 5, 22, 37 to 9, 24, 48, respectively. The results indicate that more faults can be located when checking the top 1, 3 or 5 statements, so the fault localization accuracy is enhanced. Furthermore, a Test Case Granularity (TCG) concept is introduced to describe fine-grained test cases, and empirically studies demonstrate that there is a strong correlation between TCG and fault localization accuracy.**

Index Terms—**Programming Education, Fault Localization, Fine-grained Test Cases, Test Case Granularity**

I. Introduction

Online Judges(OJ) system is used for evaluating the programming code submitted by users, which is next compiled and tested in a homogeneous environment [1]. A user-submitted program will be subject to strict execution restrictions under OJ system, including run-time limits, memory usage limits, and security limits. The results of the execution of the user program will be captured and saved, and then transferred to a judge client. The judge client will compare the output of the user program with the standard output to judge whether the submission meets certain logic conditions.

Due to the popularity of Massive Open Online Courses (MOOC) [2], providing feedback on programming assignments is a tedious, error-prone, and time-consuming part of a class on introductory programming and requires a lot of effort form the teaching personnel [3], [4]. This problem becomes more and more impressing with the increasing demand for programming education. Therefore, more and more universities utilize OJ system to relieve the pressure of providing feedback on programming.

Utilization of OJ system allows teachers to assess students' assignments automatically [5]–[7]. Firstly, teachers can verify the correctness of solutions submitted by students with higher accuracy. When the teacher constructs the complete set of test cases covering all corner cases resulting from the problem definition, the possibility of acceptance of an incorrect solution is almost negligible. Second, the time cost for evaluation is much shorter, therefore, the teacher can free their time to assign to students more exercise. Finally, students receive an almost instant answer as to whether their solution is correct. However, it is difficult for novice programmers to map such feedback to the root failures' cause in their programs. Which test case failed or which statements contain faults are not provided in the feedback information of OJ system.

Personalized feedback aims to assist students in locating their problems quickly. But providing accurate and effective personalized feedback is quite a challenge [8]–[10]. Fault localization technique is an effective way to direct programmers to focus on specific parts of a program. fault localization techniques will generate a sorted list of suspicious program locations, such as functions, lines or statements [11]. A programmer can save time during debugging by focusing attention on the most suspicious locations. Nowadays Dozens of fault localization techniques have been proposed, and experimental studies proved that fault localization techniques are quite effective on artificial faults [12]. However, as for real-world faults, Qi et al [13] conduct an experiment on 15 popular fault localization techniques with 11 subjects real-world programs and a user study finding that all the fault localization techniques performing well in prior studies do not have better localization effectiveness according to user feedback. Kochhar et al [14] investigate practitioners' expectations on automated fault localization, finding that although practitioners are enthusiastic about research in fault localization, they have high thresholds for adoption. Araujo et al [15] Applying Spectrum-based Fault Localization (SBFL) on Novice's Programs and finding that nearly 40% student wrong programs even do not meet the fault localization method conditions which require at least one pass test case and one or more fail test cases.

In this paper, we firstly employ a serial of fault localization techniques on locating the real faults in novice programs, and the results shown that their effectiveness are not so good. One reason is the difference between artificial defects and real faults in novice programs, since the latter ones are more complex than the former ones. Another reason is that the test cases used in OJ system are designed for fault detection, not fault localization. Our work in this paper tries to improve the fault localization accuracy from the test cases point, where fine-grained test cases are introduced to do fault localization. Moreover, a Test Case Granularity(TCG) concept is proposed to analyze the correlation between test case and fault localization accuracy. To facilitate the evaluation of future novice program fault localization techniques, our source code and data set used in this paper are all available in the GitHub repository [1].

The contributions of this paper are summarized as follows:

- We evaluate the effectiveness of several fault localization techniques on novice programs and use fine-grained test cases to improve their fault localization accuracy.
- We introduce the Test Case Granularity (TCG) concept to better define fine-grained test cases, and the correlation between TCG and fault localization accuracy is studied.
- Empirical studies are conducted on 77 real novice faulty programs from OJ system, and the results indicate that: (1) the accuracy of fault localization techniques can be improved by using fine-grained test cases; (2) there is a strong correlation between fault localization accuracy and granularity of test cases.
- Wilcoxon signed-rank test is employed to compare the fault localization accuracy of test cases with different TCG. The results show that the fault localization accuracy with test cases of TCG-1 outperforms other test cases, where the confidence level is more than 99% in most cases.

The rest of this paper is structured as follows. Section II presents the related work of this paper. In Section III, we present our approach in detail. Experimental setup is introduced in Section IV. In Section V, we discuss the experimental results which demonstrate the influences of test case granularity on fault localization accuracy. Section VI presents the threats to validity and Section VII concludes this work.

II. Related Work

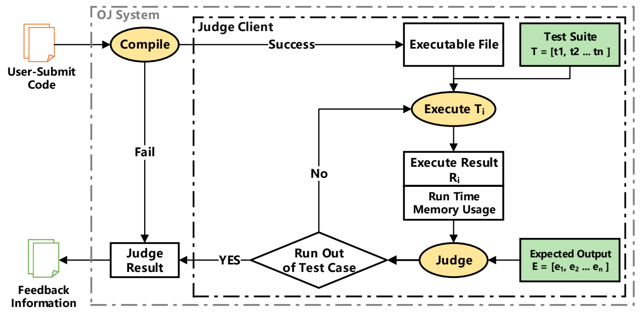
A. Online Judge System

Online Judge(OJ) system was originally used to judge programs submitted to solve problems and rank contestants automatically in programming competitions like ACM-ICPC (International Collegiate Programming Competition) and OI (Olympiad in Informatics) [16]. Nowadays, it is widely used in practicing programming skills for students, the training and selection of contestants, and the automatic submission and judgment of programming courses. For instance, the National University of Singapore has already successfully applied OJ system in the teaching of Algorithms and Data Structures courses [1].

Typically, OJ system managers should prepare multiple test cases for each problem, 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

[1]https://github.com/Dearyyyyy/TCG.git

Fig. 1. Framework of OJ System



system will return the judging result of each program. Figure 1 gives an overview of the framework of OJ system.

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

In recent years, some researchers paid attention to OJ programs related topics. Ahmed et el [17] attempted to fix compile errors in student programs by deep learning techniques. Drummond et el [7] 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 [3], [5], [6], [18].

In this paper, we aim to improve OJ system by providing fault-position-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 position of faults in programs is very time-consuming and costs a lot of human efforts. To alleviate the difficulty of debugging, a lot of fault localization techniques were proposed, such as SBFL [24], MBFL [25], [26], information retrieval-based fault localization(IR-based FL) [27], predicate switching [28], dynamic program slicing [29], stack trace analysis [30], history-based fault localization [31] and so on. Such fault localization techniques were widely studied in enterprise or open-source programs with artificial faults.

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 two widely studied fault localization techniques, SBFL and MBFL, 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 position of faults in programs by employing the spectrum information and execution results of test cases [32].

TABLE I
SUSPICIOUSNESS FORMULAS

name	CBFL formula	MBFL formula
$Jaccard$ [19]	$Sus(s) = \frac{fail(s)}{totalpass+pass(s)}$	$Sus(s) = \frac{a_{kf}}{a_{kf}+a_{nf}+a_{kp}}$
$Tarantula$ [20]	$Sus(s) = \frac{\frac{fail(s)}{totalfail}}{\frac{fail(s)}{totalfail}+\frac{pass(s)}{totalpass}}$	$Sus(s) = \frac{\frac{a_{kf}}{a_{kf}+a_{nf}}}{\frac{a_{kf}}{a_{kf}+a_{nf}}+\frac{a_{kp}}{a_{kp}+a_{np}}}$
$Ochiai$ [21]	$Sus(s) = \frac{fail(s)}{\sqrt{totalfail*(fail(s)+pass(s))}}$	$Sus(s) = \frac{a_{kf}}{\sqrt{(a_{kf}+a_{nf})*(a_{kf}+a_{kp})}}$
$OP2$ [22]	$Sus(s) = fail(s) - \frac{pass(s)}{totalpass+1}$	$Sus(s) = a_{kf} - \frac{a_{kp}}{(a_{kp}+a_{np})+1}$
$Dstar^*$ [23]	$Sus(s) = \frac{fail(s)^*}{pass(s)+(totalfail-fail(s))}$	$Sus(s) = \frac{a_{kf}^*}{a_{kp}+a_{nf}}$

TABLE II
AN EXAMPLE OF STATEMENT COVERAGE AND TEST CASE EXECUTION RESULTS

Statement	Program Code	Coverage	
		T1	T2
S_1	while True:	●	●
S_2	a, b, c = map(int, input().split())	●	●
S_3	if a+b>c and a+c>b and c + b>a:	●	●

S_4 (fault)	if a==b or a==c or b==c:
	     # (correct) if a==b and a!=c or a==c and a!=b or b==c and b!=a:
		●	●

S_5	print("isosceles triangle")	●	●
S_6	elif a*a + b*b == c*c or a*a + c*c == b*b or b*b + c*c == a*a:	●	●
S_7	print("right triangle")	●	●
S_8	elif a==b and b==c:		●
S_9	print("equilateral triangle")		
S_{10}	else:		
S_{11}	print("regular triangle")		●
S_{12}	else:		
S_{13}	print("error")		●
Execution Results		Pass	Fail

Fig. 2. Framework of SBFL

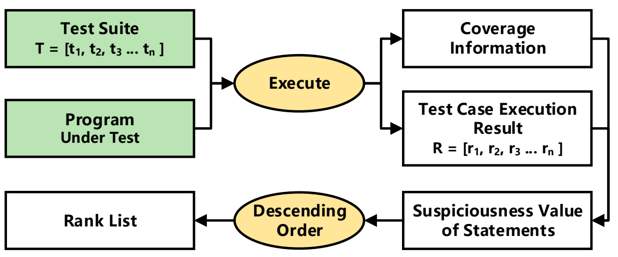


Figure 2 shows the framework of SBFL. It first collects coverage information and results (failed/passed) of test cases during the execution process. Next, SBFL uses suspiciousness formulas to calculate the probability of each statement contains bugs. Finally, a rank list will be generated by descending ordering all statements according to their suspiciousness values. The rank list is used to assist developers to find bugs quickly by checking the rank list from top to bottom. The higher rank of the exact faulty statement is, the better of the corresponding fault localization technique. 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 information and execution results of the test cases, as shown in Table III. Among the five formulas shown in Table I, DStar is a state-of-the-art technique proposed by Wong, and the parameter '$*$' of the DStar is set it to 3 in this paper according to the literature recommendation of Wong et al [23].

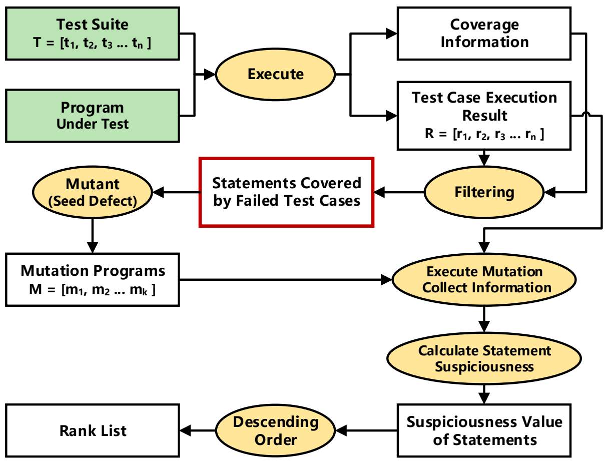
TABLE III
PARAMETERS FOR SBFL

$fail(s)$	Number of failed tests that execute statement s
$pass(s)$	Number of passed tests that execute statement s
$totalfail$	Total number of failed test cases
$totalpass$	Total number of passed test cases

SBFL is a light technique because it calculates the suspiciousness of each statement by using test cases' execution results and coverage information, such process avoids complex semantic and code analysis. However, the accuracy of SBFL techniques is often low to localize faults in real-world programs [11].

Table V lists the illustration example of how SBFL works in the novice program shown in Table II. Originally, there are two test cases in the OJ system to test the novice program shown in Table II, and from the middle part of Table V, it can be seen that the rank of the exact faulty statement (S4) is only higher than three other statements, which means that the fault localization accuracy is not so good to provide help for students fixing their bugs.

Fig. 3. Framework of MBFL


Test Suite
T = [t_1, t_2, t_3 ... t_n]
Program Under Test
Execute
Coverage Information
Test Case Execution Result
R = [r_1, r_2, r_3 ... r_n]
Filtering
Statements Covered by Failed Test Cases
Mutant (Seed Defect)
Mutation Programs
M = [m_1, m_2 ... m_k]
Execute Mutation Collect Information
Calculate Statement Suspiciousness
Suspiciousness Value of Statements
Descending Order
Rank List

2) Mutation-based fault localization: Mutation-based fault localization (MBFL) [25], [26], [33] employs mutation analysis to find the location of faults in software. Figure 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 statements will be calculated and a rank list will be generated to assist developers to find the faults in software [34]. A mutant can be considered as a similar version or a partial fix of real faults is the main foundation of MBFL [33].

From different aspects, there are mainly two kinds of MBFL techniques, MUSE [25] and Metallaxis [26]. MUSE considers each mutant as a partial fix of real faults, while Metallaxis considers each mutant as a similar version of real faults.

The key assumption of Metallaxis is that mutants which mutated from the faulty statements will show similar behaviors with the program under test [33]. Metallaxis uses similar suspiciousness formulas with SBFL, where five of them are listed in Table I. Take Jaccard for instance:

$$M(m) = \frac{a_{kf}}{a_{kf} + a_{nf} + a_{kp}}$$

Where, a_{np} means the number of passed test cases that did not kill m, a_{kp} indicates the number of passed test case that killed m, a_{nf} represents the number of failed test case that did not kill m,and a_{kf} counts the number of failed test cases that killed m. These formulas in MBFL are used to calculate the suspiciousness value of mutants, and there will be many mutants generated from mutating in one statement. The suspiciousness value of a statement s is defined as : $Sus(s) = Max_{m \in mut(s)}\{M(m)\}$.

MUSE considers each mutant as a partial fix of program under test, and then it assumes that mutating the faulty statement is more likely to turn failed test cases to passed test cases, and similarly, when the mutation occurs in the correct statement, it is more likely to turn passed test cases to failed test cases. Based on the above assumption, MUSE calculates the suspiciousness value of each statement by the following formula:

$$Sus(s) = \frac{1}{|mut(s)|} \sum_{m \in mut(s)} \left(\frac{|f_p(s) \cap p_m|}{|f_p|} - \alpha \frac{|p_p(s) \cap f_m|}{|p_p|} \right)$$

For a statement s of program P, $|mut(s)|$ 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. 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 α are used for adjusting the average values of the two terms to be the same [25].

Recent researches demonstrated that the two kinds of MBFL techniques, Metallaxis and MUSE, showed significantly advantage over state-of-the-art SBFL techniques [25], [26]. Moreover, Metallaxis are twice as much accurate as MUSE with only half of the cost [11]. In the rest of this paper, we chose Metallaxis as MBFL technique to conduct our experiments.

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

III. Fine-Grained Test Cases For Improving Fault Localization Accuracy

Table II shows an example of a novice program which tries to solve the ***Judge triangle*** problem in OJ system. It takes several lines of data as inputs, and each line contains three integers represent the sides of a triangle. The outputs are the corresponding kinds of triangles defined by the inputs. Table II, black dots indicate that statement s_i is covered by the corresponding test case T_i. The detail of the original test cases are displayed in Table IV. Usually, the OJ system will show simple test cases like T1 to students and hide complex test cases like T2. In this example, the OJ system will show a "WRONG ANSWER" result since T2 is failed on the program.

TABLE IV
TEST CASES OF PROGRAM IN TABLE II

Test cases		Input	Output
Original	T1	3 4 5	right triangle
		2 2 3	isosceles triangle
	T2	3 3 3	equilateral triangle
		2 3 4	regular triangle
		1 1 9	error
		8 6 10	right triangle
		7 7 3	isosceles triangle
Fine-grained	T3	3 3 3	equilateral triangle
	T4	2 3 4	regular triangle
	T5	1 1 9	error
	T6	8 6 10	right triangle
	T7	7 7 3	isosceles triangle

A. Fine-Grained Test Cases for Fault Localization

In terms of fault localization, Table V shows that the rank of faulty statement (S4) is 10 and 4 when using SBFL and MBFL respectively with the original test cases. Considering that there are only 13 statements in the whole program, such fault localization accuracy is poor.

Test cases that contain too many test tasks, or called as elements, have a negative effect on fault localization, especially those containing both test elements that can detect fault statements and test elements that cannot. The original test cases shown in Table V is a typical example, T2 tests multiple possible output results corresponding to multiple execution paths. If any execution path detected a fault, T2 will get a failed execution result which means a fault could be detected by T2. But on the contrary, if a test case failed, multiple execution paths would be considered to contain faults. Specifically, we have no idea which execution path has more chance to contain fault, and all the statements on those execution paths are suspected to be faulty.

Hence, to eliminate the phenomenon mentioned above and make sure that the suspiciousness value of irrelevant statements will not be increased by mistake, we attempt to refine test cases to split test elements into different test cases(shown in Table IV fine-grained part). The fault localization results are shown in Table V right part indicates test cases do affect the fault localization accuracy, the fine-grained test cases could split each execution path clearly and get a higher fault localization accuracy than original test cases. In detail, the rank of faulty statement (S4) is 2 in both SBFL and MBFL techniques with fine-grained test cases, which is better than that with original test cases.

B. Test Case Granularity

To better study this relationship between test cases and fault localization accuracy, we introduce a *Test Case Granularity (TCG)* concept to represent the number of test elements contained in each test case.

Suppose we want to solve a problem whose possible result set is $TotalReqSet = \{r_1, r_2 ... r_n\}$. Then a test case Ti could be represented as $T_iCaseSet = \{< i_1, r_{i1} >, < i_2, r_{i2} > ... < i_m, r_{im} >\}$. We define a test element is a valid detailed input/output pair in a test case which could be formalized as $< a, b >$, correspondingly, where a is a valid input, and b is a possible element in TotalReqSet. So TCG could be count by $TCG(T_i) = sizeof(T_iCaseSet)$.

For instance, {'equilateral triangle', 'regular triangle', 'right triangle', 'isosceles triangle', 'error'} are the possible outputs of problem ***Judge triangle***, T_1 is represented as $T_1CaseSet = \{< 3\ 4\ 5, right\ triangle >, < 2\ 2\ 3, isosceles\ triangle >\}$ and obviously $TCG(T_1) = 2$. In Table IV fine-grained part is some new test cases with TCG equals to 1. While the TCG increasing, a test case could cover more possible results and have a higher fault detection capability.

Due to long-term usage habits, OJ system manager prefers to upload a few test cases whose test elements are unevenly distributed. To ease this problem, In this paper, we recombine different numbers of test elements into several new test cases according to different levels of TCGs, and then the impact of test cases on fault localization and fault detection is investigated.

IV. EXPERIMENT DESIGN

A. Research Questions

To evaluate the effectiveness of fine-grained test cases on fault localization, we conduct a number of empirical studies, and the following research questions are defined and investigated:

- **RQ1:** Compared to original test cases, how do fault localization techniques perform when using fine-grained test cases?

To answer this question, we apply SBFL and MBFL with original test cases and fine-grained test cases whose TCG=1 on 77 real programs from OJ platform, and then evaluate the results using the metrics described in Section IV-C.

- **RQ2:** How do test cases with different TCGs affect the performance of fault localization?

In this research question, we investigate the effect of test cases with different TCGs on the overall performance of SBFL and MBFL. Specifically, we change the TCG from 1 to 10 with a step size of 1 and evaluate the results.

- **RQ3:** When locating real faults in novice programs, which fault localization technique has better fault localization accuracy?

In this research question, two main kinds of fault localization techniques and five popularly-studied suspiciousness calculation formulas are employed to investigate how these different combinations affect the performance of fault localization on novice programs.

- **RQ4:** Do test cases with different TCGs affect the bug detection capability of OJ system?

This paper mentioned in Section II-A shows that the OJ system relies on the execution results of test cases to verify the

TABLE V
AN ILLUSTRATION EXAMPLE OF USING FINE-GRAINED TEST CASES TO IMPROVE FAULT LOCALIZATION

Statement	Original						Fine-grained								
	Test cases		SBFL (Ochiai)		MBFL (Ochiai)		Test cases					SBFL (Ochiai)		MBFL (Ochiai)	
	T1	T2	Sus	Rank	Sus	Rank	T3	T4	T5	T6	T7	Sus	Rank	Sus	Rank
S_1	●	●	0.707	10	0.707	5	●	●	●	●	●	0.447	5	0.447	3
S_2	●	●	0.707	10	0	13	●	●	●	●	●	0.447	5	0	13
S_3	●	●	0.707	10	1	4	●	●	●	●	●	0.447	5	0.577	1
S_4 (fault)	●	●	0.707	10	1	4	●	●		●	●	0.5	2	0.5	2
S_5	●	●	0.707	10	0	13	●				●	0.707	1	0	13
S_6	●	●	0.707	10	1	4		●		●		0	13	0	13
S_7	●	●	0.707	10	0	13				●		0	13	0	13
S_8		●	1	3	1	4		●				0	13	0	13
S_9			0	13	0	13						0	13	0	13
S_{10}			0	13	0	13						0	13	0	13
S_{11}		●	1	3	0	13		●				0	13	0	13
S_{12}			0	13	0	13						0	13	0	13
S_{13}		●	1	3	0	13			●			0	13	0	13
Execution Results	Pass	Fail					Fail	Pass	Pass	Pass	Pass				

correctness of programs. Therefore, in this research question, we explored the impact of TCG changes of test cases on the normal work of OJ system.

In summary, this paper concerns three aspects of test cases, TCG, fault localization ability, and fault detection ability. RQ1, RQ2 and RQ3 investigate the fault localization ability of different TCGs, and RQ4 analyze the correlation between TCG and fault detection ability.

B. Subject Programs

In the experimental study, we chose the programs submitted to a programming examination from an introductory Python programming course as subject programs. There are seven problems in the programming examination, and the brief introduction of them is listed as follows:

- **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 and test cases can be seen in Table II.
- **Palindrome:** It is a classical string problem. Palindrome is a special kind of string which is the same as the corresponding reversed string. For example, ”abcba” is a palindrome string, since the reversed string of ”abcba” is also the same as the original string. Students have to write a program to determine if the input string is a palindrome or not.
- **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.
- **Basketball game:** It is a simulation problem involving string manipulation. The input includes multiple lines of data, each line represents a basketball player's technical statistic string in a single game. For each line of input, students have to write a program to calculate the player's efficiency value by the given formula.
- **Minimum currency payment problem:** It is a typical greedy problem. Students need to design an algorithm to calculate the minimum number of currencies required to pay the given money number.
- **Full permutation:** It is a common problem in Data Structure. Users write a program to list all the permutations that can be formed by N integers from 1 to N, and output them in lexicographic order.
- **Rotate string:** This is a problem involving string manipulation and looping. Users determine if the two strings are rotated string. The rotation of string A refers to moving the leftmost character of A to the far right. For example, move the string ’abc’ once to ’bca’ and move twice to ’cab’, so ’abc’, ’bca’, ’cab’ are the rotated string.

It should be noticed there are 30 students in the examination and a total of 313 Python programs were submitted to solve these seven problems. After executing and judging by OJ system, the results showed that there are 122 programs were identified as faulty programs. In this paper, we choose 77 faulty programs of them to conduct empirical study, and other 45 faulty programs were excluded for the reasons like: (1) some programs contain syntax errors and can not be executed correctly; (2) some programs contain crashed errors when executing test cases; (3) some programs are too far to be correct and the whole code should be rewritten. Finally, 77 programs are selected as subject programs, whose detailed information, including problem title, the number of faulty programs, the number of statements, total number of faulty statements and the number of original test cases, are listed in Table VI.

C. Evaluation Metrics

1) EXAM Score: EXAM Score (EXAM) [34], [35] gives the percentage of statements that need to be examined until the faulty statement is located. EXAM Score is commonly used to evaluate the accuracy of fault localization techniques [36], [37], and a lower EXAM means it needs to check fewer program statements to find the exact faulty statement, then

TABLE VI
SUBJECT PROGRAMS

Problem	#Faulty Programs	Language	#LOC	#Faulty statements	#Original Test cases
Judge triangle	13	Python	14 - 26	16	2
Palindrome	3	Python	8 - 11	7	2
Division	15	Python	10 - 18	21	4
Basketball game	7	Python	19 - 32	13	3
Minimum currency payment problem	6	Python	13 - 67	8	2
Full permutation	7	Python	12 - 32	12	2
Rotate string	26	Python	8 - 15	45	2

the corresponding technique has a better fault localization accuracy. The EXAM formula is defined as follows:

$$EXAM = \frac{rank\ of\ the\ faulty\ statement}{number\ of\ the\ executable\ statements} \quad (1)$$

The numerator in Equation 1 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 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 [11], [12], [38]. 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 share the same suspiciousness value as S is B, then the final rank of the exact faulty statement S is $\frac{(A+1)+(A+B)}{2}$.

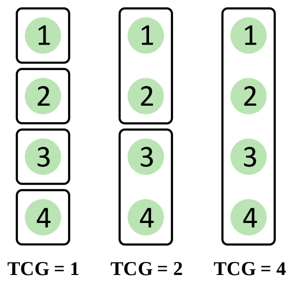
Besides, this equation would be disturbed when multiple faulty statements exist. For example, multiple faulty statements will result in multiple EXAM score. To overcome this problem, we would only evaluate the cost of locating the first buggy statement in rank list when there are multiple faulty statements in a program. This strategy is widely used to evaluate the ability to locate the first buggy statement for a bug [35], [39], [40].

2) TOP-N: This metric reports the number of bugs, whose buggy statements can be discovered by examining the top N (N=1,2,3,...) statements of the returned suspiciousness ranking list of code entities [39], [41], [42]. The higher the value of the TOP-N metric, the fewer efforts required for developers to locate the bug, 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 [43] suggested that programmers check only the first few statements in the ranking list, and TOP-N metric reflects this.

3) Test Case Granularity: To design different granularity of test cases for each program under test, firstly we split each test case into multiple test elements. Secondly, we have designed some additional test elements for each program to ensure that each executable statement is executed at least once and to guarantee there are enough test elements to restructure new test cases with different TCGs. Then we combine different numbers of test elements into test cases according to different values of TCGs.

An illustrative example of TCG is shown in Figure 4, where the circles numbered 1 to 4 represent 4 test elements. When TCG is 1, each test case will contain 1 test elements, and when TCG is 4, each test case will contain 4 test elements.

Fig. 4. An Example of Test Case Generation



4) Wilcoxon Signed-Rank Test: Additionally, to compare the fault localization techniques in different situations from the statistical view, we employ the Wilcoxon Signed-Rank Test [44] to evaluate the confidence level of fault localization technique with test cases of TCG-1 outperforms that with other kinds of test cases.

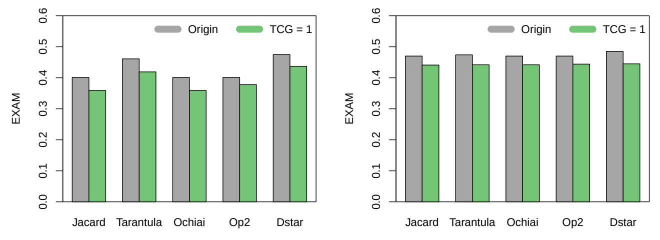
In this paper, all the experiments were performed on Linux system (version 3.10.0-957.el7.x86-64) with 18 cores CPU (Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz). We collected most of the common mutation operators [25], [26] and adjusted them for python programs. Program coverage information is collected by *coverage* [2] tool.

V. RESULTS ANALYSIS

A. RQ1: Improvement of fault localization performance when using Fine-Grained Test Cases

To answer RQ1, we compared the fault localization accuracy of each technique with two kinds of test cases, original and fine-grained (TCG-1), and results in terms of EXAM metric are shown in Figure 5, which show that using fine-grained test cases can improve the fault localization accuracy compared with using original test cases, where the original test cases refer to the unrefined test cases that are actually applied to our OJ system without any change.

Fig. 5. Fault localization performance comparison between original test cases and fine-grained test cases



(a) SBFL (b) MBFL

TABLE VII
FAULT LOCALIZATION PERFORMANCE COMPARISON WITH DIFFERENT KINDS OF TEST CASES

Technique	Formula	EXAM		
		Origin	TCG - 1	Impr(%)
SBFL	Jaccard	0.401	0.359	10.47%
	Tarantula	0.461	0.419	9.11%
	Ochiai	0.401	0.359	10.47%
	OP2	0.401	0.378	5.74%
	Dstar	0.475	0.437	8.00%
MBFL	Jaccard	0.470	0.441	6.17%
	Tarantula	0.474	0.442	6.75%
	Ochiai	0.470	0.442	5.96%
	OP2	0.470	0.444	5.53%
	Dstar	0.485	0.445	8.25%
Average				7.65%

Table VII shows the detailed data of Figure 5, and its results demonstrate that the EXAM value of each fault localization technique is smaller when using fine-grained test cases, and the improvement ratio varies from 5.53% to 10.47%. On average, using fine-grained test cases can improve 7.65% fault localization performance. What's more, since students' programs are generally not very long, another metric, TOP-N, should have a better evaluation of fault localization performance. As shown in Table VIII, in terms of TOP-1 and TOP-3 metrics, fault localization techniques with fine-grained test cases perform better in all fault localization formulas. What's more, in the real world, with the help of fault localization techniques using fine-grained test cases, about 25% of student's submissions could find their bugs by checking no more than 3 statements.

In summary, for **RQ1**, using fine-grained test cases can obviously improve the performance of all fault localization techniques.

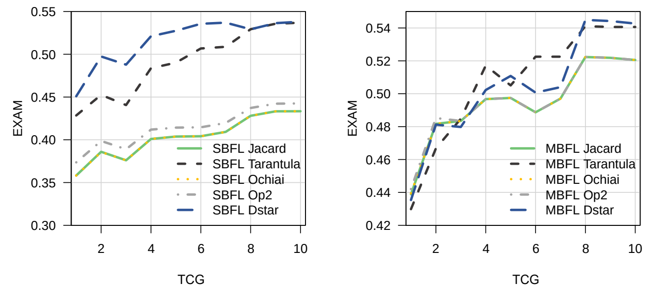
B. *RQ2: Correlation of fault localization accuracy and TCG*

To answer RQ2, we compare the fault localization accuracy of 10 techniques when using test cases with a series of different TCGs. Figure 6 plots the average fault localization results of 10 techniques with test cases of different TCGs in terms of EXAM metric for all subjects programs. More specifically, the x-axis represents the TCG introduced in Section IV-C3 and the y-axis is the values of EXAM of all techniques with test cases of corresponding TCG.

TABLE VIII
COMPARISON OF THE PERFORMANCE OF FAULT LOCALIZATION

Tech	Formula	TOP-1		TOP-3		TOP-5	
		origin	fine	origin	fine	origin	fine
SBFL	Jaccard	5	9	16	21	37	48
	Tarantula	5	9	8	16	20	28
	Ochiai	5	9	16	21	37	48
	OP2	5	8	16	21	37	48
	Dstar	0	5	4	13	24	32
MBFL	Jaccard	4	9	22	23	28	27
	Tarantula	4	9	17	24	23	27
	Ochiai	4	9	22	23	28	27
	OP2	4	9	22	23	28	27
	Dstar	2	9	18	24	22	28

Fig. 6. Fault localization performance comparison of different situations in terms of EXAM metric



The results shown in Figure 6 indicate that the performance of SBFL and MBFL with all formulas share similar patterns. The less of TCG value, the lower the corresponding EXAM of each fault localization technique. It should be noted that a lower EXAM value indicates a better fault localization accuracy, and when TCG is set as 1, all fault localization techniques obtain their best performance.

We further compare the TOP-N values of various fault localization techniques with test cases of under different TCGs, and results are shown in Figure 7. It can be clearly seen that the value of TOP-N is the largest when TCG is 1, and it becomes significantly smaller as the TCG increases.

From the statistical analysis point, a Wilcoxon signed-rank test is employed to further verify the credibility of the conclusion. More specifically, since it aims to show that the EXAM value of TCG-1 is lower than other TCG levels, so the hypothesis used in this paper is like this, H_0: The EXAM value calculated by each fault localization technique with test cases of TCG-1 is smaller than that with test cases of other TCGs. Table IX presents the confidence level of H_0 towards other situations. Take the first data in the second column and third row for example, it can be said that the confidence level of SBFL technique of Jaccard formula with TCG-1 test cases has better performance than that with Original test cases is 99.99%. As Table IX shows, the confidence level to accept the hypothesis in most situations is more than 99%. So it is safe to conclude that the performance of fault localization techniques with TCG-1 test cases is better than other situations.

[2]https://pypi.org/project/coverage/

Fig. 7. Fault localization performance comparison of different situations in terms of TOP-1, TOP-3 and TOP-5 metrics

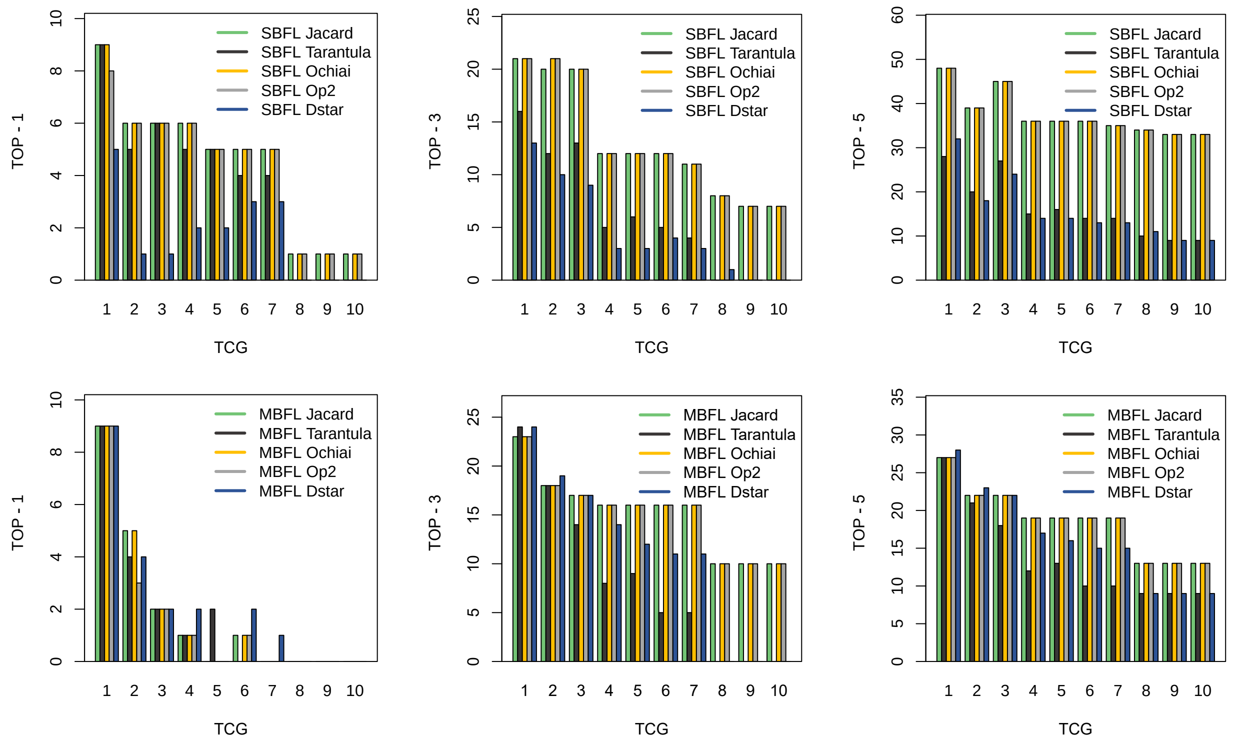


TABLE IX
CONFIDENCE OF HYPOTHESIS H_0 IN SECTION IV-C4

Level of TCG	SBFL					MBFL				
	Jaccard	Tarantula	Ochiai	OP2	Dstar	Jaccard	Tarantula	Ochiai	OP2	Dstar
Origin	99.99%	99.90%	99.99%	99.99%	99.66%	99.56%	92.21%	99.56%	97.96%	82.76%
TCG-2	99.96%	94.65%	99.94%	99.94%	99.98%	99.34%	94.07%	99.34%	99.90%	99.63%
TCG-3	98.43%	64.81%	98.43%	98.41%	99.36%	99.28%	97.55%	99.20%	99.20%	97.18%
TCG-4	99.99%	99.98%	99.99%	99.96%	99.99%	99.76%	99.90%	99.76%	99.76%	98.88%
TCG-5	99.99%	99.99%	99.99%	99.98%	99.99%	99.82%	99.83%	99.82%	99.82%	99.85%
TCG-6	99.99%	99.99%	99.99%	99.98%	99.99%	97.34%	99.97%	97.34%	97.34%	97.13%
TCG-7	99.99%	99.99%	99.99%	99.99%	99.99%	99.44%	99.97%	99.44%	99.44%	98.86%
TCG-8	99.99%	99.99%	99.99%	99.99%	99.97%	99.60%	99.99%	99.60%	99.60%	99.97%
TCG-9	99.99%	99.99%	99.99%	99.99%	99.98%	99.60%	99.99%	99.60%	99.60%	99.97%
TCG-10	99.99%	99.99%	99.99%	99.99%	99.98%	99.41%	99.99%	99.41%	99.41%	99.97%

In summary, for **RQ2**, there is a strong correlation between TCG and fault localization accuracy, and all fault localization techniques will obtain its best performance when the TCG is the lowest.

C. RQ3: When using fine-grained test cases, which fault localization technique is better?

To answer RQ3, we analyze the fault localization accuracy results of all techniques with fine-grained test cases evaluated by TOP-1, TOP-3, TOP-5 and EXAM metrics, and the results are shown in Table X. Cells highlighted with green color denotes the corresponding technique has the best performance in terms of the corresponding metric.

The results shown in Table X indicate that, SBFL techniques perform better than MBFL in terms of TOP-5 and EXAM metrics, while MBFL techniques are better than SBFL in

TABLE X
FAULT LOCALIZATION PERFORMANCE COMPARISON OF DIFFERENT TECHNIQUES WITH FINE-GRAINED TEST CASES

Technique	Formula	TOP-N			EXAM
		1	3	5	
SBFL	Jaccard	9	21	48	0.358
	Tarantula	9	16	28	0.428
	Ochiai	9	21	48	0.358
	OP2	8	21	48	0.374
	Dstar	5	13	32	0.451
MBFL	Jaccard	9	23	27	0.439
	Tarantula	9	24	27	0.430
	Ochiai	9	23	27	0.439
	OP2	9	23	27	0.442
	Dstar	9	24	28	0.435

terms of TOP-1 and TOP-3 metrics. Since TOP-1 and TOP-3 metrics are more important than TOP-5 and EXAM, so MBFL techniques have better performance than SBFL when locating real faults in novice programs. But the advantage of MBFL against SBFL is not significant. Previous studies showed that the fault localization accuracy of MBFL is much better than that of SBFL when locating artificial faults [34], [45], [46], but the results are different in this paper. The main reason is that artificial mutation operations cannot simulate real bugs in novice programs [35].

In summary, for **RQ3**, MBFL techniques are slightly better than SBFL techniques when locating novice programs with fine-grained test cases.

D. RQ4: Do test cases with different TCGs affect the bug detection capability of OJ system?

TABLE XI
OJ SYSTEM'S BUG DETECTION CAPABILITY WITH DIFFERENT TEST CASES

Level of test cases	#Detected wrong answers	#Total test cases	#Failed test cases	Failed/Total
Origin	77	191	141	73.82%
TCG - 1	77	730	385	52.73%
TCG - 2	77	501	319	63.67%
TCG - 3	77	372	281	75.53%
TCG - 4	77	312	260	83.33%
TCG - 5	77	281	259	92.17%
TCG - 6	77	265	243	91.69%
TCG - 7	77	256	243	94.92%
TCG - 8	77	240	232	96.66%
TCG - 9	77	240	232	96.66%
TCG - 10	77	240	233	97.08%

To answer RQ4, we compared the number of wrong answers detected in subject programs with different test cases. 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 cases designed for each problem. Previous results of this paper show that fine-grained test cases can assist fault localization, but it should under the premise that the fault detection ability will not be affected by replacing original test cases with fine-grained test cases. The results shown in Table XI clearly assure that premise is satisfied. Furthermore, no matter what level of TCG is, test cases can identify all the 77 wrong answers, so there will be no false positive identification of OJ system.

Traditionally, an OJ system just provides the result of a submission as right or wrong. But nowadays, some OJ systems can grade the submission according to the ratio of passed test cases. For example, If a program passed all test cases, it will get a full score on the problem, but if it passes half of the test cases, it will get 50% score. The are results shown in Table XI also indicate that fine-grained test cases can support scoring wrong answers better than original test cases.

VI. Threats to Validity

Threats to Internal Validity

Threats to internal validity relate to experimental errors and biases. During the locating phase, we have selected two SBFL and MBFL fault localization techniques and five suspiciousness calculation formulas, which strategies contain widely studied programs with real faults [47]–[51], which makes these threats limited.

Threats to External Validity

The threat of external validity is related to the scalability of our proposed method. We use 77 real-world novice programs as the experimental dataset. Furthermore, both single-fault and multiple-faults programs are considered. But our conclusions may be limited by the choice of language (Python).

Threats to Construct Validity

The threat to construct validity is related to the scalability of our proposed method. Threats to construct validity includes 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 [52] and ranking techniques [53].

VII. Conclusion

In this paper, we firstly investigate the effect of two widely studied fault localization accuracy on novice programs in OJ system, and then propose fine-grained test cases to improve fault localization accuracy. Besides, a test case granularity (TCG) concept is introduced and the correlation between TCG and fault localization accuracy is analyzed.

We evaluate the performance of our proposed method on 77 real novice programs from a python programming course, and the experimental results show that TCG will significantly affect the performance of fault localization accuracy. All fault localization techniques will perform the best when using fine-grained test cases, while the fault detection ability remains the same as original test cases in OJ system.

In the future, we plan to employ more methods to improve the fault localization accuracy on novice programs. The potential directions include: (1) using higher-order MBFL to locate multi-fault programs; (2) combining static structure and dynamic semantic information to assist fault localization; (3) employing machine learning algorithms to predict the fault position.

References

[1] S. Wasik, M. Antczak, J. Badura, A. Laskowski, and T. Sternal, "A survey on online judge systems and their applications," *Acm Computing Surveys*, vol. 51, no. 1, pp. 1–34.

[2] K. Masters, "A brief guide to understanding moocs," *The Internet Journal of Medical Education*, vol. 1, no. 2, p. 2, 2011.

[3] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," *Acm Sigplan Notices*, vol. 48, no. 6, p. 15.

[4] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "A feasibility study of using automated program repair for introductory programming assignments," 2017.

[5] S. Gulwani, I. RadiÄek, and F. Zuleger, "Feedback generation for performance problems in introductory programming assignments," 2014.

[6] S. Gulwani, I. Radiček, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 465–480, 2018.

[7] A. Drummond, Y. Lu, S. Chaudhuri, C. Jermaine, J. Warren, and S. Rixner, "Learning to grade student programs in a massive open online course," in *2014 IEEE International Conference on Data Mining*. IEEE, 2014, pp. 785–790.

[8] W. Zhou, Y. Pan, Y. Zhou, and G. Sun, "The framework of a new online judge system for programming education," in *Proceedings of ACM Turing Celebration Conference-China*, 2018, pp. 9–14.

[9] V. J. Marin, T. Pereira, S. Sridharan, and C. R. Rivero, "Automated personalized feedback in introductory java programming moocs," in *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 2017, pp. 1259–1270.

[10] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: data-driven feedback generation for introductory programming exercises," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 481–495.

[11] S. Pearson, J. Campos, R. Just, G. Fraser, and B. Keller, "Evaluating and improving fault localization," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017.

[12] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[13] Y. Qi, X. Mao, L. Yan, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013.

[14] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," 2016.

[15] E. Araujo, M. Gaudencio, D. Serey, and J. Figueiredo, "Applying spectrum-based fault localization on novice's programs," in *2016 IEEE Frontiers in Education Conference (FIE)*, Oct 2016, pp. 1–8.

[16] J. Wu, S. Chen, and R. Yang, "Development and application of online judge system," in *2012 International Symposium on Information Technologies in Medicine and Education*, vol. 1. IEEE, 2012, pp. 83–86.

[17] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani, "Compilation error repair: for the student programs, from the student programs," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, 2018, pp. 78–87.

[18] K. Rivers and K. R. Koedinger, "Data-driven hint generation in vast solution spaces: a self-improving python programming tutor," *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 37–64, 2017.

[19] P. Jaccard, "Etude de la distribution florale dans une portion des alpes et du jura," *Bulletin De La Societe Vaudoise Des Sciences Naturelles*, vol. 37, no. 142, p. 547â57, 1901.

[20] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," *International Conference on Software Engineering*, pp. 467–477, 2005.

[21] A. Rui, P. Zoeteweij, and A. J. C. V. Gemund, "An evaluation of similarity coefficients for software fault localization," *Pacific Rim International Symposium on Dependable Computing*, pp. 39–46, 2006.

[22] L. Naish, J. L. Hua, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *Acm Transactions on Software Engineering and Methodology*, vol. 20, no. 3, pp. 1–32, 2011.

[23] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.

[24] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, Sep. 2007, pp. 89–98.

[25] S. Yoo, S. Moon, Y. Kim, and M. Kim, "Ask the mutants: Mutating faulty programs for fault localization," 2014.

[26] M. Papadakis and Y. Le Traon, "Metallaxis-fl: Mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, 09 2013.

[27] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 14–24.

[28] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 272–281.

[29] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*. IEEE, 1995, pp. 143–151.

[30] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 181–190.

[31] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 489–498.

[32] F. Feyzi and S. Parsa, "A program slicing-based method for effective detection of coincidentally correct test cases," *Computing*, vol. 100, no. 9, pp. 927–969, 2018.

[33] D. Shin and D.-H. Bae, "A theoretical framework for understanding mutation-based testing methods," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 299–308.

[34] Yong, Liu, Zheng, Li, Ruilian, Zhao, Pei, and Gong, "An optimal mutation execution strategy for cost reduction of mutation-based fault localization."

[35] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[36] W. E. Wong, V. Debroy, and D. Xu, "Towards better fault localization: A crosstab-based statistical approach," *IEEE Transactions on Systems Man & Cybernetics Part C*, vol. 42, no. 3, pp. 378–396, 2012.

[37] R. Gao and W. E. Wong, "Mseer-an advanced technique for locating multiple bugs in parallel," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2017.

[38] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 314–324.

[39] M. Wen, J. Chen, Y. Tian, R. Wu, D. Hao, S. Han, and S.-C. Cheung, "Historical spectrum based fault localization," *IEEE Transactions on Software Engineering*, 2019.

[40] J. Sohn and S. Yoo, "Fluccs: using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 273–283.

[41] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–30, 2017.

[42] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 177–188.

[43] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 international symposium on software testing and analysis*, 2011, pp. 199–209.

[44] V. R. Prybutok, "An introduction to statistical methods and data analysis," *Technometrics*, vol. 31, no. 3, pp. 389–390, 2012.

[45] M. Papadakis and Y. Le Traon, "Using mutants to locate "unknown" faults," *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, 04 2012.

[46] ——, "Effective fault localization via mutation analysis: a selective mutation approach," in *Proceedings of the 29th annual ACM symposium on applied computing*, 2014, pp. 1293–1300.

[47] S. Pearson, "Evaluation of fault localization techniques," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 1115–1117.

[48] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 261–272.

[49] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3, pp. 158–171, 1996.

[50] R. Gopinath, C. Jensen, and A. Groce, "Mutations: How close are they to real faults?" in *2014 IEEE 25th International Symposium on Software Reliability Engineering*. IEEE, 2014, pp. 189–200.

[51] T.-D. B. Le, D. Lo, and F. Thung, "Should i follow this fault localization toolâs output?" *Empirical Software Engineering*, vol. 20, no. 5, pp. 1237–1274, 2015.

[52] A. Moffat and J. Zobel, "Rank-biased precision for measurement of retrieval effectiveness," vol. 27, no. 1, pp. 1–27, 2008.

[53] S. Yue, A. Karatzoglou, L. Baltrunas, M. Larson, and A. Hanjalic, "Climf: Learning to maximize reciprocal rank with collaborative less-is-more filtering," in *6th ACM Recommender Systems (RecSys)*, 2012.