The Journal of Systems & Software 205 (2023) 111822

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

VsusFL: Variable-suspiciousness-based Fault Localization for novice programs☆

Zheng Li [a], Shumei Wu [a], Yong Liu [a,*], Jitao Shen [a], Yonghao Wu [a], Zhanwen Zhang [a], Xiang Chen [b]

[a] College of Information Science and Technology, Beijing University of Chemical Technology, Beijing, China
[b] School of Information Science and Technology, Nantong University, Nantong, China

ARTICLE INFO

Article history:
Received 11 November 2022
Received in revised form 1 August 2023
Accepted 4 August 2023
Available online 7 August 2023

Keywords:
Novice programs
Fault localization
Variable value sequence
Sequence mapping

ABSTRACT

Automatically localizing faulty statements is a desired feature for effective learning programming. Most of the existing automated fault localization techniques are developed and evaluated on commercial or well-known open-source projects, which performed poorly on novice programs. In this paper, we propose a novel fault localization technique VsusFL (Variable-suspiciousness-based Fault Localization) for novice programs. VsusFL is inspired by simulating the manual program debugging process and takes advantage of variable value sequences. VsusFL can trace variable value changes, determine whether the intermediate state of the variables is correct, and report the potential faulty statements for novice programs. This paper presents the implementation of VsusFL and conducts empirical studies on 422 real faulty novice programs. Experimental results show that VsusFL performs much better than Grace, ANGELINA, VSBFL, Spectrum-Based Fault Localization (SBFL), and Variable-based Fault Localization (VFL) in terms of *TOP*-1, *TOP*-3, and *TOP*-5 metrics. Specifically, VsusFL can localize 90%, 35% and 9% more faulty statements than the best-performing baseline Grace. Moreover, We analyze the correlation between VsusFL and other techniques and find a weak correlation since they perform well on different programs, indicating the potential to further enhance fault localization performance through strategic integration of VsusFL with other methods.

1. Introduction

The rapid adoption of information technology brings light to program education and training (Reis et al., 2019; Guo, 2015). A user-friendly human–computer interaction programming method is critical for effective learning programming, and one of the most critical concerns is to localize faults accurately and provide valuable feedback during programming.

Usually, a beginner will face various faults during the process of learning programming. Among them, non-logical errors such as compilation and runtime can generally be detected or even automatically repaired by modern compilers. For logical failures that can be successfully compiled, developers usually insert assertions into the program and trace incorrect variable values to find the faulty statement. However, beginners usually lack programming experience and have difficulty localizing faults and fixing them quickly and accurately via manual debugging.

One promising way to improve students' programming efficiency is automatic debugging. Software debugging typically involves three phases: program inspection, Fault Localization (FL), and bug fixing (Howden, 1978). During debugging, FL is one of the most time-consuming activities and a prerequisite for fixing bugs, which motivates numerous studies on automated FL techniques (Weimer et al., 2009; Wen et al., 2018; Kim et al., 2021; Ju et al., 2014). Existing automated FL techniques have been widely used by developers. This technique usually generates a list of suspicious entities (such as statements or functions), and most of the faulty entities appear at the top of the list (Pearson et al., 2017). Many different FL methods have been proposed (Li et al., 2021c). For example, Abreu et al. (2007) used coverage information to assign different suspiciousness to each statement. Moon et al. (2014), Papadakis and Le Traon (2015) used mutated programs to localize faults. Zhou et al. (2012) proposed BugLocator, which used information retrieval to localize the relevant faulty files.

Although many studies have examined FL techniques in established open-source projects and commercial software (Wong et al., 2016; Wen et al., 2019; Zou et al., 2019; Gao and Wong, 2017), novice programs, which are written by new programmers (Johnson, 1990; Spohrer and Soloway, 1986), have not been

☆ Editor: Gabriele Bavota.
* Corresponding author.
E-mail addresses: lyong@mail.buct.edu.cn (Y. Liu), appmlk@outlook.com (Y. Wu).

https://doi.org/10.1016/j.jss.2023.111822
0164-1212/

extensively researched (Li et al., 2021d). Existing research highlights substantial differences between novice programs and those developed by experienced developers in commercial or open-source contexts (Mow, 2008). Novice programmers tend to avoid complex operations, perceiving them as challenging, and often use repetitive code structures rather than more sophisticated functions/classes (Lahtinen et al., 2005; Pane and Myers, 1996). Consequently, novice programs typically feature fewer custom functions and branching structures, with much of the code concentrated in a few statement blocks. This ultimately results in traditional FL techniques performing poorly in identifying faults within novice programs.

The primary challenge of using common FLs for novice programs arises from their dependence on execution paths and test case outcomes. These FLs that depend directly or indirectly on coverage are more effective when test cases provide more distinctive information for statements. This is because they can narrow down the scope required to localize faults by comparing the execution paths between passed and failed test cases. However, novice programs, with their limited custom functions and branching structures, tend to produce similar execution paths across all test cases, as they are constrained to a single or a few code statement blocks. This similarity leads to test cases with identical or nearly identical execution paths and results. Consequently, if high-frequency-executed code blocks contain faults, it is likely that all of the affected test cases will fail. As a result, many fault localization techniques cannot achieve satisfactory outcomes in novice programs. For instance, Qi et al. (2013) conducted experiments on 15 popular FL techniques with real novice programs. Experimental results showed the poor performance of those techniques. Besides, Araujo et al. (2016) tried to apply Spectrum-Based Fault Localization (SBFL) on novice programs. Their experimental results found that nearly 40% of novice programs did not even satisfy the preconditions for effective fault localization techniques, since some of them could not pass even one test case.

Therefore, traditional FL methods can barely discern differences between test cases in novice programs by comparing execution paths and results. As such, it is essential to propose a novel fault localization method that examines test case differences from alternative perspectives, thereby ensuring precise fault localization in novice programs. It has been observed that during manual program debugging, developers typically employ breakpoint debugging to assess whether variable changes throughout test case execution align with expectations. Deviations in expected variable changes often indicate the presence of program faults, and this process remains uninfluenced by code statement blocks or program branches. Based on this observation, we conjecture that comparing the sequences of variables offers a viable solution for distinguishing test cases in novice programs and can help to achieve highly accurate fault localization. Therefore, our proposed method should (1) automatically trace changes in variable values and (2) autonomously determine the correctness of intermediate variable states.

For the first one, static and dynamic program analysis can help to identify the variables and variable value sequence, which presents the intermediate values of the variable. For the second one, the correct intermediate values should be provided, however, which is impossible to obtain without the correct version of faulty programs. To obtain a sufficient number of correct program versions, we collect a large amount of code from the Online Judge (OJ) system. OJ system assesses program correctness by executing instructor-prepared test cases. A program is judged to be correct if it successfully processes all test cases and generates the expected output. Thus, each OJ test problem can receive numerous correct program versions submitted by students, and these versions are executed to derive the correct reference for the intermediate variable values. Additionally, we also note that these correct versions may be quite different due to the diverse problem-solving ideas of different programmers (Corbett and Anderson, 2001). Therefore, it is necessary to develop an effective method to find a correct version corresponding to the faulty novice program.

Based on these insights, we propose a novel fault localization technique VsusFL (**V**ariable-**sus**piciousness-based **F**ault **L**ocalization), to automatically localize faulty statements for novice programs. To simulate the manual debugging process, VsusFL first identifies the variables used in a program via static analysis. The tool 'CppSnooper' is developed to collect the variable value sequences of the program through program instrumentation and record the line number for each value in variable value sequences. Then, a matching algorithm is designed to identify a correct version for the same novice problem based on the similarity of variable mapping. Finally, VsusFL compares variable value sequences between the faulty program and the correct program, and calculates the suspiciousness value of each statement by identifying those variable values which may cause the failure.

In the empirical study, 422 faulty novice programs are selected from the real world as experimental subjects. Five state-of-the-art lightweight fault localization methods, Grace (Lou et al., 2021), ANGELINA (Mechtaev et al., 2016), SBFL (Abreu et al., 2007), VFL (Variable-based Fault Localization) (Kim et al., 2019), and VSBFL (Variable value Sequence based Fault Localization) (Li et al., 2021a), are also chosen as baselines. Experimental results show that VsusFL can outperform all baselines. Specifically, VsusFL can localize 90%, 35%, 9% and 265%, 43%, 12% faulty statements more than the two best-performing FL techniques, Grace and VSBFL, respectively, in $TOP-1$, $TOP-3$, and $TOP-5$, but slightly less than Grace and VSBFL in $A-EXAM$. Interestingly, VsusFL in our study is weakly correlated with other FL techniques, especially VSBFL, indicating the potential of combining them. Further experiments show that when combining VsusFL with VSBFL, the performance of the fault localization can be further improved.

We summarize the main contributions of this study as follows:

- We present a novel fault localization method VsusFL for automatically localizing faulty statements in novice programs.
- We implement VsusFL and conduct empirical studies on 422 real-world novice programs from 33 problems. The experimental results show that our proposed method can outperform the state-of-the-art FL techniques in terms of *TOP*-1, *TOP*-3, and *TOP*-5 metrics.
- We find that VsusFL in our study is weakly correlated with other FL techniques, indicating the potential of combining them. Three combination methods are presented to further improve the fault localization performance.

Reproduction package: To facilitate other researchers to re-implement the experiments, we share our experimental subjects and the source code of our method in GitHub.[1]

2. Background and related work

2.1. Program debugging

Program debugging is defined as a process of finding faults in programs and fixing them (Kim et al., 2018). It typically involves three phases: program inspection, fault localization, and bug fixing (Howden, 1978). Program debugging requires abilities of the software developers that not only are familiar with the

[1] https://github.com/appmlk/VsusFL

design purpose of the program, but also master its structure, which brings a lot of challenges (Xu and Rajlich, 2004; Wong et al., 2007). For beginners who lack programming experience, this process is even more challenging (Lahtinen et al., 2005). Therefore, the use of automatic program debugging techniques can help developers to localize the faults or repair the program more quickly.

In recent years, several tools have been developed to automatically generate feedback to help teachers and beginners. For example, Gulwani et al. (2018) developed Clara, which uses clustering to track the process of program repair in a cluster, and tried to repair other programs. Python Tutor (Guo, 2013) can visualize the running process of the program. However, they do not tell novices why their programs fail, and where the faulty statements might be. Therefore, a custom automatic debugging technique, particularly focusing on fault localization, is essential. Fault localization is often the most time-consuming and tedious activity in the debugging process (Kim et al., 2021). The success of automated program repair also heavily depends on the accuracy of automated fault localization (Weimer et al., 2009; Wen et al., 2018; Kim et al., 2021). Hence, implementing an effective automatic fault localization technique for novice programs is important.

2.2. Fault localization

Researchers have proposed various fault localization techniques, such as SBFL (Abreu et al., 2007), Mutation-Based Fault Localization (MBFL) (Moon et al., 2014; Papadakis and Le Traon, 2015), dynamic program slicing (Agrawal et al., 1995; Renieres and Reiss, 2003), history-based fault localization (Kim et al., 2007; Rahman et al., 2011), and learning-based fault localization (Li et al., 2019a). Some researchers also attempted to combine multiple techniques. For example, Jiang et al. (2019) combined SBFL with statistical debugging. Lou et al. (2020a) used automated program repair to improve fault localization.

Spectrum-Based Fault Localization. SBFL has attracted a lot of attention from researchers due to its simplicity, effectiveness, and low computational cost. Specifically, SBFL executes the program under test through all test cases in order to collect coverage information and execution results for each test case (i.e., passed or failed). SBFL then uses this information to calculate the suspiciousness for each statement in the program based on suspiciousness formulas. Finally, these statements can be sorted according to the suspiciousness. The higher a statement ranks on this suspiciousness list, the higher probability it contains the fault. Many effective suspiciousness formulas have been designed to improve the effectiveness of SBFL, such as *Jaccard* (Jaccard, 1901), *Tarantula* (Jones et al., 2002), *Ochiai* (Abreu et al., 2006), *OP*2 (Naish et al., 2011), and *Dstar** (Wong et al., 2013).

However, while SBFL has advantages in efficiency, it is not as precise as other fault localization methods (Feyzi and Parsa, 2018). For example, some statements will be computed to the same suspiciousness (i.e., tie issues) (Steimann et al., 2013; Wong et al., 2016), which could cause beginners considerable difficulty and inconvenience. Moreover, the tie issue will be more severe in novice programs, therefore additional features are needed to help beginners localize faults.

Learning-Based Fault Localization. The use of deep learning in software testing has motivated several studies on fault localization, such as Grace (Lou et al., 2021), DEEPRL4FL (Li et al., 2021b), ALBFL (Pan et al., 2020), and DeepFL (Li et al., 2019b). For example, Grace represents both code and coverage relationships between code and test cases in a graph, and then uses a graph neural network to predict faulty methods according to this graph. However, the current state-of-the-art these works only focus on method-level fault localization while ignoring statement-level fault localization. As a result, it becomes challenging to apply these techniques to student programs that require a more detailed localization at the statement level due to their small size. Additionally, some information (e.g., mutation) is time-consuming to collect, while other information (e.g., bug reports and code change history used in some research) cannot always be available, limiting their application in novice programs.

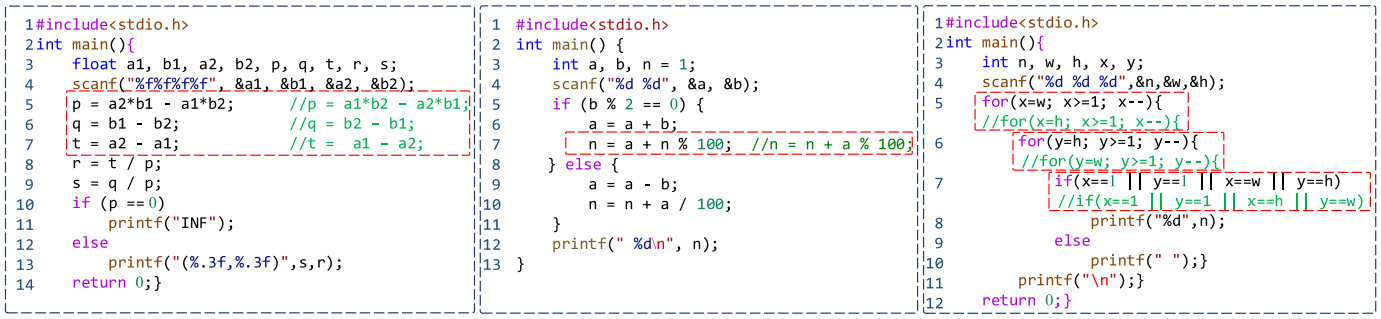
Value-Based Fault Localization. Some techniques similar to ours are VFL (Kim et al., 2019) and VSBFL (Li et al., 2021a), both of which utilize variable information to help localize faults. Specifically, VFL first extracts variables appearing in the source code and function invocations information. Then VFL assigns the suspicious ratio of the individual variables and function invocations according to coverage between them and tests, to calculate suspiciousness scores of individual statements. In fact, fault localization techniques struggle to utilize function invocation information in novice programs, due to beginners' lack of understanding of functions and encapsulation. In addition, VFL still depends on coverage to assign suspicious scores to variables, which suffers from the same issue as SBFL in novice programs. In other words, such a VFL technique cannot effectively localize faults when coverage differences between variables are unclear. Similarly, VSBFL assigns a degree of suspicion to a variable by comparing the variable value sequences in the incorrect program with the variable value sequences in the correct program. Consequently, each program statement containing the variable is attributed with an identical suspiciousness value. Thus, VSBFL remains incapable of effectively differentiating the suspiciousness among different statements.

Besides, ANGELINA (Mechtaev et al., 2016) is a tool supporting angelic debugging (Chandra et al., 2011), which is also a fault localization technique using finer information. Unlike our proposed VsusFL, ANGELINA works at an expression level. For each expression, ANGELINA analyses whether its result can be modified to reverse the results of failed tests while maintaining the results of the passed tests. Expressions that meet the above conditions are identified as potential repair expressions. However, such analysis requires symbolic reasoning, which needs high computational costs and has limited scalability.

2.3. Feedback generation methods for novice programs

There are many techniques that are proposed to provide feedback for novices. Existing work typically focuses on assisting novices in identifying faulty statements or repairing their programs based on fault localization.

Fault Localization on Novice Programs. Several studies have applied deep learning to improve the effectiveness of fault localization for student programs in recent years, including NBL (Gupta et al., 2019) and FFL (Nguyen et al., 2022). NBL, a deep learning based semantic fault-localization technique, aims to localize the faults in a faulty program utilizing failed tests. Toward this, NBL first represents the program as an Abstract Syntax Tree (AST) and then uses a tree convolutional neural network to predict whether or not the program passes a given test. Finally, NBL utilizes a neural prediction attribution (Sundararajan et al., 2017) to infer fault locations based on the prediction results. Like Grace, FFL utilizes more information to localize faults in student programs at the cost of increasing time and computing resources. Specifically, FFL uses a graph-based representation of a program that comprises syntax and program semantic information via AST and coverage into one graph. Then, FFL utilizes a graph neural network to predict faulty statements. While these techniques can be effective, their success often depends on the availability of substantial amounts of data, which may not always be accessible.



(a) bug lies in Simple EXpressions (SE) (b) bug lies in Conditional EXpressions (CD) (c) bug lies in loop statements (Loops)

Fig. 1. Motivation examples.

Furthermore, these techniques require significant investments of time and computing resources. In particular, training a specialized network for each problem is necessary to achieve optimal results. Compared to NBL and FFL, our proposed VsusFL is a lightweight technique. VsusFL only requires information about variable values and easily obtains variable value information without additional human efforts through the tool we provide.

Program Repair on Novice Programs. Some research has focused on repairing novice programs by repairing faulty statements, typically including InFix (Endres et al., 2019), Clara (Gulwani et al., 2018), and the technique proposed by Singh et al. (2013). Endres et al. (2019) proposed InFix, a randomized search algorithm for automatically repairing program inputs for novice programs. InFix can repair input data without test cases and special comments. Instead, they utilized patterns commonly used by novice programmers to generate input repairs automatically. Gulwani et al. proposed Clara (Gulwani et al., 2018), which aims to repair novice programs by clustering the existing correct beginner solutions. Clara can learn how to repair new incorrect attempts by selecting the target program from each cluster and execute a trace-based repair procedure. Lou et al. (2020b) proposed a technique for automatically providing feedback to novices that can complement manual and test cases based techniques. The technique uses an error model describing the potential corrections and constraint-based synthesis to compute minimal corrections to beginner's incorrect solutions. Most previous studies require model training, which requires many novice program samples (usually thousands of samples), and some have to train for each problem. Compared with these studies, VsusFL only needs dozens of programs to localize the fault and is universal in any problem in this study.

3. Motivation example

Test Scenario. To better describe our motivation, we first introduce our test scenario. With the development of computers and the popularity of programming education, various online programming platforms have emerged, including Leetcode,[2] Codechef,[3] Codeforces,[4] Jutge,[5] and BUCTCoder.[6] These platforms enable teachers to post programming tasks (problems) and provide students an online programming environment to submit their code. For the platform to automatically assess the correctness of the code, the teacher must provide the corresponding test cases. Since these test cases are typically created manually, the number of test cases provided is generally limited. Code that successfully passes all test cases is regarded as correct. Consequently, there may be several correct codes. All code submitted by students is saved in the database, which includes both correct and faulty programs. For faulty programs, the failed tests are returned to students as feedback. However, students may struggle to debug their programs if they receive no hints about where the faults are.

Motivating Examples. To better illustrate the limitations of existing fault localization techniques, we present three motivating examples in this section. Fig. 1 displays three programs written by students who are new to programming. The faults in these programs are located in simple expressions (SE), conditional expressions (CD), and loop structures (Loops), respectively. We highlight these faults using red dashed boxes and provide the correct code in green as comments. In addition, Table 1 further provides detailed information on the test cases and coverage for the three motivating examples. Column "st" presents the line number of the statement, while column "Coverage" presents statement-level coverage of both failed and passed tests (i.e., *ft* and *pt*). Columns "S","V" and "Vsus" rank the faulty probability of each statement, computed by Jaccard (Jaccard, 1901) (i.e., one of the most popular SBFL formulas), VFL (Kim et al., 2019) and our approach VsusFL. As shown in Table 1, these faulty statements are highlighted in gray.

Other FLs. Unfortunately, as shown by the table, SBFL fails to localize neither faulty statements within $TOP - 1$, since it always considers the statement covered by more failed tests and less passed tests as more suspicious. In fact, the beginner in programming lacks experience and may not be familiar with concepts such as functions and encapsulation. They often write code consisting of one or more code blocks Nguyen et al. (2022). In such cases, either the failed tests or passed tests cover most of the code. Therefore, SBFL is misled to make a wrong judgment Qi et al. (2013), Araujo et al. (2016), Steimann et al. (2013). In addition, SBFL cannot distinguish statements with similar coverage information, making it susceptible to encountering tie problems (Steimann et al., 2013; Wong et al., 2016). For example, in the three examples presented in Table 1, SBFL assigns the same rank to multiple statements because they have the same suspiciousness. Similar to Jaccard, other existing SBFL formulas also fail to rank faulty statements before correct ones, due to sharing the same ranking intuition. Also, a notable finding is that SBFL performs better for faults located in CD, with the faulty statement ranked within $TOP - 2$. This could be due to the coverage information being more discriminating within the branch structure, as demonstrated in Table 1. Another example family is VFL (Kim et al., 2019). VFL also fails to discover these faults within $TOP - 1$ using Tarantula (Jones et al., 2002) (one of the most widely used formulas). As analyzed in Section 2, VFL assigns suspiciousness scores to variables by directly utilizing test

[2] https://leetcode.cn
[3] https://www.codechef.com
[4] https://codeforces.com
[5] https://jutge.org
[6] https://buctcoder.com

Table 1
Coverage of there motivation examples, where 'S', 'V', and 'Vsus' represent SBFL, VFL, and VsusFL, respectively.

faults Lies in SE										faults Lies in CD								faults Lies in Loops										
st	Coverage						Rank			st	Coverage				Rank			st	Coverage							Rank		
	pt_1	pt_2	pt_3	pt_4	pt_5	ft_6	S	V	Vsus		pt_1	ft_2	ft_3	pt_4	S	V	Vsus		ft_1	ft_2	ft_3	ft_4	pt_5	pt_6	ft_7	S	V	Vsus
4	✓	✓	✓	✓	✓	✓	9	7	10	3	✓	✓	✓	✓	6	1	7	4	✓	✓	✓	✓	✓	✓	✓	8	6	8
5	✓	✓	✓	✓	✓	✓	9	3	10	4	✓	✓	✓	✓	6	5	8	5	✓	✓	✓	✓	✓	✓	✓	8	6	7
6	✓	✓	✓	✓	✓	✓	9	7	2	5	✓	✓	✓	✓	6	7	4	6	✓	✓	✓	✓	✓	✓	✓	8	6	1
7	✓	✓	✓	✓	✓	✓	9	7	1	6		✓	✓		2	4	4	7	✓	✓	✓	✓	✓	✓	✓	8	6	6
8	✓	✓	✓	✓	✓	✓	9	3	10	7		✓	✓		2	4	1	8	✓	✓	✓	✓	✓	✓	✓	8	6	6
9	✓	✓	✓	✓	✓	✓	9	3	10	9	✓			✓	8	4	7	10	✓	✓	✓	✓	✓		✓	1	8	6
10	✓	✓	✓	✓	✓	✓	9	7	10	10	✓			✓	8	6	7	11	✓	✓	✓	✓	✓	✓	✓	8	6	6
11		✓			✓		10	10	10	12	✓	✓	✓	✓	6	8	4	12	✓	✓	✓	✓	✓	✓	✓	8	8	6
13	✓		✓	✓		✓	1	10	10																			
14	✓	✓	✓	✓	✓	✓	9	10	10																			

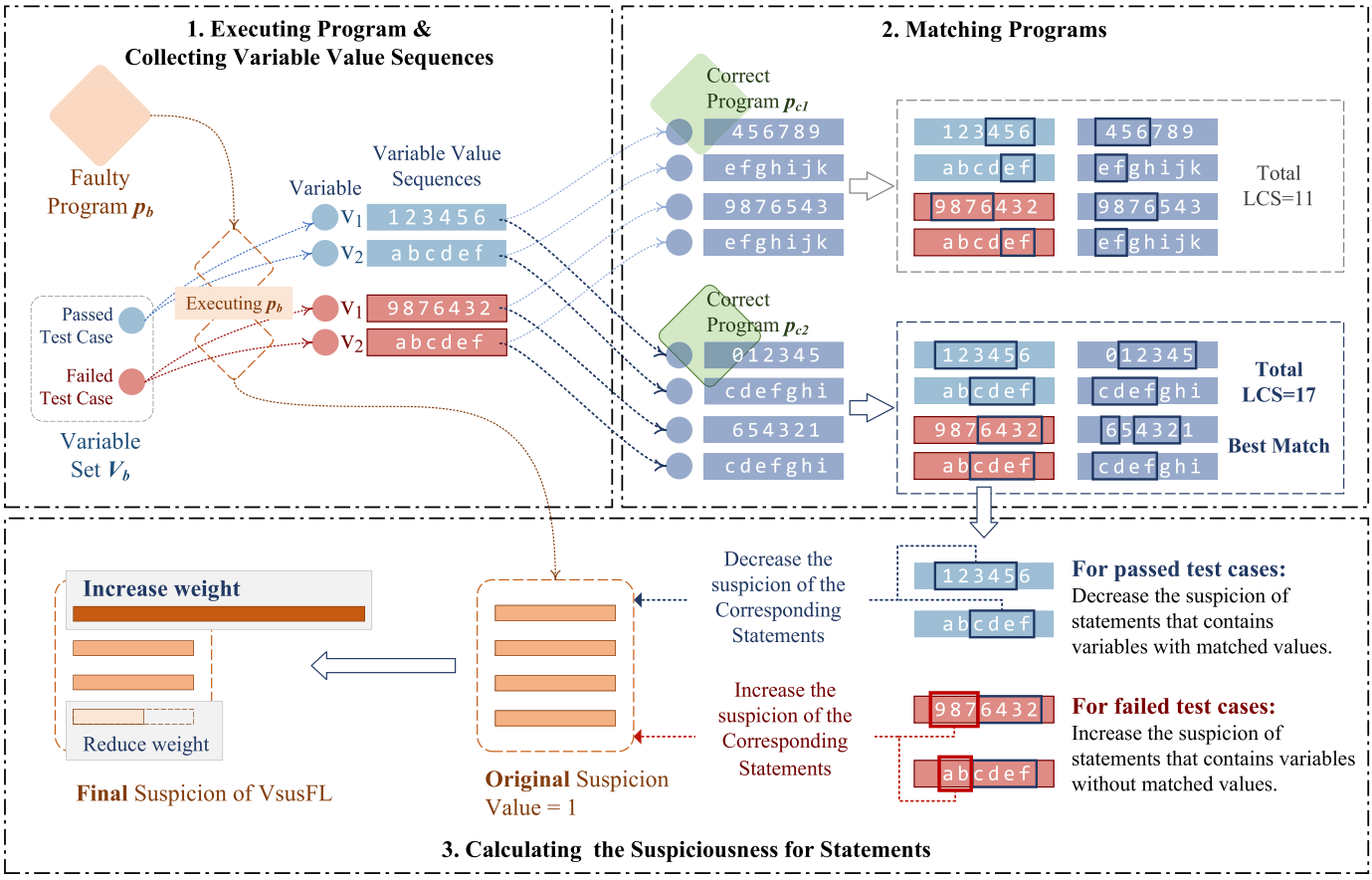


Fig. 2. Framework of VsusFL.

coverage and pass/fail results, which inherently suffers from the same issues as SBFL in novice programs. In particular, VFL cannot distinguish the suspicious degrees of different variables with similar coverage, which limits its usefulness in novice programs. To sum up, the effectiveness of FLs, which depend directly/indirectly on coverage, is challenged when the execution and results of test cases cannot provide identifiable information for different statements.

Our Approach VsusFL. Unlike the above FLs, our approach takes a value sequences view on fault localization. Let us consider the value sequences of all variables in the faulty program. Given value sequences of all variables in a correct program similar to the current faulty program as an observation, our approach assesses the probability of each variable value being faulty to estimate the probability that each statement produces faulty values efficiently. This allows for VsusFL to gain more effective fine-grained information when test cases are sparse, avoiding the inability to distinguish statements with similar coverage.

4. Method

In modern education scenarios, teachers often use OJ (Online Judge) systems to assist in teaching. In the OJ system, each problem may contain a large number of correct programs, and these programs will be stored in the database of the OJ system. VsusFL aims to compare these samples with the faulty program to localize the faults. Fig. 2 shows the framework of our proposed VsusFL. First, we obtain the variable value sequences of the faulty program by executing the test cases. Then in **Matching Similar Program** part, we match a faulty program with a correct program that implements similar logic, and **Fault Localization** part is to determine the correctness of variable values and calculate the suspiciousness of program statements to localize the faulty statements in the target faulty program.

To improve the efficiency of our proposed method VsusFL, we use 'CppSnooper'[7] to preprocess and store the variable value sequence of the correct program in the OJ platform. Moreover, we further added the ability to capture the line number of the variable value in this study. In the rest of this section, we will show the preliminary and then describe the detailed information of these steps.

4.1. Preliminary

The key step of matching similar programs is to obtain variable value sequences and calculate the similarities between these sequences. To better describe our method, we present a formal description of variable value sequences and their relationship.

4.1.1. Variable value sequence

Since a novice program under test is executed by a test suite, it can generate a series of variable value sequences. These sequences can be employed to assess the similarity between two programs, especially when they share similarities in their implementation. It is crucial to acknowledge that the value sequences of two correct programs implemented using distinct approaches may differ significantly due to different algorithms and data structures. However, when two programs have similar implementations, their value sequences tend to be more similar, thus enabling the evaluation of program similarity.

[7] https://github.com/appmlk/VsusFL

Consider a target OJ problem Q with a test suite T, which includes k test cases ($t_x \in T$, $x = 1, 2, \ldots, k$), and a program set P. Then n_i variables in the program p_i ($p_i \in P$) construct a variable set V_i ($v_{ij} \in V_i$, $j = 1, 2, \ldots, n_i$). These parameters represent the important factors of our technique. Then, we will show formal definitions in sequence.

Definition 1. A test case t_{target} ($t_{target} \in T$) of the test suite T is used to verify the correctness of the programs under test, which consists of standard input $T_{input_{target}}$ and expected output $T_{output_{target}}$.

For example, when submitting p_i for verification, OJ will execute p_i and input the standard input $T_{input_{target}}$. If the execution result equals to the expected output $T_{output_{target}}$, the test case t_{target} is passed for p_i, or t_{target} is failed.

Definition 2. Program set P consist of correct program set P_c and faulty program set P_f.

A program can be regarded as a correct program when all test cases are passed. Otherwise, this program can be regarded as a faulty program when at least a test case is failed. Notice that some of the submitted programs contain runtime errors or compile errors, which means they cannot generate valid execution results. In our study, we do not consider these programs since these programs can be effectively identified by the current OJ systems. Besides, such a filtering strategy of the program under test has been adopted by numerous fault localization studies (Gao and Wong, 2017; Kim et al., 2019; Wu et al., 2020).

Definition 3. Variable value sequences are value record lists of variables. For the program p_i, the variable v_{ij} ($v_{ij} \in V_i$), and the test case t_{target}, the initial value of v_{ij} is X_{ij_0}, and the value of variable v_{ij} after the mth change is X_{ij_m}, then the variable value sequence of v_{ij} in p_i is $E(p_i, v_{ij}, t_{target}) = \{X_{ij_0}, X_{ij_1}, \ldots, X_{ij_M}\}$ (Here the variable has changed M times).

Specifically, a variable change refers to an update in the value of a variable during the execution of a program, reflecting the program's underlying algorithm design and data structures. It represents a modification in the state of a variable as the program processes data and performs operations. The value after the mth change is the value of the variable after it has undergone m updates. By examining the variable value sequences, we can capture the changes in variable values throughout the program execution and use this information to assess the similarity between programs, as these sequences provide insight into the program's inherent characteristics.

In the next section, we will further examine how variable value sequences can be utilized to analyze program similarities and discuss related definitions.

4.1.2. Relations between sequences

Two programs can be mapped through their variable value sequences, and the similarities of the two sequences determine the similarities of the two programs. In the rest of this subsection, we describe the related definitions.

Definition 4. The similarity of two variables under each test case can be calculated through Largest Common Subsequence (LCS). LCS is used to determine the level of similarity between two strings. For the variable $v_{i_1j_1}$ in p_{i_1}, the variable $v_{i_2j_2}$ in p_{i_2}, and the test case t_{target}, the LCS of $v_{i_1j_1}$ and $v_{i_2j_2}$ is $LCS(p_{i_1}, v_{i_1j_1}, p_{i_2}, v_{i_2j_2}, t_{target})$, and its length is $S(p_{i_1}, v_{i_1j_1}, p_{i_2}, v_{i_2j_2}, t_{target})$.

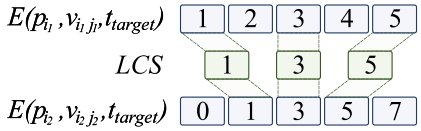


Fig. 3. Illustration on LCS calculation via an example.

Fig. 3 illustrates LCS calculation via an example, where $E(p_{i_1}, v_{i_1j_1}, t_{target}) = \{1, 2, 3, 4, 5\}$ and $E(p_{i_2}, v_{i_2j_2}, t_{target}) = \{0, 1, 3, 5, 7\}$. In this example, $LCS(p_{i_1}, v_{i_1j_1}, p_{i_2}, v_{i_2j_2}, t_{target}) = \{1, 3, 5\}$, and $S(p_{i_1}, v_{i_1j_1}, p_{i_2}, v_{i_2j_2}, t_{target}) = 3$, which means the length of LCS is three.

Definition 5. For multiple test cases, the similarity of two variables is the sum of LCS lengths under each test case. Consider the test suite T, the similarity of the variable $v_{i_1j_1}$ and $v_{i_2j_2}$ can be calculated as:

$$S_{total}(p_{i_1}, v_{i_1j_1}, p_{i_2}, v_{i_2j_2}) = \sum_{x=1}^{k} S(p_{i_1}, v_{i_1j_1}, p_{i_2}, v_{i_2j_2}, t_x)$$

According to Definition 5, the similarity between variables can be used to evaluate the semantic similarity between programs. However, when the sequence of the variable $v_{i_1j_1}$ contains the sequence of the variable $v_{i_2j_2}$, although $S_{total}(p_{i_1}, v_{i_1j_1}, p_{i_2}, v_{i_2j_2})$ is very high at this time, their semantics are different. Therefore, to solve this problem, we further design the normalized similarity calculation formula shown in Definition 6.

Definition 6. For multiple test cases, the normalized similarity of two variables is the sum of LCS lengths under each test case divided by the sum of these sequences' lengths. Consider the test suite T, the normalized similarity of the variables $v_{i_1j_1}$ and $v_{i_2j_2}$ can be calculated as:

$$S_{normal}(p_{i_1}, v_{i_1j_1}, p_{i_2}, v_{i_2j_2}) = \frac{2 \times S_{total}(p_{i_1}, v_{i_1j_1}, p_{i_2}, v_{i_2j_2})}{Len(p_{i_1}, v_{i_1j_1}) + Len(p_{i_2}, v_{i_2j_2})}$$

where Len is the sum of the length of a variable's variable value sequence across all test cases, which can be calculated as:

$$Len(p_{i_1}, v_{i_1j_1}) = \sum_{x=1}^{k} sizeof(E(p_{i_1}, v_{i_1j_1}, t_x))$$

Finally, we obtain the similarity between programs by calculating the similarity of multiple variables (more details can be found in Section 4.2).

4.2. Matching similar program

When beginners debug the faults in the code, they may observe whether the value of the variable is consistent with the expected value. This way of inferring the possible position of the fault is a common manual debugging method. The essence of this method is similar to comparing the variable value sequence of the faulty program with the variable value sequence of the correct program, and the statements where the variable values are different are more likely to contain faults. However, the effectiveness of this way is based on the following two prerequisites:

- The two programs should have similar implementation logic.
- The two variables used for comparison should have similar functions.

Algorithm 1 Matching programs

Input: p_b: faulty program;
Input: P_c: correct program set;
Output: matched program set $Sim(p_b)$;

$V_b \leftarrow$ variables in p_b;
for p_{c_i} in P_c **do**
  $V_{c_i} \leftarrow$ variables in p_{ci};
  $G \leftarrow$ a graph with size of $|V_b| \times |V_{c_i}|$
  for $j_1 = 1$ to $|V_b|$ **do**
    for $j_2 = 1$ to $|V_{c_i}|$ **do**
      $G(j_1, j_2) \leftarrow S_{total}(p_b, v_{bj_1}, p_{c_i}, v_{c_ij_2})$
    end for
  end for
  $G'(p_b, p_{c_i}) \leftarrow$ maximum complete matching of G
  $l \leftarrow$ a list of length $|V_b|$
  for j_1 in range($|V_b|$) **do**
    $pair(v_{bj_1}, p_b, p_{c_i}) \leftarrow$ corresponding variable of v_{bj_1} in $G'(p_b, p_{c_i})$
    $l_{j_1} \leftarrow S_{normal}(p_b, v_{bj_1}, p_{c_i}, pair(v_{bj_1}, p_b, p_{c_i}))$
  end for
  similarity between p_b and $p_{c_i} \leftarrow \sum_{j_1=1}^{|V_b|} l_{j_1}$
end for
$Sim(p_b) \leftarrow argmax$(similarity between p_b and p_{c_i})
return $Sim(p_b)$

To ensure these two prerequisites, it is necessary to get the matching relationship of variables and programs, find logically similar programs and functionally similar variables to ensure the performance of the subsequent fault localization technique. However, when a novice submits a faulty program, it is unrealistic to match the program manually, because there are many correct programs with different implementations for each a problem in the OJ system. If we choose the program randomly, the similarity between the programs cannot be guaranteed, and then the following comparison will also be meaningless. Therefore, variable value sequence can be used to find the correct program which shares similar logic with the target faulty program, because the more similar the change process of the two program variables, the more similar the logic of the two programs will be.

Algorithm 1 describes the process of program matching. The algorithm's input consists of the target faulty program p_b and the correct sample program set P_c (which has been preprocessed to obtain the corresponding variable value sequence). This algorithm first applies static analysis on p_b to extract the variable set V_b contained in p_b, and then initialize the variable set of each program p_{c_i} in P_c to V_{c_i}.

Following that, we can construct a bipartite graph G with two node sets: one for the variable V_b in the faulty program and another for the variable V_{c_i} in the correct program, and their numbers equal to the number of variables in the two programs, represented by $|V_b|$ and $|V_{c_i}|$. The length $S_{total}(p_b, v_{bj_1}, p_{c_i}, v_{c_ij_2})$ of the longest similar subsequence can be calculated for each variable v_{bj_1} in V_b and each variable $v_{c_ij_2}$ in V_{c_i}, and corresponding to the bipartite graph, an edge of length $S_{total}(p_b, v_{bj_1}, p_{c_i}, v_{c_ij_2})$ is connected between the node v_{bj_1} and the node $v_{c_ij_2}$. After connecting all of the edges, we apply the Kuhn–Munkres algorithm (Kuhn, 1955) to determine G's maximum complete matching. In this matching relationship between p_b and p_{c_i} ($p_{c_i} \in P_c$), each variable in V_b can connect to a variable in V_{c_i} for each p_{c_i} in P_c, which will be recorded as $pair(v_{bj_1}, p_b, p_{c_i})$ for each variable in V_b.

Later this algorithm calculates the value of all $S_{normal}(p_b, v_{bj_1}, p_{c_i}, pair(v_{bj_1}, p_b, p_{c_i}))$ ($v_{bj_1} \in V_b$) in $G'(p_b, p_{c_i})$, then we can get the similarity between these two programs. Specifically, the number of variables in different programs is not always same, which may result in certain variables being mismatched. In this case, we set the similarity value of those variables to 0. Finally, the correct programs with the highest degree of similarity to p_b are added to the output set $Sim(p_b)$, and any correct program in $Sim(p_b)$ can be used in the following fault localization process.

Notice that the *LCS* of the failed test cases should also be calculated, because sometimes a simple fault (such as an incorrect start value of the loop) will cause the program to fail on all test cases, but at this time the two programs are still logically similar. When this situation happens, only calculating the *LCS* of passed test cases will not be able to match the correct program.

Then we introduce how to calculate the similarity between the faulty program p_1 and the correct program p_2 in Fig. 4. For example, for two programs p_1 and p_2, their contained variables are *a*, *b*, *n* and *x*, *y*, *m*. When the inputs given by the test cases are "5 3", "206 4", "388 6", and "666 5", these three variables will produce variable value sequences as shown in Table 2.

After getting the variable value sequence in the above table, we can calculate the longest common subsequence between each variable in p_1 and each variable in p_2. When the t_{target} = "206 4", we can get $S(p_1, a, p_2, x, t_{target}) = 2$, $S(p_1, b, p_2, y, t_{target}) = 1$ and $S(p_1, n, p_2, m, t_{target}) = 1$ through calculation, and remaining results are all zero. By calculating the data under the remaining test cases, we can get the final result: $S_{total}(p_1, a, p_2, x) = 8$, $S_{total}(p_1, b, p_2, y) = 4$, $S_{total}(p_1, n, p_2, m) = 6$, while the results among the remaining variables are all zero.

Then, as illustrated in Table 3, we can construct a bipartite graph. The rows *a*, *b*, and *n* in this matrix correspond to the variable value sequence in p_1, while the columns *x*, *y*, and *m* correspond to the variable value sequence in p_2. We connect an edge of length 8 between *a* and *x*, because $S_{total}(p_1, a, p_2, x)$ is 8. As seen in Table 3, the cell which in the row *a* and the column *x* is marked

Faulty Program p_1

```
1 #include<stdio.h>
2 int main() {
3     int a, b, n = 1;
4     scanf("%d %d", &a, &b);
5     if (b % 2 == 0) {
6         a = a + b;
7         n = a + n % 100;
          //(correct) n = n + a % 100;
8     } else {
9         a = a - b;
10        n = n + a / 100;
11    }
12    printf(" %d\n", n);
13 }
```

Correct Program p_2

```
#include<stdio.h>
int main() {
    int x, y, m = 1;
    scanf("%d %d", &x, &y);
    if (y % 2 == 0) {
        x = x + y;
        m = m + x % 100;

    } else {
        x = x - y;
        m = m + x / 100;
    }
    printf("%d\n", m);
}
```

Fig. 4. Sample of program.

Table 2
Variable sequences.

Program	Variables	Inputs			
		5 3	206 4	388 6	666 5
p_1	*a*	{5,2}	{206,210}	{388,394}	{666,661}
	b	{3}	{4}	{6}	{5}
	n	{1,1}	{1,211}	{1,395}	{1,7}
p_2	*x*	{5,2}	{206,210}	{388,394}	{666,661}
	y	{3}	{4}	{6}	{5}
	m	{1,1}	{1,11}	{1,95}	{1,7}

Table 3
LCS between variables and the mapping results.

	x	*y*	*m*	Matched variable
a	**8**	0	0	*x*
b	0	**4**	0	*y*
n	0	0	**6**	*n*

Algorithm 2 Calculating the suspiciousness for statements

Input: p_b: faulty program;
Input: P_c: correct program set;
Input: T: a set of tests;
Output: Sus_{p_b}: a set of statements' suspiciousness scores;
1: $V_b \leftarrow$ variables in p_b;
2: $Sim(p_b) \leftarrow$ match correct programs similar to p_b from P_c
3: $p_c \leftarrow$ select a program from $Sim(P_b)$ as a reference
4: $Sus_{p_b} \leftarrow$ initialize suspiciousness scores of statements in p_b to -1;
5: **for** $j_1 = 1$ to $|V_b|$ **do**
6: $v_{cj_2} \leftarrow$ corresponding variable of v_{bj_1} in p_c
7: **for** $t \in T$ **do**
8: $LSC(p_b, v_{bj_1}, p_c, v_{cj_2}, t) \leftarrow$ LSC between v_{bj_1} and v_{cj_2} under the test case t
9: **for** $st \in Sus_{p_b}$ **do**
10: $d \leftarrow$value of v_{bj_1} in the statement st
11: **if** $d \notin LSC(p_b, v_{bj_1}, p_c, v_{cj_2}, t)$ & t fails **then**
12: $Sus_{p_b}(st) = Sus_{p_b}(st) * \alpha$
13: **end if**
14: **if** $d \in LSC(p_b, v_{bj_1}, p_c, v_{cj_2}, t)$ & t passs **then**
15: $Sus_{p_b}(st) = Sus_{p_b}(st) * (2 - \alpha)$
16: **end if**
17: **end for**
18: **end for**
19: **end for**
20: **if** $\forall Sus_{p_b}(st) == 1$ **then**
21: **for** $st \in Sus_{p_b}$ **do**
22: **if** st is output statement **then**
23: $Sus_{p_b}(st) = Sus_{p_b}(st) * \alpha$
24: **end if**
25: **end for**
26: **end if**
27: **return** Sus_{p_b}

as '8'. After connecting all the edges, the matching result in the table can be obtained by bipartite graph matching: a matches x, b matches y, and m matches n, the weights of these three edges are 8, 4 and 6 respectively. Finally, after calculating their similarity through Definition 6, the results are $S_{normal}(p_1, a, p_2, x) = 1$, $S_{normal}(p_1, b, p_2, y) = 1$ and $S_{normal}(p_1, n, p_2, m) = 0.75$. Adding these three similarity values can achieve the final similarity between p_1 and p_2 (i.e., 1+1+0.75=2.75).

4.3. Variable-suspiciousness-based fault localization

If we directly show novices the correct programs that are matched by the faulty program, this method does little to improve novices' programming ability, because novices need to know the locations of the faulty statements in their programs, rather than simply being given the correct answer. Moreover, novices may plagiarize the provided correct programs, which contradicts the intended purpose of programming education.

As a result, it is essential to provide exact information to help fault localization. Algorithm 2 describes the details of how our method works. Given a faulty program p_b, a set of correct programs P_c, and a test suite T, VsusFL first gains a set of correct programs $Sim(p_b)$ similar to p_b according to the similarity calculated in Section 4.2 (Lines 1–2). Then a correct program p_c is selected from $Sim(p_b)$ as the reference to assess whether the variable values are correct (Line 3). In Section 4.2, we can get the matching relationship between variables. Suppose that for each variable v_{bj_1} in the faulty program p_b, we can get its corresponding variable v_{cj_2} in the correct program p_c (Line 6). $LCS(p_b, v_{bj_1}, p_c, v_{cj_2}, t)$ is the longest common subsequence between v_{bj_1} in the faulty program p_b and the variable v_{cj_2} in the correct program p_c under the test case t (Line 8). We conjecture that for the variable v_{bj_1}, the values that appear in $LCS(p_b, v_{bj_1}, p_c, v_{cj_2}, t)$ have a high probability of being correct. To further determine the location of the fault in the program, we compare the variable value sequence $E(p_b, v_{bj_1}, t)$ of each variable v_{bj_1} with $LCS(p_b, v_{bj_1}, p_c, v_{cj_2}, t)$ (Lines 9–18). Specially, the variable suspiciousness of all statements is initialized to -1 (Line 4). If the test case t failed in execution and the value of v_{bj_1} in the statement st does not exist in $LCS(p_b, v_{bj_1}, p_c, v_{cj_2}, t)$, st should have a higher probability of being faulty. We denote the influence weight of an incorrect variable value on the suspiciousness score of a statement using the coefficient α, which ranges from 0 to 1. The degree of influence of the variable value on the statement's suspiciousness increases as α approaches 0, and decreases as α approaches 1. Similarly, 2-α represents the influence weight of a correct variable value on the suspiciousness score of a statement. Hence, the statement st is multiplied by α (the α is set to 0.8 in our study) to increase the suspiciousness (Lines 14–16). On the contrary, if the test case t is executed successfully and the value of v_{bj_1} in the statement st does exist in $LCS(p_b, v_{bj_1}, p_c, v_{cj_2}, t)$, the statement st is multiplied by the coefficient (2-α) to decrease the suspiciousness (Lines 14–16). It is worth noting that the mapping between variable values and statements is automatically implemented by the tool 'CppSnooper' we provide. After comparing all the variable value sequences under all test cases, if the variable suspiciousness of all statements has not changed, we multiply the output statements by α, because at this time, we conjecture that there is no fault in the calculation process, and the fault is more likely in the output statement (Lines 20–26).

Still taking the faulty program in Fig. 4 as the example, Table 2 shows the generated variable value sequences, and the matching relationship between the variables is shown in Table 3. Each statement is initialized with a suspiciousness of -1. When the test case is t_{target}="206 4", we can get the longest common subsequence between the three variable pairs $LCS(p_1, a, p_2, x, t_{target}) = \{206, 210\}$, $LCS(p_1, b, p_2, y, t_{target}) = \{4\}$, $LCS(p_1, n, p_2, m, t_{target}) = \{1\}$ according to Definition 4. Comparing the variable value sequence in the faulty program with the longest common subsequence, we can find that the value sequences of the target a and b is exactly the same as $LCS(p_1, a, p_2, x, t_{target})$, $LCS(p_1, b, p_2, y, t_{target})$. However, in the value sequence of the variable n, "211" does not exist in $LCS(p_1, n, p_2, m, t_{target})$, and the test case fails, so we multiply the variable suspiciousness by 0.8 in line 7, as it produced this value. In the same way, for the test case, t_{target}="388 6", the variable suspiciousness needs to be multiplied by 0.8 again in line 7. On the contrary, in the passed test cases are t_{target}="5 3" and t_{target}="666 5", since the variable values are all in the longest common subsequence, all the variable suspiciousness of statements that produce these variable values need to be multiplied by 1.2. Finally, we would get the result as shown in Table 4.

In Table 4, the grids marked with "✓" indicates that the test case passed and the variable value generated by this statement is in the longest common subsequence, while the grid marked with "×" indicates that the test case failed and the variable value generated by the statement does not exist in the longest common subsequence. Finally, we can obtain the final rank list in the final column by ranking these suspiciousness scores from large to small. The actual faulty statement (i.e., the seventh line) is ranked first in this example, and the faulty statement can be successfully localized.

Table 4
Variable suspiciousness and the final rank.

Line	Inputs												Suspiciousness value	Final rank
	5 3			206 4			388 6			666 5				
	a	*b*	*n*	*a*	*b*	*n*	*a*	*b*	*n*	*a*	*b*	*n*		
1														
2														
3			✓									✓	−1.44	7
4	✓	✓								✓	✓		−2.0736	8
5													−1	4
6													−1	4
7						×			×				−0.64	1
8														
9	✓									✓			−1.44	7
10			✓									✓	−1.44	7
11														
12													−1	4
13														

5. Experiment setup

5.1. Research questions

To evaluate the effectiveness of our proposed method VsusFL, we aim to investigate the following three research questions:

RQ1: How effective is our proposed method VsusFL in fault localization for novice programs?

We design this RQ to verify VsusFL's ability to localize faults in novice programs. To answer this RQ, we conducted a series of experimental studies and compared VsusFL with other state-of-the-art FLs, including Grace (Lou et al., 2021), ANGELINA (Mechtaev et al., 2016; Chandra et al., 2011), VSBFL (Li et al., 2021a), SBFL (Abreu et al., 2007), and VFL (Kim et al., 2019). Notice that ANGELINA does not support C++ language (Chandra et al., 2011), which means that the experiments for ANGELINA are limited to C language programs. Besides, MBFL (Moon et al., 2014; Papadakis and Le Traon, 2015) and ProFL (Lou et al., 2020b) are not selected since they are too time-consuming to generate timely feedback for beginners. We also ignore bug report-based fault localization methods (e.g., BugLocator (Zhou et al., 2012), BRTracer (Wong et al., 2014), and DeepFL (Li et al., 2019a)) because the bug reports required by these methods are not available for novice programs.

RQ2: How about the performance of VsusFL on different categories of novice programs?

This RQ attempts to explore the performance of VsusFL on different categories of novice programs, so as to analyze the applicability of VsusFL and provide research directions for further performance improvement of VsusFL. To answer this RQ, we divide our used experimental subjects into three categories (i.e., Simple Expression, Conditionals, and Loops), and then compare the performance of VsusFL and other baselines on different categories of experimental subjects in terms of fault localization performance.

RQ3: How effectively can we combine VsusFL with VSBFL?

In this RQ3, we aim to investigate whether VsusFL can further boost other techniques that use various information. To do this, we first explores the possibilities of combining VsusFL and other techniques by analyzing their correlation. Then we use three methods for combining VsusFL and VSBFL that can effectively improve the fault localization performance of VsusFL and VSBFL.

5.2. Methods for combining VsusFL and SBFL

In this study, we utilize three methods for combining VsusFL and VSBFL. Specifically, For each statement in the faulty program, we can get its variable suspiciousness through VsusFL and get its statement suspiciousness through VSBFL. Then we can combine these two to obtain the final suspiciousness of statements through the combination method. Since the range of suspiciousness of each technique is quite different, all suspiciousness should be normalized to be between 0 and 1 according to Eq. (1).

$$Sus(st_i) = \frac{Sus(st_i) - \mu}{\sigma} \tag{1}$$

$$\sigma = \sqrt{\frac{\sum_{st_j \in p_b}(Sus(st_j) - \mu)^2}{\mu}} \tag{2}$$

where μ is the average of the suspiciousness scores of all statements in faulty program p_b computed by a FL technique.

For a statement st of a faulty program, sus_1 is the statement st's the variable suspiciousness calculated by VsusFL, sus_2 is st's suspiciousness calculated by VSBFL. The $sus(st)$ denotes the final suspiciousness score achieved by utilizing the ensemble approach. Our three methods considered are linear combination, multiplication combination, and polynomial regression. Formula (3) shows the linear combination calculation process. The value of the coefficient β ranges from 0 to 1 with an interval of 0.1.

$$sus(st) = \beta \times sus_1 + (1 - \beta) \times sus_2 \tag{3}$$

Formula (4) shows how the final suspiciousness score of the statement st can be obtained by multiplication combination integrating the suspiciousness scores calculated by two FL approaches. The reason for adding 1 in front of each suspiciousness is to prevent the suspiciousness from being 0, which can cause another suspiciousness to lose effectiveness.

$$sus(st) = (1 + sus_1) \times (1 + sus_2) \tag{4}$$

Polynomial regression is the machine learning algorithm. When there are multiple independent variables, multiple regression analysis can fit a curve as shown in Formula (5) by specifying the highest term r in the polynomial, where the independent variable st represents each statement in a faulty program, and $st = \langle sus_1, sus_2 \rangle$.

$$sus(st) = a_0 + a_1 \times st + \cdots + a_r \times st^r \tag{5}$$

The advantage of polynomial regression is that the curve can be approximated by increasing the r until it is satisfied. In fact, polynomial regression can handle quite a class of non-linear problems, because any function can be approximated by a polynomial. In order to evaluate the performance of polynomial regression accurately, we use 5-fold cross validation in our experiment.

5.3. Subject programs

In this study, we utilize the dataset shared by Yi et al. (2017) as subject programs for our experiment. This dataset has gained

Table 5
Problems in the dataset.

Types	Language	Problems	Programs	Faults	Size
Simple Expressions	C	2810, 2811, 2812, 2813	56	91	9-24
Conditionals	C	2824, 2825, 2827, 2828, 2830, 2831, 2832, 2833	104	239	9-124
	C++	1642, 2122, 2208	50	56	27-211
Loops or Nested Loops	C	2864, 2865, 2866, 2867, 2868, 2869, 2870, 2871	63	119	11-64
	C++	1335, 1500, 1687, 1912, 1927, 1947, 1951, 2111, 2112, 2327	149	159	13-57

widespread recognition and has been employed in numerous prior research studies (Gulwani et al., 2018; Li et al., 2019a; Hua et al., 2018). The programs in this dataset were written by novice programmers enrolled in an introductory C language course at the Indian Institute of Technology Kanpur (IIT-K).

Furthermore, we augment our dataset by incorporating novice C++ programs sourced from a student programming platform,[8] which is actively used in the real-world educational environment.

Lastly, our subject programs are collected from a pool of 33 problems. After excluding 76 programs that either exceeded the time limit, reported runtime errors, or failed to compile using GCC, we finally consider a total of 422 programs, containing 664 faults.

Table 5 shows detailed information on our subject programs. These problems can be classified into three categories based on their annotations: (1) Simple Expressions, (2) Conditions, and (3) Loops or Nested Loops. The corresponding category has 56, 154, and 162 programs, respectively. We also show the programming language they use and their program sizes (i.e., LOC), which range from 9 to more than 211 lines, in this table. We chose this dataset as the experimental subjects because the faults contained in these programs were gathered from the real world, rather than artificially generated (such as SIR (Do et al., 2005)), and this dataset contains a large number of programs that belong to different categories.

5.4. Evaluation metrics

5.4.1. EXAM score

EXAM Score (*EXAM*) is defined as the percentage of statements that have to be checked until the first faulty statement is localized. The *EXAM* is usually used to evaluate the accuracy of fault localization techniques, while a lower *EXAM* indicates that fewer statements need to be checked to find the faulty statement. Then the corresponding fault localization technique has better performance. Specifically, *EXAM* can be determined as the following formula:

$$EXAM = \frac{rank\ of\ the\ faulty\ statement}{number\ of\ the\ executable\ statements} \tag{6}$$

Simply put, *EXAM* is equal to the proportion of the rank of the faulty statement in the number of executable statements. Thus, the efforts of a developer in localizing a faulty statement can be quantified for each faulty program.

However, several statements may have the same suspiciousness value, which can result in the tie issue. We present their final ranks using the worst rank. This strategy has been widely utilized by previous fault localization algorithms to deal with the tie issue (Lou et al., 2021; Li et al., 2019b; Li and Zhang, 2017; Benton et al., 2020). Using the example of a faulty statement s, A indicates the number of statements in the program that have a higher suspiciousness than s, while B represents the number of statements that have the same suspiciousness as s. The final rank of the faulty statement s can be determined by calculating their sum $A + B$.

Moreover, the equation will be affected for multi-fault programs. The original *EXAM* is then extended to $A-EXAM$ (Average *EXAM*), as indicated in Eq. (7) (Wu et al., 2020). Eq. (7) can be used to calculate the effort of a developer in localizing all faults in multi-fault programs.

$$A{-}EXAM = \frac{\Sigma_{n=1}^{N} EXAM(n)}{N} \tag{7}$$

where $EXAM(n)$ is the *EXAM* of the nth faulty statement in the ranking list. N is the total number of faulty statements. A smaller $A-EXAM$ value indicates a more effective technique for fault localization.

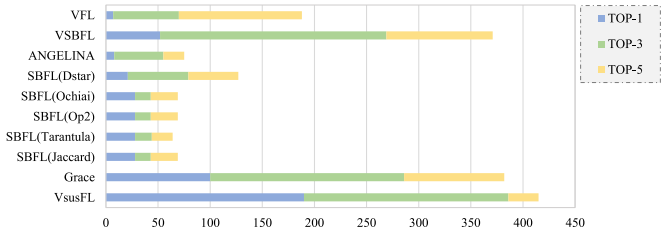


Fig. 5. Fault localization performance comparison in terms of $TOP-N$ Metric.

5.4.2. TOP − N

$TOP-N$ is the number of programs for which an algorithm ranks all faulty statements in the top-N positions of the ranked list. The higher the value of $TOP-N$, the less effort it takes for developers to localize faulty statements, and the corresponding fault localization technique will perform better. According to the previous research (Parnin and Orso, 2011), developers should only review the first few statements in the ranking list, as shown by the metric $TOP-N$. Note that this metric is crucial in practice since developers usually only inspect top-ranked statements. For example, a recent study found over 70% of developers only check $TOP-5$ ranked statements in practical software debugging (Kochhar et al., 2016). Therefore, in line with previous studies (Lou et al., 2021; Li et al., 2019b; Li and Zhang, 2017; Benton et al., 2020), we are primarily focused on $TOP-1$, $TOP-3$, and $TOP-5$.

6. Results analysis

6.1. RQ1. Effectiveness of VsusFL

In order to evaluate the performance of VsusFL, we consider Grace, SBFL, ANGELINA, VFL, and VSBFL as the baselines, and separately count their experimental results. The results are shown in Table 6. To make the results more intuitive, Fig. 5 further shows the performance of the above techniques in terms of the metric $TOP-N$. In practice, developers often only pay attention to a few statements which are listed on the top of the suspiciousness

[8] https://buctcoder.com

Table 6
Performance of each technique.

Techniques	$A-EXAM$	TOP-1	TOP-3	TOP-5
VsusFL	0.309	**190**	**386**	**415**
Grace	**0.234**	100	286	382
SBFL(Jaccard)	0.440	28	43	69
SBFL(Tarantula)	0.472	28	44	64
SBFL(Op2)	0.438	28	43	69
SBFL(Ochiai)	0.440	28	43	69
SBFL(Dstar)	0.766	21	79	127
ANGELINA	0.702	8	55	75
VSBFL	0.250	52	269	371
VFL	0.439	6	57	178

list (Parnin and Orso, 2011), so the metric $TOP-N$ can better represent the effectiveness of fault localization techniques.

As can be found in Fig. 5, VsusFL can significantly outperform all comparison baselines in $TOP-1$, $TOP-3$, and $TOP-5$. Compared with the two best-performing techniques, Grace and VSBFL, in terms of each metric, the performance is improved by 90%, 35%, 9%, and 265%, 43%, 12%, respectively, and this improvement can be even higher when compared with the SBFL, VFL, ANGELINA.

Furthermore, we have the following findings from Table 6. (1) Value-based FL techniques (e.g., VFL, VSBFL, VsusFL) surpass traditional SBFL formulae in almost all metrics. (2) Learning-based FL technique Grace can outperform VSBFL, VFL, ANGELINA, and traditional SBFL On the observed metrics. (3) VsusFL outperforms all comparison baselines in $TOP-N$, but is weaker than Grace and VSBFL in $A-EXAM$.

We analyze in detail the reasons for these results, and find that VSBFL can combine variable suspiciousness with statement suspiciousness in a sense, while Grace utilizes information about code structure, statement type, and coverage. By leveraging multiple sources of information, them can local more faults than VsusFL in $TOP-10$. In particular, VSBFL and Grace can respectively localize 477 and 506 faults in $TOP-10$, which is 57 and 84 more than what VsusFL can localize. On the other hand, VsusFL outperforms Grace and VSBFL by a large margin in $TOP-1$, $TOP-3$, and $TOP-5$, but not good in $A-EXAM$ and $TOP-10$, which seems to be a pair of contradictory data. This reason is that VsusFL can use fine-grained information provided by variable values to achieve good performance for some specific types of novice programs. Therefore, VsusFL can rank a large number of faulty statements at the top of suspiciousness list but also rank some faulty statements at the end.

These results show that VsusFL is different from the comparison baselines because VsusFL has a good distinction between program statements and will not be troubled by the tie issue. Overall, VsusFL stands out as the best among all methods in localizing faults for novice programs.

6.2. RQ2. Effectiveness of vsusfl on different categories of novice programs

The programs in the dataset can be divided into three categories: Simple Expression (SE), Conditionals (CD), and Loops. For different categories of programs, the performance of fault localization techniques may be different.

Table 7 shows the performance of different fault localization techniques in different categories of programs. To simplify the table, we only selected Jaccard, which performed best to represent SBFL. As can be seen from the table, VsusFL outperforms VSBFL, Grace, ANGELINA, SBFL and VFL in almost all categories under $TOP-1$, $TOP-3$ and $TOP-5$. In particular, VsusFL performs similarly to Grace in SE but outperforms the other baselines. In

Table 7
Performance of techniques for different categories.

Categories	Techniques	$A-EXAM$	TOP-1	TOP-3	TOP-5
SE	VsusFL	0.314	17	67	74
	Grace	0.162	19	58	82
	Jaccard	0.350	7	7	18
	ANGELINA	0.714	1	15	17
	VSBFL	0.213	10	64	76
	VFL	0.392	1	8	48
CD	VsusFL	0.371	48	132	146
	Grace	0.220	39	114	149
	Jaccard	0.368	19	33	45
	ANGELINA	0.698	1	26	38
	VSBFL	0.269	25	98	120
	VFL	0.508	5	28	53
Loops	VsusFL	0.270	125	187	195
	Grace	0.258	42	114	151
	Jaccard	0.503	2	3	6
	ANGELINA	0.694	6	14	20
	VSBFL	0.246	17	107	175
	VFL	0.345	0	21	77

CD, VsusFL is slightly better than Grace and considerably better than the other baselines. For Loops, VsusFL is far superior to both Grace and the other baselines. We believe that the effectiveness of VsusFL increases as the richness and amount of information provided by variable values in novice programs increases.

Another finding is that while VSBFL performs slightly worse than Grace in $TOP-1$ and $TOP-3$, it performed equally as well as Grace in $TOP-5$. In addition, our analysis in Section 1 and 3 is supported by the noteworthy observation that SBFL shows better performance on CD code as compared to that SE and Loops. This is attributable to the fact that CD programs tend to have more branching structures, which allows test cases to provide more identifiable information about statements, facilitating the higher efficacy of SBFL. In summary, these results are consistent with the result in RQ1.

6.3. RQ3. Performance of the combined technique

This RQ3 aims to explore the possibility of combining our proposed VsusFL with other fault localization techniques. Two techniques are correlated if they are good at identifying the same types of faults. Conversely, if two techniques display a low correlation, they may offer different information, and combining them has the potential to yield superior results compared to either individual technique.

Table 8 shows the p-value and effect size (d-value) between VsusFL and the other techniques, where p-values are calculated using the Wilson two-sided test based on the $A-EXAM$ values provided by each technique within the 422 programs, and the effect size is measured using the Cohen's d (Sullivan and Feinn, 2012; Grissom and Kim, 2005). The effect size value is considered small for $0.147 < d-value < 0.33$, medium for $0.33 < d-value < 0.474$, and large for $0.474 < d$-value, indicating the extent of the observed effect (Grissom and Kim, 2005).

In Table 8, there are eight pairs of different techniques. Among them, only three show statistically significant correlation as their p-values exceed the significance level $\alpha = 0.05$. The remaining pairs of techniques have p-values less than 0.05, indicating no statistically significant correlation between them. Additionally, the pairs of VsusFL and VSBFL, as well as VsusFL and SBFL (Jaccard, Op2, Ochiai), exhibit large d-values among the five pairs, meaning a statistically significant difference with a substantial effect size. In particular, the effect size between VsusFL and VSBFL is larger

Table 8
P-Value and Cohen's D (*d*-value) for correlation analysis between each FL technique and VsusFL, where P-values<0.05 and large D-values are highlighted.

	Correlation	Grace	SBFL(Jaccard)	SBFL(Tarantula)	SBFL(Op2)	SBFL(Ochiai)	SBFL(Dstar)	VFL	VSBFL
VsusFL	*p*-value	0.0699	**0.0149**	**0.0016**	**0.0149**	**0.0149**	0.8250	0.6634	**1.6544^{-6}**
	d-value	0.4449	**0.5018**	0.4151	**0.5018**	**0.5015**	0.0129	**0.7878**	**0.7502**

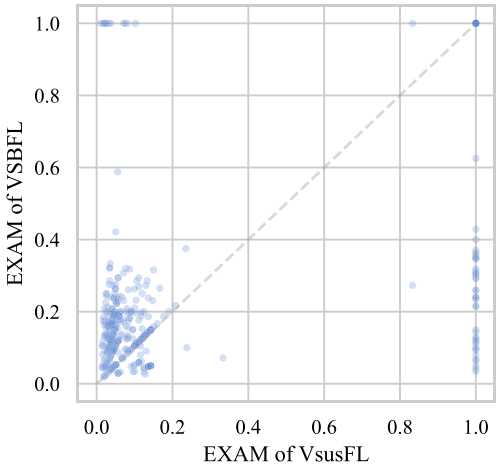


Fig. 6. The correlation of the pair of VsusFL and VSBFL. The X and Y values for A point show the $A-EXAM$ values for two techniques on the same faulty program. $A-EXAM$ Is the effort that a user in localizing all faulty statements.

than that of the other pairs, suggesting a more significant difference between VsusFL and VSBFL. Besides, it is worth noting that VSBFL outperforms SBFL in fault localization, as demonstrated in Table 6.

To provide a more intuitive analysis of the correlation between VsusuFL and VSBFL, we further present the results via the scatter plot of the pair of VsusFL and VSBFL, as shown in Fig. 6. The plot has 422 points, one for each faulty program in our dataset. The coordinate (x, y) for a faulty program means on this program, the $A-EXAM$ for VsusFL is x, and the $A-EXAM$ for VSBFL is y.

In Fig. 6, there are many dots located in the upper-left and lower-right regions. This pattern indicates that there are many faulty programs where one technique performs well, but the other does not. These programs suggest that the two techniques are not positively correlated, and each has the potential to provide unique information that the other cannot.

We further calculate the correlation coefficient between VSBFL and VsusFL, i.e., 0.2308. A correlation coefficient that is almost 0 often indicates an almost negligible correlation. On the other hand, a correlation coefficient that is near to -1 or 1 denotes a considerable correlation. The value of 0.2308, being close to zero, implies that there exists a weak linear correlation between VSBFL and VsusFL.

The results show that they may offer distinct information to one another. This also provides theoretical support for the subsequent combination of these two techniques to achieve higher fault localization accuracy.

Next, we combine VsusFL and VSBFL using each of the three combined methods introduced in Section 5.2, and then evaluate the performance of the combined fault localization technique at different coefficients.

Table 9 shows the effect of the linear combination method under different values of β, where the best-performing results are bolded. It can be seen from the table that after combining VsusFL with VSBFL, the effect of both has been significantly improved. When using $A-EXAM$ as the metric, the best performance can be obtained when β is 0.1. It is 17.2% higher than VSBFL and

Table 9
Effects of linear combination.

β	$A-EXAM$	TOP-1	TOP-3	TOP-5	TOP-10
0	0.250	52	269	371	477
0.1	**0.207**	213	392	438	**482**
0.2	0.211	212	393	438	481
0.3	0.211	213	393	438	480
0.4	0.209	217	397	440	481
0.5	0.209	218	**401**	440	480
0.6	0.210	218	399	**441**	480
0.7	0.210	218	399	**441**	480
0.8	0.219	**219**	394	430	472
0.9	0.220	217	390	430	472
1	0.309	190	386	415	420

Table 10
Effects of polynomial regression.

r	$A-EXAM$	TOP-1	TOP-3	TOP-5	TOP-10
1	1	7	7	10	0.989
2	1	7	7	10	0.989
3	1	7	7	10	0.989
4	1	7	7	10	0.989
5	1	7	7	10	0.989
6	1	7	7	10	0.989
7	1	7	7	10	0.989
8	1	7	7	10	0.989
9	1	7	7	10	0.989
10	1	7	7	10	0.989

33% higher than VsusFL. Meanwhile, when using $TOP-N$ as an evaluation metric, β is best at 0.5, and this improvement is quite obvious in $TOP-1$, $TOP-3$, $TOP-5$, and $TOP-10$, with 319%, 49%, 19%, 1% and 15%, 4%, 6%, 14% improvements compared to VSBFL and VsusFL, respectively. Compared with the previous best-performing technique Grace, the performance after using the linear combination method can also be improved, which improved by 118% in $TOP-1$ and 11.5% in $A-EXAM$.

Similarly, when using the multiplicative combination, we can get similar results: $A-EXAM$ is 0.304, $TOP-1$ is 220, $TOP-3$ is 389, $TOP-5$ is 415, and $TOP-10$ is 420. Like the linear combination, the performance of fault localization has also been effectively improved in $TOP-1$ and $TOP-3$. These results also strongly support that the VsusFL and VSBFL can utilize the information from each other to improve the effectiveness of fault localization on novice programs.

In contrast to the previous two combined methods, polynomial regression exhibits considerably worse performance, as shown in Table 10. Moreover, it can be seen from Table 10 that the variation in parameters has little effect on its performance. After conducting a thorough analysis, we have found that the polynomial regression method results in a severe tie problem. This means that nearly all statements in the program are assigned the same suspiciousness score, which greatly limits the effectiveness of the combined method.

In summary, experimental results show that the combination methods can effectively improve the fault localization performance of VsusFL and outperform those fault localization techniques when used alone. Besides, the linear combination approach is considered the most suitable combined method for integrating VsusFL with other fault localization techniques, and it is suggested to set the β-value at 0.5.

Table 11
Performance of the combined method (V+V), which combines VsusFL and VSBFL using the linear combination with a β-value of 0.5, on different categories.

Categories	Technique	*A − EXAM*	*TOP*-1	*TOP*-3	*TOP*-5	*TOP*-10
SE	V+V	**0.197**	**24**	**67**	75	**86**
	VsusFL	0.314	17	67	74	76
	VSBFL	0.213	10	64	**76**	85
CD	V+V	**0.225**	**68**	**145**	**168**	**186**
	VsusFL	0.371	48	132	146	146
	VSBFL	0.269	25	98	120	172
Loops	V+V	**0.203**	**126**	**189**	**197**	**208**
	VsusFL	0.269	125	187	195	198
	VSBFL	0.246	17	107	175	220

We further investigate the effectiveness of the optimal combination approach, i.e., integrating VsusFL and VSBFL using the linear combination with a β-value of 0.5, on different categories of programs. The results are shown in Table 11, where results that outperform both VsusFL and VSBFL are bolded. It is evident that regardless of program categories or observation metrics, the combined method consistently outperforms VsusFL and VSBFL, even Grace. Moreover, the improvement is more noticeable in the CD and SE than in Loops. In particular, this finding also indicates that integrating VsusFL with other techniques can further enable more powerful fault localization.

7. Threats to validity

Threats to internal validity

The threats to internal validity are related to potential implementation errors in our proposed VsusFL and baseline techniques. To alleviate this threat, we firstly employ an open-source python package "munkres" to implement the bipartite graph-based program matching algorithm. Then we conduct VsusFL in several simple programs to verify the implementation correctness. Moreover, we directly used the open source code provided by prior work (Lou et al., 2021; Mechtaev et al., 2016; Chandra et al., 2011; Li et al., 2021a). Finally, we reimplement SBFL and VFL. To mitigate the possible error in our code, we implement these baselines according to their original description and carefully tested these baselines.

Besides, the choice of parameters poses an internal threat to our study that may impact the empirical results. The performance of our method may be affected by configuration of the coefficient α. To mitigate this particular threat, we experiment with several configurations, ultimately finding that the recommended setting is 0.8.

Threats to external validity

The threats to external validity are related to the dataset we used in this paper. It is essential to consider that passing all test cases might not ensure error-free code, as some faulty statements may remain untriggered. To minimize this issue, we manually examined the correctness of the programs identified as accurate within our dataset. Therefore, this threat is limited.

In addition, another threat comes from the labeled location of faulty statements. For example, as an important source of our dataset, Yi et al. only submitted the faulty programs and corresponding correct programs, but did not label the location of faulty statements. Therefore, we manually label the faulty statements for each faulty program. To avoid possible labeling errors, we invite several volunteers (including the authors of this study) to manually check the correctness of the labeled faulty statements.

Threats to construct validity

The threats to construct validity are related to the measurement of our experiments. To alleviate this threat, we adopt *EXAM* and *TOP − N* metrics to evaluate the performance of fault localization, since these two metrics have been widely used in previous fault localization studies (Lou et al., 2021; Li et al., 2021a).

8. Conclusion

In this study, we present VsusFL, which is a novel fault localization approach for automatically localizing faulty statements in novice programs. Inspired by stimulating the process of program debugging, VsusFL employs the variable value sequences to localize faulty statements in novice programs. To evaluate the effectiveness of VsusFL, empirical studies on 422 real faulty novice programs are conducted. The experimental results show that VsusFL performs better than Grace, VSBFL, ANGELINA, SBFL, and VFL. VsusFL's fault localization performance can outperform Grace and VSBFL, respectively, by 90%/265%, 35%/43%, and 9%/12% in terms of the *TOP* − 1, *TOP* − 3, and *TOP* − 5 metrics. Of more interest is the fact that VsusFL is weakly correlated with other FL techniques during the experiments, especially VSBFL. Further experiments show that when combining VsusFL with other techniques, the performance of the fault localization can be further improved significantly.

CRediT authorship contribution statement

Zheng Li: Funding acquisition, Resources, Writing – review & editing, Supervision. **Shumei Wu:** Methodology, Software, Validation, Formal analysis, Data curation, Writing – original draft. **Yong Liu:** Funding acquisition, Writing – review & editing, Supervision. **Jitao Shen:** Software, Writing – original draft. **Yonghao Wu:** Visualization, Writing – original draft. **Zhanwen Zhang:** Software, Data curation. **Xiang Chen:** Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: National Natural Science Foundation of China reports financial support was provided by National Natural Science Foundation of China.

Data availability

We have shared a Github link of our code in our paper.

Acknowledgments

The work described in this paper is supported by the National Natural Science Foundation of China (Grant nos. 61902015 and 61872026), the Nantong Application Research Plan (Grant no. JC2021124).

References

Abreu, R., Zoeteweij, P., c. Van Gemund, A.J., 2006. An evaluation of similarity coefficients for software fault localization. In: 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06). IEEE, pp. 39–46.

Abreu, R., Zoeteweij, P., Van Gemund, A.J., 2007. On the accuracy of spectrum-based fault localization. In: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007). IEEE, pp. 89–98.

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

Araujo, E., Gaudencio, M., Serey, D., Figueiredo, J., 2016. Applying spectrum-based fault localization on novice's programs. In: 2016 IEEE Frontiers in Education Conference. FIE, IEEE, pp. 1–8.

Benton, S., Li, X., Lou, Y., Zhang, L., 2020. On the effectiveness of unified debugging: An extensive study on 16 program repair systems. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. pp. 907–918.

Chandra, S., Torlak, E., Barman, S., Bodik, R., 2011. Angelic debugging. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 121–130.

Corbett, A.T., Anderson, J.R., 2001. Locus of feedback control in computer-based tutoring: Impact on learning rate, achievement and attitudes. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, pp. 245–252.

Do, H., Elbaum, S.G., Rothermel, G., 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empir. Softw. Eng. 10 (4), 405–435.

Endres, M., Sakkas, G., Cosman, B., Jhala, R., Weimer, W., 2019. Infix: Automatically repairing novice program inputs. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 399–410.

Feyzi, F., Parsa, S., 2018. A program slicing-based method for effective detection of coincidentally correct test cases. Computing 100 (9), 927–969.

Gao, R., Wong, W.E., 2017. MSeer—An advanced technique for locating multiple bugs in parallel. IEEE Trans. Softw. Eng. 45 (3), 301–318.

Grissom, R.J., Kim, J.J., 2005. Effect Sizes for Research: A Broad Practical Approach.. Lawrence Erlbaum Associates Publishers.

Gulwani, S., Radiček, I., Zuleger, F., 2018. Automated clustering and program repair for introductory programming assignments. ACM SIGPLAN Notices 53 (4), 465–480.

Guo, P.J., 2013. Online python tutor: embeddable web-based program visualization for cs education. In: Proceeding of the 44th ACM Technical Symposium on Computer Science Education. ACM, pp. 579–584.

Guo, P.J., 2015. Codeopticon: Real-time, one-to-many human tutoring for computer programming. In: Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology. ACM, pp. 599–608.

Gupta, R., Kanade, A., Shevade, S., 2019. Neural attribution for semantic bug-localization in student programs. Adv. Neural Inf. Process. Syst. 32.

Howden, W.E., 1978. Theoretical and empirical studies of program testing. IEEE Trans. Softw. Eng. (4), 293–298.

Hua, J., Zhang, M., Wang, K., Khurshid, S., 2018. Towards practical program repair with on-demand candidate generation. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018. ACM, pp. 12–23.

Jaccard, P., 1901. Etude de la distribution florale dans une portion des alpes et du jura. Bullet. Soc. Vaudoise Sci. Nat. 37, 547–579.

Jiang, J., Wang, R., Xiong, Y., Chen, X., Zhang, L., 2019. Combining spectrum-based fault localization and statistical debugging: An empirical study. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 502–514.

Johnson, W.L., 1990. Understanding and debugging novice programs. Artif. Intell. 42 (1), 51–97.

Jones, J.A., Harrold, M.J., Stasko, J., 2002. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering. ICSE 2002. ACM, pp. 467–477.

Ju, X., Jiang, S., Chen, X., Wang, X., Zhang, Y., Cao, H., 2014. HSFal: Effective fault localization using hybrid spectrum of full slices and execution slices. J. Syst. Softw. 90, 3–17.

Kim, J., An, G., Feldt, R., Yoo, S., 2021. Ahead of time mutation based fault localisation using statistical inference. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 253–263.

Kim, J., Kim, J., Lee, E., 2019. VFL: Variable-based fault localization. Inf. Softw. Technol. 107, 179–191.

Kim, C., Yuan, J., Vasconcelos, L., Shin, M., Hill, R.B., 2018. Debugging during block-based programming. Instruct. Sci. 46 (5), 767–787.

Kim, S., Zimmermann, T., Whitehead Jr., E.J., Zeller, A., 2007. Predicting faults from cached history. In: 29th International Conference on Software Engineering. ICSE'07, IEEE, pp. 489–498.

Kochhar, P.S., Xia, X., Lo, D., Li, S., 2016. Practitioners' expectations on automated fault localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, pp. 165–176.

Kuhn, H.W., 1955. The hungarian method for the assignment problem. Naval Res Logist Quart 2 (1–2), 83–97.

Lahtinen, E., Ala-Mutka, K., Järvinen, H.-M., 2005. A study of the difficulties of novice programmers. ACM Sigcse Bullet. 37 (3), 14–18.

Li, X., Li, W., Zhang, Y., Zhang, L., 2019a. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp. 169–180.

Li, X., Li, W., Zhang, Y., Zhang, L., 2019b. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In: Zhang, D., ller, A.M. (Eds.), Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019. ACM, pp. 169–180.

Li, Z., Shen, J., Wu, Y., Liu, Y., Sun, Z., 2021a. VSBFL: Variable value sequence based fault localization for novice programs. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C). IEEE, pp. 494–505.

Li, Y., Wang, S., Nguyen, T., 2021b. Fault localization with code coverage representation learning. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering. ICSE, pp. 661–673.

Li, Z., Wu, Y., Wang, H., Chen, X., Liu, Y., 2021c. Review of software multiple fault localization approaches. Chinese J. Comput. 1–33.

Li, X., Zhang, L., 2017. Transforming programs and tests in tandem for fault localization. Proc. ACM Program. Lang. 1 (OOPSLA), 1–30.

Li, Z., Zhou, X., Yu, D., Wu, Y., Liu, Y., Chen, X., 2021d. Fault localization-guided test data generation approach for novice programs. In: 8th International Conference on Dependable Systems and their Applications, DSA 2021, Yinchuan, China, August 5-6, 2021. IEEE, pp. 54–65.

Lou, Y., Ghanbari, A., Li, X., Zhang, L., Zhang, H., Hao, D., Zhang, L., 2020a. Can automated program repair refine fault localization? a unified debugging approach. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp. 75–87.

Lou, Y., Ghanbari, A., Li, X., Zhang, L., Zhang, H., Hao, D., Zhang, L., 2020b. Can automated program repair refine fault localization? a unified debugging approach. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp. 75–87.

Lou, Y., Zhu, Q., Dong, J., Li, X., Sun, Z., Hao, D., Zhang, L., Zhang, L., 2021. Boosting coverage-based fault localization via graph-based representation learning. In: Spinellis, D., Gousios, G., Chechik, M., Penta, M.D. (Eds.), ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021. ACM, pp. 664–676.

Mechtaev, S., Yi, J., Roychoudhury, A., 2016. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). pp. 691–701.

Moon, S., Kim, Y., Kim, M., Yoo, S., 2014. Ask the mutants: Mutating faulty programs for fault localization. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. IEEE, pp. 153–162.

Mow, I.C., 2008. Issues and difficulties in teaching novice computer programming. In: Innovative Techniques in Instruction Technology, E-Learning, E-Assessment, and Education. Springer Netherlands, pp. 199–204.

Naish, L., Lee, H.J., Ramamohanarao, K., 2011. A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. (TOSEM) 20 (3), 1–32.

Nguyen, T.-D., Le-Cong, T., Luong, D.-M., Duong, V.-H., Le, X.-B.D., Lo, D., Huynh, Q.-T., 2022. FFL: Fine-grained fault localization for student programs via syntactic and semantic reasoning. In: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 151–162.

Pan, Y., Xiao, X., Hu, G., Zhang, B., Li, Q., Zheng, H., 2020. ALBFL: A novel neural ranking model for software fault localization via combining static and dynamic features. In: 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications. TrustCom, pp. 785–792.

Pane, J.F., Myers, B.A., 1996. Usability issues in the design of novice programming systems.

Papadakis, M., Le Traon, Y., 2015. Metallaxis-FL: mutation-based fault localization. Softw. Test. Verif. Reliab. 25 (5–7), 605–628.

Parnin, C., Orso, A., 2011. Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. ACM, pp. 199–209.
Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B., 2017. Evaluating and improving fault localization. In: 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE, IEEE, pp. 609–620.
Qi, Y., Mao, X., Lei, Y., Wang, C., 2013. Using automated program repair for evaluating the effectiveness of fault localization techniques. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. ACM, pp. 191–201.
Rahman, F., Posnett, D., Hindle, A., Barr, E., Devanbu, P., 2011. BugCache for inspections: hit or miss? In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. ACM, pp. 322–331.
Reis, R., Soares, G., Mongiovi, M., de L. Andrade, W., 2019. Evaluating feedback tools in introductory programming classes. In: IEEE Frontiers in Education Conference, FIE 2019, Cincinnati, OH, USA, October 16-19, 2019. IEEE, pp. 1–7.
Renieres, M., Reiss, S.P., 2003. Fault localization with nearest neighbor queries. In: 18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.. IEEE, pp. 30–39.
Singh, R., Gulwani, S., Solar-Lezama, A., 2013. Automated feedback generation for introductory programming assignments. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, pp. 15–26.
Spohrer, J.G., Soloway, E., 1986. Analyzing the high frequency bugs in novice programs. In: Papers Presented At the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers. pp. 230–251.
Steimann, F., Frenkel, M., Abreu, R., 2013. 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. IEEE, pp. 314–324.
Sullivan, G.M., Feinn, R., 2012. Using effect size—or why the P value is not enough. J. Graduate Med. Educ. 4 (3), 279–282.
Sundararajan, M., Taly, A., Yan, Q., 2017. Axiomatic attribution for deep networks. In: International Conference on Machine Learning. PMLR, pp. 3319–3328.
Weimer, W., Nguyen, T., Le Goues, C., Forrest, S., 2009. Automatically finding patches using genetic programming. In: 2009 IEEE 31st International Conference on Software Engineering. IEEE, pp. 364–374.
Wen, M., Chen, J., Tian, Y., Wu, R., Hao, D., Han, S., Cheung, S.-C., 2019. Historical spectrum based fault localization. IEEE Trans. Softw. Eng. 47 (11), 2348–2368.
Wen, M., Chen, J., Wu, R., Hao, D., Cheung, S.-C., 2018. Context-aware patch generation for better automated program repair. In: Proceedings of the 40th International Conference on Software Engineering. pp. 1–11.
Wong, W.E., Debroy, V., Gao, R., Li, Y., 2013. The dstar method for effective software fault localization. IEEE Trans. Reliab. 63 (1), 290–308.
Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740.
Wong, W.E., Qi, Y., Zhao, L., Cai, K.-Y., 2007. Effective fault localization using code coverage. In: 31st Annual International Computer Software and Applications Conference, vol.. COMPSAC 2007, IEEE, pp. 449–456.
Wong, C., Xiong, Y., Zhang, H., Hao, D., Zhang, L., Mei, H., 2014. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: 30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014. IEEE Computer Society, pp. 181–190.
Wu, Y.-H., Li, Z., Liu, Y., Chen, X., 2020. FATOC: Bug isolation based multi-fault localization by using optics clustering. J. Comput. Sci. Tech. 35 (5), 979–998.
Xu, S., Rajlich, V., 2004. Cognitive process during program debugging. In: Proceedings of the Third IEEE International Conference on Cognitive Informatics, 2004.. IEEE, pp. 176–182.
Yi, J., Ahmed, U.Z., Karkare, A., Tan, S.H., Roychoudhury, A., 2017. A feasibility study of using automated program repair for introductory programming assignments. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, pp. 740–751.
Zhou, J., Zhang, H., Lo, D., 2012. 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, pp. 14–24.
Zou, D., Liang, J., Xiong, Y., Ernst, M.D., Zhang, L., 2019. An empirical study of fault localization families and their combinations. IEEE Trans. Softw. Eng. 47 (2), 332–347.

Zheng Li received the B.Sc. degree in the computer science and technology from the Beijing University of Chemical Technology, Beijing, China in 1996, and the Ph.D. degree in computer science from the CREST Centre, King's College London, London, U.K., in 2009. He is currently a Professor with the College of Information Science and Technology, Beijing University of Chemical Technology. He was a Research Associate with King's College London and University College London, London. He has authored or coauthored more than 60 papers in referred journals or conferences, such as IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, International Conference on Software Engineering, Journal of Software: Evolution and Process, Information and Software Technology, Journal of Systems and Software, International Conference on Software Maintenance, IEEE International Conference on Software Maintenance and Evolution, International Computer Software and Applications Conference, IEEE International Working Conference on Source Code Analysis and Manipulation, and IEEE International Conference on Software Quality, Reliability and Security. His research interests include software engineering, in particular program testing, source code analysis, and manipulation.

Shumei Wu received the B.S. degree in information security from Qingdao University, Nanchang, China, in 2019. She is currently working toward the Ph.D. degree in the Beijing University of Chemical Technology, Beijing, China. Her research interests include fault localization, GUI testing, and software testing.

Yong Liu received the B.Sc. and M.Sc. degrees in the computer science and technology from Beijing University of Chemical Technology, China in 2008 and 2011, respectively. Then he received the Ph.D. degree in control science and engineering from Beijing University of Chemical Technology in 2018. He is with the College of Information Science and Technology at Beijing University of Chemical Technology as an associate professor. His research interests are mainly in software engineering. Particularly, he is interested in software debugging and software testing, such as source code analysis, mutation testing, and fault localization. In these areas, he has published more than ten papers in referred journals or conferences, such as Journal of Systems and Software, Information Sciences, QRS, SATE, and COMPSAC. He is a member of CCF in China, IEEE, and ACM.

Jitao Shen received a B.Sc. degree from Hangzhou Normal University in 2019 and a master's degree from Beijing University of Chemical Technology in 2022. He is currently working at the Agricultural Bank of China. His research interests are fault localization and software testing.

Yonghao Wu received his B.S. degree from Nanchang Hangkong University, China in 2017, and M.Sc. degrees from Beijing University of Chemical Technology, China in 2020. He is currently pursuing his doctorate degree in computer science and technology at Beijing University of Chemical Technology, China. His research interests are fault localization, software testing.

Zhanwen Zhang received the B.S. degree in network engineering from Henan University, Henan, China, in 2021. He is currently working toward the master's degree in the Beijing University of Chemical Technology, Beijing, China. His research interests include fault localization and software testing.

Xiang Chen received the B.Sc. degree in the school of management from Xi'an Jiaotong University, China in 2002. Then he received the M.Sc., and Ph.D. degrees in computer software and theory from Nanjing University, China in 2008 and 2011, respectively. He is with the Department of Information Science and Technology at Nantong University as an associate professor. His research interests are mainly in software engineering. Particularly, he is interested in software maintenance and software testing, such as software defect prediction, combinatorial testing, regression testing, and fault localization. In these areas, he has published over 60 papers in referred journals or conferences, such as IEEE Transactions on Software Engineering, Information and Software Technology, Journal of Systems and Software, IEEE Transactions on Reliability, Journal of Software: Evolution and Process, Software Quality Journal, Journal of Computer Science and Technology, ASE, ICSME, SANER and COMPSAC. He is a senior member of CCF, China and a member of IEEE and ACM.