The Journal of Systems and Software 218 (2024) 112189

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

GBSR: Graph-based suspiciousness refinement for improving fault localization☆

Zheng Li [a], Mingyu Li [a], Shumei Wu [a,*], Shunqing Xu [a], Xiang Chen [b], Yong Liu [a]

[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

Keywords:
Fault localization
Suspiciousness refinement
Graph-based representation
Mutation analysis
PageRank algorithm

ABSTRACT

Fault Localization (FL) is an important and time-consuming phase of software debugging. The essence of FL lies in the process of calculating the suspiciousness of different program entities (e.g., statements) and generating a ranking list to guide developers in their code inspection. Nonetheless, a prevalent challenge within existing FL methodologies is the propensity for program entities with analogous execution information to receive a similar suspiciousness. This phenomenon can lead to confusion among developers, thereby reducing the effectiveness of debugging significantly. To alleviate this issue, we introduce fine-grained contextual information (such as partial code structural, coverage, and features from mutation analysis) to enrich the characteristics of program entities. Graphical structures are proposed to organize such information, where the passed and failed tests are constructed separately with the consideration of their differential impacts. In order to support the analysis of multidimensional features and the representation of large-scale programs, the PageRank algorithm is adopted to compute each program entity's weight. Rather than altering the fundamental FL process, we leverage these computed weights to refine the suspiciousness produced by various FL techniques, thereby providing developers with a more precise and actionable ranking of potential fault locations. The proposed strategy Graph-Based Suspiciousness Refinement (GBSR) is evaluated on 243 real-world faulty programs from the Defects4J. The results demonstrate that GBSR can improve the accuracy of various FL techniques. Specifically, for the refinement with traditional SBFL and MBFL techniques, the number of faults localized by the first position of the ranking list (Top-1) is increased by 189% and 68%, respectively. Furthermore, GBSR can also boost the state-of-the-art learning-based FL technique Grace by achieving a 2.8% performance improvement in Top-1.

1. Introduction

As software systems become larger and more complex, they are plagued by a growing number of faults (Glass et al., 2002; Basili, 1989; Lin et al., 2022). The probability of encountering program faults and the difficulty of resolving them also increase accordingly. Localizing and fixing these faults requires developers to invest significant time and effort. It also heavily relies on the expertise and experience of the developer. Existing research has shown that such a software debugging process constitutes approximately 80% of the overall software development budgets (Planning, 2002). A promising way to facilitate the software debugging process is automatic Fault localization (FL) (Wong et al., 2016; Wu et al., 2020; Liu et al., 2017; Kim et al., 2021; Li et al., 2023; Ju et al., 2014).

FL focuses on automatically diagnosing faulty program entities by utilizing program execution information to calculate the probability of a program entity being faulty (i.e., suspiciousness), and has been extensively studied (Planning, 2002; Pearson et al., 2017; Chen et al., 2017, 2019). Until now, researchers have successively proposed different effective FL techniques, such as Spectrum-based FL (SBFL) (Jones et al., 2002; Jiang et al., 2019; Sarhan and Beszédes, 2022), slicing-based FL (Zhang and Santelices, 2016), Mutation-based FL (MBFL) (Hong et al., 2015; Li et al., 2020a; Papadakis and Le Traon, 2012), and deep learning-based FL (Briand et al., 2007; Sohn and Yoo, 2017; Li et al., 2019; Lou et al., 2021). Among them, SBFL has been widely studied due to its effectiveness and lightweight, while MBFL offers higher FL accuracy (Gong et al., 2015; Wang et al., 2022) because of the abundant information provided by mutants.

However, traditional FL techniques still face the following challenges: (1) traditional FL techniques solely rely on the available information (such as code, coverage, or killed information) represented

☆ Editor: Dr. Aldeida Aleti.

* Corresponding author.
E-mail addresses: wsm@mail.buct.edu.cn (S. Wu), lyong@mail.buct.edu.cn (Y. Liu).

https://doi.org/10.1016/j.jss.2024.112189
Received 5 March 2024; Received in revised form 10 August 2024; Accepted 19 August 2024
Available online 26 August 2024

in the form of binary vectors, resulting in limited utilization of available feature information (Gong et al., 2015; Sasaki et al., 2020), (2) spectrum-based feature representation only indicates the correlation between features without consideration of the importance degree among them (Wang et al., 2022; Jones and Harrold, 2005), and (3) adding feature information and conducting detailed analysis and computation of each feature's impact on FL can significantly increase computational costs (Li et al., 2019; Wu et al., 2023). Due to the limited utilization of feature information in traditional FL techniques and the lack of consideration for the differential impact of different features, entities with similar execution information exhibit similar suspiciousness (Xu et al., 2011). Therefore, excessive similarity in suspiciousness harms the effectiveness of FL techniques.

In this work, we propose a novel strategy GBSR (**G**raph-**B**ased **S**uspiciousness **R**efinement) for suspiciousness refinement. It aims to: (1) strengthen the gap of suspiciousness between similar program entities, and (2) assign faulty program entities higher suspiciousness and non-faulty program entities lower suspiciousness. The core of achieving this goal lies in utilizing the importance of program entities to explicitly consider the contributions of various information. The intuition of GBSR is that different program entities have different information associations, and the entity that is associated with information that has a higher impact on the suspiciousness calculation is of higher importance.

We enhance feature information extraction and utilization by adopting graph-based multi-feature representation. We represent richer information using a fault-oriented interaction graph, which includes not only program entities and tests but also mutants, along with their interrelationships. In the graph, nodes represent program entities, tests, and mutants. Edges depict the structural connections between program entities, the generation relationships of mutants, coverage information, and details about which mutants were killed. We further calculate the weight of each node in the constructed graph to characterize the importance of features. Finally, we refine the suspiciousness based on the obtained importance information, thus obtaining a more effective fault localization result.

Note that GBSR is a strategy of refining suspiciousness without altering the existing FL techniques. It modifies and improves suspicion levels based on the original FL techniques, making it highly applicable. Moreover, GBSR calculates node weights based on the structural information of the graph to represent the importance of feature relationships. We utilize the PageRank (Page et al., 1998; Langville and Meyer, 2004) algorithm due to its simplicity and efficiency, making it suitable for high-dimensional feature information and large-scale program computations.

Furthermore, based on previous findings in fault localization, the impact of passed tests and failed tests on FL differs significantly (de Oliveira et al., 2018), where failed tests positively affect program entity weights and passed tests have a negative impact. Therefore, GBSR splits tests and constructs a graph structure comprising passed or failed tests during the graph representation process. Separating the consideration of these two types of tests prevents cross-impact on other features, thereby enhancing the effectiveness of feature importance computation.

We evaluate GBSR on 243 real-world faulty programs from the widely used benchmark Defects4J (V2.0.0) (Just et al., 2014). The results show that GBSR significantly improves the localization accuracy of traditional SBFL (Abreu et al., 2006) and MBFL (Papadakis and Le Traon, 2015; Moon et al., 2014), and outperforms PRFL (Zhang et al., 2017) and PRMA (Yan et al., 2023). Specifically, in terms of $Top1$, $Top3$ and $Top5$, GBSR can localize 189%, 84%, 54% and 68%, 30%, 12% faults more than the traditional SBFL and MBFL, respectively. Additionally, GBSR outperforms PRFL and PRMA with an average improvement of 80%, 46%, and 27%. Moreover, GBSR can also boost the state-of-the-art learning-based FL technique, Grace (Lou et al., 2021), which has a 2.8% improvement in Top-1. These promising results highlight the potential research direction of combining GBSR with deep learning-based FL techniques for further exploration.

In summary, the refinement strategy we have proposed is general for various FL techniques and remains unaffected by the choice of different FL algorithms or formulas. This generalizability ensures that GBSR can be seamlessly integrated with a wide array of FL techniques, enhancing their overall performance and utility across diverse software debugging scenarios.

This paper makes the following contributions:

- **The Fine-grained Graphical Representation** that integrally reserves enriched contextual information by representing the static information (e.g., mutants and mutation relationship) and the dynamic information (e.g., killed information) of the program into an interaction graph. In particular, this mutation and killed representation could provide fine-grained information regarding the impact of program entities on program behavior.
- **A Novel Strategy for Suspiciousness Refinement** that can refine the suspiciousness by assigning weights to different program entities based on contextual information within the graph, thereby enhancing the performance of various FL techniques.
- **The Evaluation** on widely used Defects4J shows the effectiveness of GBSR in refining suspiciousness for various FL techniques, and its performance improvement is not affected by the formula used.

The remainder of this paper is structured as follows. Section 2 provides background information on fault localization. Section 3 demonstrates the advantages of our strategy through examples, followed by Section 4, which presents our approach in detail. Sections 5 and 6 describe our experimental setup and findings, respectively. Section 7 discusses threats to the validity of our approach, while Section 8 discusses related work. Finally, Section 9 concludes and presents future work.

Replication Package: To facilitate other researchers to follow and replicate our study, we share our experimental subjects, source code, and the experimental results of GBSR in a Github repository.[1]

2. Background

In this section, we introduce the background of FL techniques (Section 2.1) and the PageRank algorithm (Section 2.2).

2.1. Fault localization techniques

2.1.1. Mutation-based fault localization

MBFL (Hong et al., 2015; Li et al., 2020a; Papadakis and Le Traon, 2012; Gong et al., 2015; Wang et al., 2022) is a highly accurate technique for fault localization, which utilizes mutation analysis (Papadakis et al., 2019) to derive a suspiciousness for each program entity and identifies the location of the fault. Specifically, given a faulty program p and a set of test T, MBFL partitions T into two subsets (i.e., passed tests T_p and failed tests T_f) based on their test results. Then artificial faults (mutation operators) into each statement covered by failed tests to generate mutants M. Subsequently, all mutants must undergo comprehensive testing. If the outcome differs from that of the original program, it indicates that M is killed by the test. Otherwise, the mutant is not killed.

In MBFL, the kill information for each program entity is counted as four numbers: a_{kf}, a_{nf}, a_{kp}, and a_{np}. Specifically, a_{kf} refers to the number of failed tests that killed M. a_{kp} is the number of passed tests that killed M. T_{np} refers to the number of passed tests that did not kill M. a_{nf} is the number of failed tests that did not kill M. Then MBFL calculates the suspiciousness of each program entity using a formula (e.g., Tarantula Li et al., 2020b; Jones and Harrold, 2005, Jaccard Liu

[1] https://github.com/my0501/GBSR.git

Table 1
Suspiciousness formulas for SBFL and MBFL.

Name	SBFL-Formula	MBFL-Formula
Tarantula	$\frac{\frac{a_{ef}}{a_{ef}+a_{ep}}}{\frac{a_{ef}}{a_{ef}+a_{nf}}+\frac{a_{ep}}{a_{ep}+a_{np}}}$	$\frac{\frac{a_{kf}}{a_{kf}+a_{kp}}}{\frac{a_{kf}}{a_{kf}+a_{nf}}+\frac{a_{kp}}{a_{kp}+a_{np}}}$
Jaccard	$\frac{a_{ef}}{a_{ef}+a_{nf}+a_{ep}}$	$\frac{a_{kf}}{a_{kf}+a_{nf}+a_{ep}}$
Dstar^2	$\frac{a^2_{ef}}{a_{ep}+a_{nf}}$	$\frac{a^2_{kf}}{a_{kp}+a_{nf}}$
Wong1	a_{ef}	a_{kf}
Hamming	$a_{ef}+a_{np}$	$a_{kf}+a_{np}$
Hamann	$\frac{a_{ef}+a_{np}-a_{ep}-a_{nf}}{a_{ef}+a_{nf}+a_{ep}+a_{np}}$	$\frac{a_{kf}+a_{np}-a_{kp}-a_{nf}}{a_{kf}+a_{nf}+a_{kp}+a_{np}}$

et al., 2018; Chen et al., 2002, Dstar Wong et al., 2013, Wong1 Xie et al., 2013, Hamming Dao et al., 2021, and Hamann (Naish et al., 2011) as shown in Table 1).

2.1.2. Spectrum-based fault localization

SBFL (Abreu et al., 2006; Jones and Harrold, 2005) is a coverage-based fault localization technique. Due to its lightweight and effectiveness, SBFL is widely used in fault localization, making it a popular choice for both researchers and practitioners (Li et al., 2023; Sasaki et al., 2020). In contrast to MBFL, SBFL employs program coverage information rather than killed information from mutation testing. Specifically, for each program entity e, the SBFL technique distinguishes tests into four subsets T_{ef}, T_{nf}, T_{ep}, and T_{np} by analyzing the test outcomes and the coverage of e during testing. Among them, T_{ef} refers to failed tests covering e, T_{ep} refers to passed tests covering e, T_{nf} refers to failed tests uncovering e, and T_{np} refers to passed tests uncovering e. Similarly, the numbers of elements in these sets are defined as a_{ef}, a_{ep}, a_{nf}, and a_{np}. Finally, SBFL uses the calculation formulas shown in Table 1 to calculate the suspiciousness for each program entity. However, since SBFL only considers coverage information of test execution and does not take into account other information (e.g., program semantic information, structural information, etc.), this significantly reduces the time cost during the localization process, and it also results in a loss of accuracy.

2.1.3. Learning-based fault localization

The outstanding performance of deep learning (LeCun et al., 2015; Xia et al., 2021) has motivated many studies that integrate it with fault localization (Qian et al., 2023). For example, Lou et al. introduced a coverage-based FL technique, Grace (Lou et al., 2021). It leverages detailed coverage information and fine-grained code structure through graph-based representation. Then, Grace uses the graph neural network to identify faulty program entities, improving the precision of FL. However, the effectiveness of deep learning models, including Grace, is constrained by their need for extensive high-quality labeled data. This requirement is particularly challenging in fault localization, where obtaining accurate ground truth labels is often difficult and resource-intensive. Additionally, the black-box nature of deep learning models makes it challenging to interpret and explain their decisions, which can hinder the trust and adoption of the proposed technique in practical software debugging.

2.2. PageRank algorithm

PageRank algorithm (Page et al., 1998) is a method that calculates the importance and authority of web pages based on link analysis. It abstracts web pages as nodes in a graph, where the link relationships between web pages are represented as directed edges between the nodes. PageRank algorithm assumes an initial distribution R_0 and iteratively calculates the PageRank values of all nodes until convergence is achieved. In each iteration, the PageRank value of each node is updated based on the PageRank values of the nodes that link to it and the number of outbound links from those nodes. Specifically, the PageRank value for each node is the sum of the PageRank values of all nodes pointing to it divided by the out-degree of that pointing node. This can be expressed mathematically as follows:

$$PR(v_i) = \sum_{v_j \in \mathbb{M}(v_i)} \frac{PR(v_j)}{L(v_j)} \tag{1}$$

where $PR(v_i)$ denotes the PageRank value of node v_i, $\mathbb{M}(v_i)$ represents the set of nodes that have directed edges pointing to node v_i, and $L(v_j)$ represents the number of outgoing directed edges from node v_j.

The initial probability distribution assigned to each node is represented as R_0, and $\mathcal{M}$ is the transition matrix. At each time step (1, 2, and 3), the probabilities of visiting each node are obtained by applying matrix $\mathcal{M}$ to the initial distribution R_0, resulting in $\mathcal{M}R_0$, $\mathcal{M}^2R_0$ and $\mathcal{M}^3R_0$, respectively. As the number of iterations increases, a certain limit is eventually reached.

$$\lim_{t\to\infty} \mathcal{M}^t R_0 = R \tag{2}$$

where R is defined as the PageRank value of each node, and it satisfies the stationary distribution that satisfies the Markov chain (Ching and Ng, 2006) (i.e., $\mathcal{M}R = R$). This indicates that, following the transformation using transition matrix $\mathcal{M}$, the probabilities of visiting each node reach a state of equilibrium and remain constant.

However, the fundamental definition of the PageRank algorithm has limitations in addressing exceptional cases such as isolated nodes. Therefore, a generalized formulation of PageRank has been proposed to improve its applicability and effectiveness across various domains. This approach addresses the issue of isolated nodes by incorporating damping factors and teleportation probabilities, which facilitate random transitions to any node in the graph. Consequently, even isolated nodes can acquire some PageRank score.

$$R' = (\delta\mathcal{M} + \frac{1-\delta}{n}\mathcal{E})R \tag{3}$$

where the damping factor, denoted as δ (0 to 1), is a parameter that influences the probability of random transitions between nodes in the graph. The matrix $\mathcal{E}$ represents an all-ones matrix, while vectors R and R' are n-dimensional representations of node visitation probabilities. As δ approaches 1, transitions primarily follow the transition matrix $\mathcal{M}$, whereas as δ approaches 0, the probability of random visits to all nodes becomes equal.

3. Motivating example

To illustrate the limitations of existing fault localization techniques, we present a motivating example (Lang-1) in this section.

Table 2 shows a process of SBFL. The first column represents the ID of methods (m_0, m_1, and m_2), where the fault occurred in m_1. The second column represents the signature of the method, and the third column represents the coverage information of tests, encompassing four passed tests (pt_0, pt_1, pt_2, and pt_3) and one failed test (ft_0). In this table, '*' indicates that the corresponding program entity was covered by the test, and '-' indicates that the program entity was not covered by the test. The last two columns represent the suspiciousness and the ranking list of each program entity computed by Dstar (Wong et al., 2013) for traditional and GBSR combined SBFL, respectively.

Table 3 illustrates the process of MBFL conducted on Lang-1. The first two columns represent the ID of the program methods and the mutants' ID. The third column indicates information on whether each of the five tests kills the mutant. And '*' denotes that a test killed the corresponding mutant in that row, while '-' indicates that it did not. The last two columns respectively represent the suspiciousness and the ranking list of each program entity computed by Dstar (Wong et al., 2013) for traditional and GBSR combined MBFL.

SBFL assigns identical suspiciousness to all program entities, as m_0, m_1, and m_2 share the same values for a_{ef} and a_{nf}. While MBFL does assign the highest level of suspiciousness to m_1, it is not the sole entity with maximum suspiciousness. Both m_0 and m_1 share the same level of

Table 2
The motivating example Lang-1 for SBFL The highlighted line indicates the faulty method and corresponding suspiciousness and rank. The suspiciousness was calculated based on SBFL through Dstar[2].

m_{ID}	Method Signature	Coverage					SBFL		$SBFL_G$	
		ft_0	pt_0	pt_1	pt_2	pt_3	Sus	Rank	Sus	Rank
m_0	org/apache/commons/lang3/StringUtils@isBlank	*	*	*	*	–	0.33	3	0.41	2
m_1	**org/apache/commons/lang3/math/NumberUtils@createNumber**	*	*	*	*	-	**0.33**	**3**	**0.53**	**1**
m_2	org/apache/commons/lang3/math/NumberUtils@createInteger	*	*	*	–	*	0.33	3	0.38	3

Table 3
The motivating example Lang-1 for MBFL. The highlighted line is the faulty method and corresponding suspiciousness and rank. The suspiciousness was calculated based on MBFL through Dstar[2].

m_{ID}	M_{ID}	Killed information					MBFL		$MBFL_G$	
		ft_0	pt_0	pt_1	pt_2	pt_3	Sus	Rank	Sus	Rank
m_0	M_0	*	*	–	–	–	1.0	2	1.25	2
	M_1	*	*	*	*	–				
m_1	M_2	*	*	–	–	–	**1.0**	**2**	**1.60**	**1**
	M_3	*	*	–	–	–				
m_2	M_4	–	–	–	–	–	0	3	0	3

suspiciousness, leading to a decrease in the accuracy of FL. During the localization process, we observe that the core factor affecting MBFL and SBFL results lies in the statistical count of mutants killed by tests and program entities covered by tests. However, analyzing coverage and killed information in such a simplistic manner leads to many program entities with comparable execution patterns having nearly identical suspiciousness (Xu et al., 2011). Consequently, without additional information for further analysis, it is challenging to obtain a more optimal and accurate suspiciousness ranking list.

In contrast, GBSR fully considers the different correlations among various information in a graph-based way. It uses the PageRank algorithm to integrate and calculate the weights of each program entity using context information provided by the graph structure. This weight information can provide extra guidance to established FL techniques, helping them refine their suspiciousness, and consequently improving the accuracy of FL. For example, in the process of mutation analysis, each mutant possesses distinct killed information, leading to varying relationships represented in the graph. More specifically, mutants killed by more failed tests hold higher value. Correspondingly, program entities that generate mutants with higher values become more crucial, thus being assigned higher suspiciousness.

As indicated in the last column of Tables 2 and 3, when we use the GBSR strategy in combination with both SBFL and MBFL techniques, the suspiciousness for methods m_0, m_1, and m_2 are (0.41, **0.53**, 0.38) and (1.25, **1.60**, 0), respectively. It can be observed that GBSR successfully locates m_1 at the position with the highest suspiciousness (i.e., achieving more accurate results), whether in combination with SBFL or MBFL. The complete process is available in the Github repository[1].

4. Our approach

Traditional fault localization techniques calculate suspiciousness by utilizing information from coverage or killed mutants. However, as discussed in Section 3, they neglect the structural relationships between different information during the fault localization process. Furthermore, it is important to recognize that not all information is equally valuable when it comes to pinpointing faults. In the process of fault localization, distinguishing between relevant and irrelevant information is crucial for effective and efficient debugging. Our proposed novel suspiciousness refinement strategy GBSR can integrally reserve rich contextual information by representing code structure, coverage, mutation relationship, and killed information into an interaction graph. Then it calculates the importance of program entities from the contextual information in the graph structure and then refines suspiciousness from existing fault localization. The overview of GBSR is shown in Fig. 1. More specifically, given a program under test, tests, and the corresponding mutants, GBSR employs three stages to achieve suspiciousness refinement: (1) **Construction of the graph**: we abstract the call relationships between methods, the code structure within the program, and the corresponding relationships of generated mutants, information for killed mutants, and coverage information into a graphical structure to fully utilize existing information. (2) **Weight Calculation**: we leverage the graph structure information to construct adjacency matrices separately for failed and passed tests. Applying the PageRank algorithm to calculate node weights in both cases, we integrate these weights to obtain the importance of program entities. (3) **Suspiciousness Refinement**: we refine suspiciousness calculated by traditional FL based on the weights assigned to different program entities.

4.1. Construction of the graph

4.1.1. Representation of the graph

For a better illustration, we present the formal description of a fault-oriented interaction graph. A faulty program p can be represented as a fault-oriented interaction graph ($\mathcal{G}$), in which nodes represent the program entities, tests, and mutants, while edges are the different relationships between them.

Definition 1 (*Fault-oriented Interaction Graph*)**.** The fault-oriented interaction graph $\mathcal{G}$ for a faulty program p with the test suite T is a directed heterogeneous graph, denoted by a 2-tuple$\langle \mathcal{V}, \mathcal{E} \rangle$, where:

(1) $\mathcal{V}$ consists of four types of nodes, including method nodes $\mathcal{V}_m$, statement nodes $\mathcal{V}_s$, test nodes $\mathcal{V}_t$, and mutant nodes $\mathcal{V}_M$. That is, $\mathcal{V} = \{\mathcal{V}_m, \mathcal{V}_s, \mathcal{V}_t, \mathcal{V}_M\}$. Specifically, we regard each method and statement in program p as individual method node $v_m \in \mathcal{V}_m$ and statement node $v_s \in \mathcal{V}_s$, respectively. Similarly, we represent each test t in test suite T as individual test node $v_s \in \mathcal{V}_s$. As for mutant nodes, each the mutant node $v_M \in \mathcal{V}_M$ can represent a special mutant generated by applying mutant operators on statements $\mathcal{V}_s$ in program p.

(2) Like $\mathcal{V}$, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ has a type of directed edges and four types of undirected edge, which include a set of invoking edges between method nodes $\mathcal{V}_m$, a set of structural edges between method nodes $\mathcal{V}_m$ and statement nodes $\mathcal{V}_s$, a set of coverage edges between statement nodes $\mathcal{V}_s$ and test nodes $\mathcal{V}_t$, a set of mutation edges between statement nodes $\mathcal{V}_s$ and mutant nodes $\mathcal{V}_M$, and a set of killing edges between test nodes $\mathcal{V}_t$ and mutant nodes $\mathcal{V}_M$. For an edge $e = \langle v_1, v_2 \rangle$ or $\mathbf{e} = (v_1, v_2)$ in $\mathcal{E}$, its properties are different depending on the type of nodes it connects to, as shown in the following:

- If $v_1 \in \mathcal{V}_m$ and $v_2 \in \mathcal{V}_m$, e is a invoking edge, representing that the method v_1 calls the method v_2.
- If $v_1 \in \mathcal{V}_s$ and $v_2 \in \mathcal{V}_m$ (or $v_1 \in \mathcal{V}_m$ and $v_2 \in \mathcal{V}_s$), $\mathbf{e}$ is a structural edge. It represents the statement v_1 (or v_2) belongs to the method v_2 (or v_1).
- If $v_1 \in \mathcal{V}_s$ and $v_2 \in \mathcal{V}_t$ (or $v_1 \in \mathcal{V}_t$ and $v_2 \in \mathcal{V}_s$), $\mathbf{e}$ is a coverage edge, which means that the statement v_1 (or v_2) is executed by the test v_2 (or v_1).
- If $v_1 \in \mathcal{V}_M$ and $v_2 \in \mathcal{V}_s$ (or $v_1 \in \mathcal{V}_s$ and $v_2 \in \mathcal{V}_M$), $\mathbf{e}$, a mutation edge, represents that the mutant v_1 (or v_2) is produced by modifying the statement v_2 (or v_1).

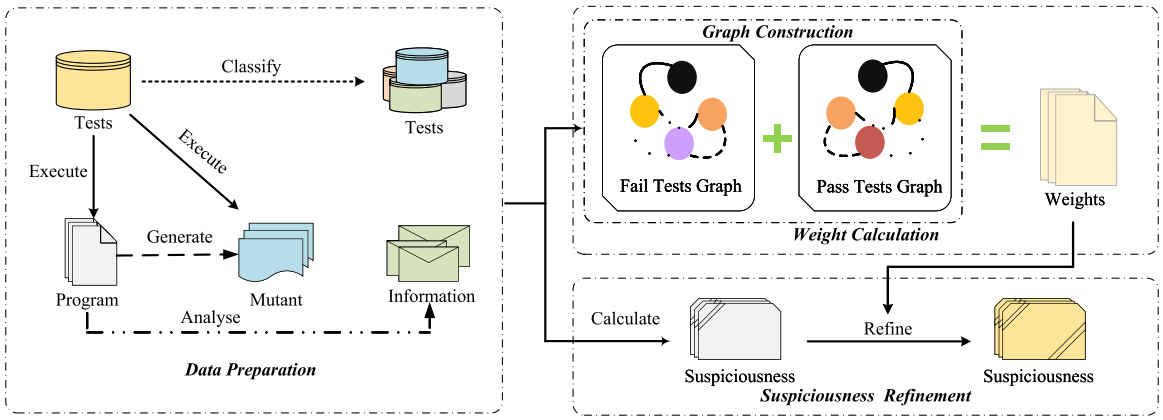


Fig. 1. The overall Framework of GBSR.

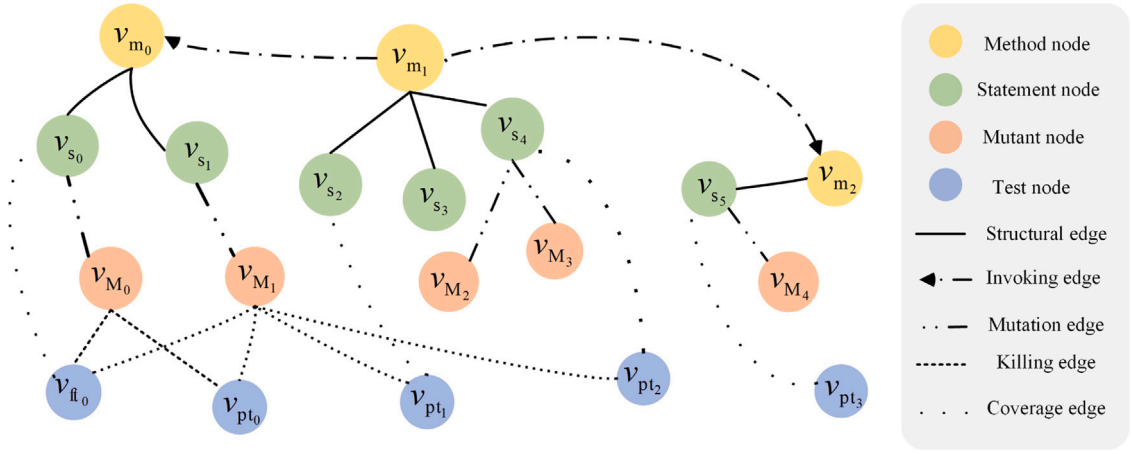


Fig. 2. Partial Graphical Structure of Motivating Example.

- If $v_1 \in \mathcal{V}_t$ and $v_2 \in \mathcal{V}_M$ (or $v_1 \in \mathcal{V}_M$ and $v_2 \in \mathcal{V}_t$), **e** is a killing edge, suggesting that represents that the test v_1 (or v_2) can kill the mutant v_2 (or v_1).

In the motivating example, Fig. 2 illustrates method, statement, test, mutant, and different relationships between them. The code structure, coverage, mutation relationship, and killed information in the buggy program also are represented in one graph, i.e., fault-oriented interaction graph, as shown in Definition 1.

4.1.2. Construction of the graph

Algorithm 1 presents the workflow for constructing the fault interaction-oriented graph. It takes the given program p, the test suite T, and mutation operators O as the input, and outputs the fault interaction-oriented graph $\mathcal{G}$ associated with p. The specific process can be summarized as follows:

Step 1: Collecting coverage relationship. We first execute the program p with the test suite T, recording coverage information and test results. For a test t, if its execution result aligns with expectation, we consider it a passed test $pt \in PT$; otherwise, we consider it a failed test $ft \in FT$. To restrain the scale of the graph, we only consider suspicious statements $\mathcal{V}_s$ covered by at least one failed test, and tests $\mathcal{V}_t$ covering at least one suspicious statement during graph construction. Then, we use the tuple $\langle v_s, v_t \rangle \in \mathcal{E}_{cov}$ to record coverage relationships between these statements and tests, where $v_s \in \mathcal{V}_s$, and $v_t \in \mathcal{V}_t$.

Step 2: Extracting structure relationship. We regard the methods to which statement nodes belong as method nodes $\mathcal{V}_m$, and use a parser to extract their abstract syntax trees. Such structural relationships are recorded as the tuple $\langle v_m, v_s \rangle \in \mathcal{E}_{str}$, where $v_m \in \mathcal{V}_m$, and $v_s \in \mathcal{V}_s$. Meanwhile, we analyze function invocation among methods, and represent them as the tuple $\langle v_{m_i}, v_{m_j} \rangle \in \mathcal{E}_{cal}$, where $v_{m_i} \in \mathcal{V}_m$, and $v_{m_j} \in \mathcal{V}_m$.

Step 3: Generating and executing mutants. By applying mutation operators O to statements, we generate multiple mutants. These mutants and mutation relationship are recorded into the list $\mathcal{V}_M$ and the tuple $\mathcal{E}_{mul} = \{\langle v_M, v_s \rangle\}$, where $v_M \in \mathcal{V}_M$, and $v_s \in \mathcal{V}_s$. Next, we execute the test cases T on the mutants $\mathcal{V}_M$. We compare the execution result of each mutant $v_M \in \mathcal{V}_M$ with the result of the program p on the test $t \in \mathcal{V}_t$. If the results differ, we consider the mutant v_M is killed the test t and represent it as a tuple $\langle v_M, t \rangle \in E_{kil}$.

Step 4: Constructing the fault interaction-oriented graph $\mathcal{G}$. We use the fault interaction-oriented graph $\mathcal{G}$ for p to integrate all the previously recorded information. It includes a sequential list recording all statements $\mathcal{V}_s$, methods $\mathcal{V}_m$, tests $\mathcal{V}_t$, and mutants $\mathcal{V}_M$, as well as an adjacency matrix integrating their relationships (i.e., $\mathcal{E}_{cal}$, $\mathcal{E}_{str}$, $\mathcal{E}_{mut}$, $\mathcal{E}_{cov}$, and $\mathcal{E}_{kil}$).

4.2. Weight calculation

To calculate the weights of program entities using GBSR, we need (1) the matrix representation of the graph structure obtained from Section 4.1, and (2) the calculation through the PageRank algorithm. To generate the adjacency matrix $\mathcal{M}$, GBSR requires the overall structural information of the program during the fault localization process. We can observe that each node may have a different number of connections. For an edge $e = \langle v_i, v_j \rangle$ or $e = (v_i, v_j)$ in the graph, the number of edges of the same type connected to v_i is n, and the initial weight of

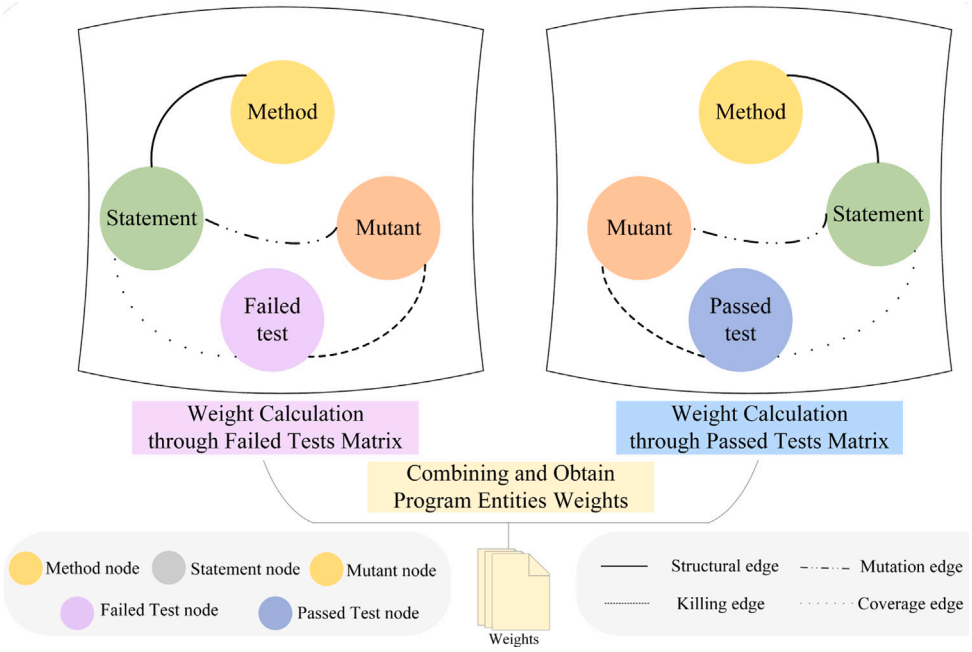


Fig. 3. Flowchart for weight calculation.

Algorithm 1 Construction of Fault-oriented Interaction Graph

Input: program p, test suite T, mutation operators O
Output: fault-oriented interaction graph $\mathcal{G}$

$m, S, \mathcal{E}_{cal}, \mathcal{E}_{str} \leftarrow$ AST constructed from p
For S_i in S **do**
  $M(S_i), \mathcal{E}_{mut} \leftarrow$ inject faults ruled by O to S_i
For t in T **do**
  For S_i in S **do**
    If S_i is covered by t :
      $\mathcal{E}_{cov} < S_i, t > \leftarrow S_i$ is covered by t
  For M_i in M **do**
    If results of p and M_i under t are not same:
      $\mathcal{E}_{kil} < M_i, t > \leftarrow M_i$ is killed by t
$Size \leftarrow$ number of m, S, M, and T
$\mathcal{G} \leftarrow$ a graph matrix with size of $|Size||Size|$
For i in range $|Size|$ **do**
  For j in range $|Size|$ **do**
    $G(i, j) \leftarrow$ set of edges
return $\mathcal{G}$

edge e is $1/n$. In order to be more specific, we give examples of directed edge and undirected edge respectively:

- For $e_1 = \langle v_{m_1}, v_{m_0} \rangle$, m_1 has two invoking edges of the same type $e_1 = \langle v_{m_1}, v_{m_0} \rangle$ and $e_2 = \langle v_{m_1}, v_{m_2} \rangle$, so the initial weight of edge e_1 is set to 1/2. Similarly, the initial weight of e_2 is also 1/2.
- For $e_1 = (v_{M_1}, v_{ft_0})$, M_1 has four killing edges of the same type $e_1 = (v_{M_1}, v_{ft_0})$, $e_2 = (v_{M_1}, v_{pt_1})$, $e_3 = (v_{M_1}, v_{pt_2})$, and $e_4 = (v_{M_1}, v_{pt_3})$ so the weight of edge e_1 is set to 1/4. Likewise, the initial weights of e_2, e_3, and e_4 are also 1/4.

With these pieces of information, the adjacency matrix $\mathcal{M}$ can be constructed as follows:

$$\mathcal{M} = \begin{vmatrix} \mathcal{M}_{mm} & \mathcal{M}_{sm} & \mathbf{0} & \mathbf{0} \\ \mathcal{M}_{ms} & \mathbf{0} & \mathcal{M}_{Ms} & \mathcal{M}_{ts} \\ \mathbf{0} & \mathcal{M}_{sM} & \mathbf{0} & \mathcal{M}_{tM} \\ \mathbf{0} & \mathcal{M}_{st} & \mathcal{M}_{Mt} & \mathbf{0} \end{vmatrix} \tag{4}$$

where $\mathcal{M}$ is the transition matrix, $\mathcal{M}_{mm}$ represents the method-to-method invocation relationships, $\mathcal{M}_{ms}$ and $\mathcal{M}_{sm}$ represent the associations between methods and statements, $\mathcal{M}_{sM}$ and $\mathcal{M}_{Ms}$ represent the correspondences between mutants and statements, $\mathcal{M}_{st}$ and $\mathcal{M}_{ts}$ represent the relationships between statements and tests, $\mathcal{M}_{Mt}$ and $\mathcal{M}_{tM}$ represent the relationships between mutants and tests. It is worth noting that we consider partitioning the transition matrix into two cases: passed tests and failed tests, instead of including all tests together in the transition matrix. We consider the killed information obtained from failed tests to have a positive influence on fault localization, while the killed information from passed tests has a negative influence on the final fault localization results. Therefore, we handle these two types of tests separately in our approach, as shown in Fig. 3.

To accommodate this, we generate two types of adjacency matrices: one for failed tests and another for passed tests. These matrices capture the respective relationships and connections between program entities in each scenario. Subsequently, we compute the weights of each program entity separately in two matrices using the PageRank algorithm, as shown in Eq. (3). The computed vector R' includes not only the weights of the method nodes but also the weight information of the statement, mutant, and test nodes. In this paper, we only consider the weights of the method nodes.

The final method weight is determined by combining the weights from the two types of matrices, as shown in the following Eq. (5):

$$w(m_i) = PR_f(m_i) - \alpha PR_p(m_i) \tag{5}$$

where w is the weight of a method. PR_f and PR_p represent method nodes' weights calculated from adjacency matrices corresponding to failed tests and passed tests respectively. m_i denotes the i_{th} method. α represents the proportion of the impact of the passed test on calculating the weights of methods. The value of α is limited between 0 and 1, where 0 means the passed tests have no impact on the method weights, and 1 means that a passed test has equal impact as a failed test.

4.3. Suspiciousness calculation

Most previous fault localization techniques assess the likelihood of a program entity containing a fault. However, due to the limitations in information utilization by traditional techniques, there is

still performance improvement room in the accuracy of suspiciousness computation. During the process of suspiciousness calculation, GBSR refines the suspiciousness of existing fault localization techniques by taking into account the influence of other information on program entities.

Specifically, we utilize the weights calculated for each method in Section 4.1 to refine the suspiciousness provided by existing FL techniques using the following equation.

$$Sus(m_i) = (1 + \frac{w(m_i)}{\sum_{j=1}^{n} w(m_j)}) Sus_o(m_i) \quad (6)$$

where Sus represents the suspiciousness refined through GBSR, and Sus_o represents the suspiciousness calculated by existing FL technique. Meanwhile, $w(m_i)$ denotes the weight value of m_i obtained in Eq. (5). The formula $\frac{w(m_i)}{\sum_{j=1}^{n} w(m_j)}$ is used to calculate the normalized weight of method m_i (where n represents the total number of executable methods). GBSR combines the traditional fault localization suspiciousness with program entities' weights and applies appropriate influences. This leads to the optimization of suspiciousness calculation and an improvement in the accuracy of fault localization.

5. Experimental design

5.1. Research questions

RQ1: How does GBSR impact different fault localization techniques?

In RQ1, we aim to determine whether the GBSR strategy can effectively improve the accuracy of fault localization. In this experiment, we combine GBSR with traditional fault localization techniques (SBFL and MBFL) to form $SBFL_G$ and $MBFL_G$. We use Dstar to calculate the suspiciousness because it has been proven to be a highly effective suspiciousness calculation formula (Wong et al., 2013). We also compare the traditional techniques (such as PRFL and PRMA) as baselines to $SBFL_G$ and $MBFL_G$. In addition, we combine the GBSR strategy with the state-of-the-art technique Grace to verify the effectiveness of GBSR in learning-based techniques.

RQ2: How does the combination of GBSR with different suspiciousness calculation formulas affect the performance of our approach GBSR?

In RQ2, we aim to verify the generalizability of GBSR in computing suspiciousness refinement by combining it with various formulas of traditional fault localization techniques. In this experiment, we combine GBSR with six commonly used formulas (i.e., Tarantula Li et al., 2020b; Jones and Harrold, 2005, Jaccard Liu et al., 2018; Chen et al., 2002, Dstar Wong et al., 2013, Wong1 Xie et al., 2013, Hamming Dao et al., 2021, and Hamann (Naish et al., 2011)) for traditional SBFL and MBFL. We evaluate the effectiveness of these approaches in terms of $Top\text{-}n$ and MAR.

RQ3: How does the impact of experimental parameters affect the results?

In RQ3, we aim to analyze the impact of different parameters on the effectiveness of GBSR. To examine the effects of various parameter configurations on GBSR, we conducted experiments using 121 parameter pairs. These parameter pairs consist of α (in Eq. (5)) and δ (in Eq. (3)). Specifically, we set the values of α and δ to range from 0 to 1 in increments of 0.1 (i.e., 0.0, 0.1, 0.2, ..., 1.0), resulting in 11 possible values for each parameter. By combining these values, we obtained $11 \times 11 = 121$ parameter configurations.

RQ4: How does GBSR impact on time costs at each stage?

In RQ4, we aim to evaluate the time overhead of the GBSR strategy. We analyze the time spent on graph construction, mutant generation, and execution, as well as the time required for calculating program entity weights using the PageRank algorithm for each subject to assess the time overhead of GBSR.

Table 4
Details of the experiment benchmark.

Subject	Name	#Test	#Loc	#Version	#Fault
Lang	Commons-lang	2,245	22K	58	83
Chart	Jfreechart	2,205	96K	24	37
Cli	Commons-cli	361	4K	36	45
JxPath	Commons-jxpath	401	21K	22	36
Math	Commons-math	3,602	85K	103	137
Total	—	8,814	228K	243	328

5.2. Benchmark

Our experiments are conducted based on the Defects4J dataset (Just et al., 2014). Defects4J is a widely used benchmark dataset in the field of software engineering for evaluating automated program repair and fault localization techniques (Martinez et al., 2017; An et al., 2021). Defects4J consists of a set of real-world Java programs with known defects. Each program in the dataset has a corresponding set of tests, including both passed and failed tests.

To evaluate the performance of GBSR, we selected 243 faulty programs from 5 subjects (Lang, Chart, Cli, JxPath, and Math), comprising 328 faults in total. Our filtering criteria consider the following aspects: (1) Exclusion of versions that lack coverage information, rendering them unavailable for analysis, and (2) Focusing solely on methods that are part of the faulty version. Any fault not originating from the faulty version is considered out of scope. Table 4 shows the details of the subjects which we selected from five open-source Java projects, including "Subject", "Name", "#Test", "#Loc", "#Version" (the number of versions), "#Fault" (the number of contained fault methods).

5.3. Evaluation metrics

We used the following three widely used metrics in previous fault localization studies (Li et al., 2020a; Gong et al., 2015; Wang et al., 2022).

5.3.1. Top-n

$Top\text{-}n$ measures the effectiveness of fault localization by counting the number of faulty program entities that have at least one faulty entity ranked within the top N positions (Li and Zhang, 2017). Apparently, a fault localization technique with a higher $Top\text{-}n$ is better than others. In our study, we considered three commonly used values for N (i.e., 1, 3, and 5).

5.3.2. Mean average rank

Mean Average Rank (MAR) is calculated by computing the average ranking of all faulty program entities in a version, and then taking the average value of the average rankings across all versions within a subject (Wang et al., 2022). The lower the MAR value, the better the fault localization performance.

5.3.3. EXAM

$EXAM$ measures (de Oliveira et al., 2018) the efficiency of a technique in finding faulty program entities. It represents the percentage of program entities that need to be inspected before the faulty program entities are found. The formula is defined as follows.

$$EXAM = \frac{rank\ of\ faulty\ element}{number\ of\ executable\ element} \quad (7)$$

A lower $EXAM$ value indicates a better fault localization technique, as it means fewer program entities need to be examined before identifying the actual faulty entities. In accordance with previous studies (Lou et al., 2021; Wu et al., 2023; Lou et al., 2020), when there are different program entities with the same suspiciousness, they adopt the worst ranking. This means that if three correct program entities and one faulty program entity are tied with each other, and there are k other program entities with higher suspiciousness than them, we consider all of them to be ranked at position $k + 4$.

Table 5
The name, description, and the corresponding examples of mutation operators used in our study.

Name	Description	Example
ABS	Replaces a variable by its negation.	$a \rightsquigarrow -a$
AOD	Replaces an arithmetic expression by one of the operand.	$a+b \rightsquigarrow a$
AOR	Replaces an arithmetic expression by another one.	$a+b \rightsquigarrow a*b$
ROR	Replaces the relational operators with another one. It applies every replacement.	$<\rightsquigarrow \geq$, $<\rightsquigarrow \leq$
UOI	Replaces a variable with a unary operator or removes an instance of a unary operator.	$a \rightsquigarrow a++$
CRCR	Replaces a constant a with its negation, or with 1, 0, $a+1$, $a-1$.	$a \rightsquigarrow -a$, $a \rightsquigarrow a-1$
OBBN	Replaces the operators & by | and vice versa.	$a\&b \rightsquigarrow a\|b$
Math	Replaces a numerical op. by another one (single replacement).	$+ \rightsquigarrow -$
Invert Neg.	Removes the negative from a variable.	$-a \rightsquigarrow a$
Inline Const.	Replaces a constant by another one or increments it.	$1 \rightsquigarrow 0$, $a \rightsquigarrow a+1$
Cond. Bound.	Replaces one relational operator instance with another one (single replacement).	$<\rightsquigarrow \leq$
Negate Cond.	Negates one relational operator (single negation).	$==\rightsquigarrow !=$
Remove Cond.	Replaces a cond. branch with true or false.	`if (...)` ⇝ `if (true)`
Increments	Replace incr. with decr. and vice versa (single replacement).	$++ \rightsquigarrow --$
Return Values	Transforms the return value of a function (single replacement).	`return 0` ⇝ `return 1`

5.4. Parameter settings

In our study, we use two parameters involved in the experimental process: α and δ. In RQ1 and RQ3, we set the parameter α to 1, indicating that the influence of passed test-based method weights on the final results is fully considered. For the parameter δ, we set it to 0.85, which is the commonly recommended value for PageRank (Zhang et al., 2017; Page et al., 1998). This value ensures a good balance between the original adjacency matrix and the results obtained from random walks, leading to more reliable and accurate fault localization outcomes.

5.5. Implementation

In the experiment, we use javalang library (Partenza et al., 2021) to extract AST of each method and Gzoltar tool (Campos et al., 2012) to collect the coverage information of the program. Additionally, we generate first-order mutants by the mutation tool PIT (Laurent et al., 2017) and consider its default mutation operators. Table 5 provides 15 types of mutation operators used, along with corresponding examples.

5.6. Environment

Our experiments were conducted on a Linux server with 64G RAM, Intel(R) Xeon(R) Gold 5320 CPU @2.20 GHz, and an 80G GPU of NVIDIA A100 80 GB PCIe, running CentOS Linux release 7.9.2009 (Core).

6. Results analysis

6.1. RQ1: Effectiveness of GBSR

Tables 6, 7, and 9 present the performance of eight FL techniques across five subjects and compare their performance using the $Top\text{-}n$ (n = 1, 3, 5), MAR, and $EXAM$. The first column and the second column in these tables list the program subject names from Defects4J and different FL techniques. The third to fifth columns represent the experimental results in Top-1, Top-3, and Top-5, respectively. The last two columns illustrate the performance of different techniques on MAR and $EXAM$.

The experimental results demonstrate a significant improvement in fault localization accuracy when combining traditional fault localization techniques with GBSR. Specifically, SBFL$_G$ shows an improvement over SBFL by locating 36, 49, and 46 more faults at Top-1, Top-3, and Top-5, respectively. MBFL$_G$ outperforms MBFL by 26, 29, and 16 additional faults at Top-1, Top-3, and Top-5. Furthermore, SBFL$_G$ and MBFL$_G$ also demonstrated significant improvements over SBFL and MBFL in terms of MAR and EXAM. Overall, FL techniques combined with GBSR show substantial improvements in fault localization accuracy compared to SBFL and MBFL.

Table 6
The comparisons between SBFL$_G$ (GBSR based SBFL) and original SBFL and PRFL. The values in bold are the best result of the three techniques for the corresponding criterion, and the highlighted total results of five datasets indicate the improvement of SBFL$_G$.

Subject	Tech	Top-1	Top-3	Top-5	MAR	EXAM
Lang	SBFL	9	26	36	10.29	0.84
	PRFL	20	31	41	**5.35**	**0.47**
	SBFL$_G$	**24**	**36**	**42**	6.36	0.54
Chart	SBFL	1	3	7	72.16	0.84
	PRFL	**5**	6	**13**	44.01	0.53
	SBFL$_G$	4	**8**	12	**39.50**	**0.51**
Cli	SBFL	**2**	2	3	40.49	0.65
	PRFL	**2**	4	7	26.57	0.42
	SBFL$_G$	**2**	**5**	**8**	**24.45**	**0.38**
JxPath	SBFL	**0**	0	0	320.49	0.90
	PRFL	**0**	**1**	1	264.10	0.72
	SBFL$_G$	**0**	**1**	**2**	**239.08**	**0.64**
Math	SBFL	7	27	38	41.64	0.88
	PRFL	21	44	62	23.26	0.46
	SBFL$_G$	**25**	**57**	**66**	**20.17**	**0.40**
Total	SBFL	19	58	84	97.014	0.72
	PRFL	48	86	124	72.656	0.52
	SBFL$_G$	**55**	**107**	**130**	**65.91**	**0.50**

We performed statistical analysis on the experimental results obtained from the integration of the new combined technique with traditional SBFL and MBFL approaches. In particular, we conducted paired comparisons between the rankings of bug elements generated by techniques using the GBSR strategy and traditional techniques, at a significance level of 0.05. We calculated the p-values using the Wilson signed-rank test (Sullivan and Feinn, 2012), resulting in a p-value of $9e^{-9}$ for SBFL and SBFL$_G$, and a p-value of $3e^{-6}$ for MBFL and MBFL$_G$. These findings indicate that the improvements in fault localization achieved by the GBSR strategy are statistically significant (i.e., p-value<0.05).

Furthermore, we conducted a comprehensive analysis to explore the reasons why GBSR improves the FL accuracy of traditional FL techniques. Table 8 presents the comparison in the number of methods sharing identical suspiciousness across various techniques. Overall, we achieve the following findings: (1) Severe tie issues were observed in traditional FL techniques (SBFL and MBFL), further confirming our research motivation. (2) PRFL and PRMA, which are the improved SBFL and MBFL techniques, effectively mitigate tie issues compared to traditional FL techniques. Specifically, PRFL reduces the number of tie issues in SBFL by 56.8%, while PRMA reduces tie issues in MBFL by 53.3%. (3) GBSR demonstrates the best performance, even eliminating tie issues entirely. For example, SBFL$_G$/MBFL$_G$ achieves an improvement of 29.7%/98.1% compared to PRFL and PRMA. In particular, each method in Lang and Cli has a unique suspiciousness with MBFL$_G$.

Table 7
The comparisons between $MBFL_G$ (GBSR based MBFL) and original MBFL and PRMA. The values in bold are the best result of the three techniques for the corresponding criterion, and the highlighted total results of five datasets indicate the improvement of $MBFL_G$.

Subject	Tech	Top-1	Top-3	Top-5	MAR	EXAM
Lang	MBFL	13	43	**56**	6.30	0.55
	PRMA	**29**	46	54	5.09	0.43
	$MBFL_G$	**29**	**47**	54	**3.61**	**0.35**
Chart	MBFL	2	6	11	29.66	0.56
	PRMA	4	**11**	**14**	27.61	0.52
	$MBFL_G$	**5**	10	12	**13.59**	0.35
Cli	MBFL	2	5	11	22.77	0.36
	PRMA	**3**	6	15	16.33	0.24
	$MBFL_G$	2	**9**	**19**	**14.67**	**0.23**
JxPath	MBFL	**0**	0	0	261.86	0.73
	PRMA	**0**	1	**3**	238.25	0.64
	$MBFL_G$	**0**	**2**	2	**190.65**	**0.51**
Math	MBFL	21	44	58	31.81	0.64
	PRMA	27	54	63	25.89	0.51
	$MBFL_G$	**28**	**59**	**65**	17.09	**0.36**
Total	MBFL	38	98	136	70.48	0.57
	PRMA	63	118	149	62.63	0.47
	$MBFL_G$	**64**	**127**	**152**	**47.92**	**0.36**

Table 8
The number of entities with the same non-zero suspiciousness in GBSR and the baseline.

Subject	SBFL	PRFL	$SBFL_G$	MBFL	PRMA	$MBFL_G$
Lang	441	201	**148**	253	91	**0**
Chart	1730	758	**427**	256	84	**10**
Cli	816	**378**	397	619	140	**0**
JxPath	5122	2201	**1596**	2324	1354	**27**
Math	2459	1026	**647**	1565	674	**8**
Avg	2113.6	912.8	**643**	1003.4	468.6	**9**

These observations align with their performance in localizing faults, highlighting the effectiveness of GBSR in addressing tie issues.

Specifically, MBFL combined with GBSR addresses the tie problem more effectively than SBFL combined with GBSR. This enhanced performance is attributable to the dependency of our final suspiciousness calculations on the original results produced by traditional FL techniques. The original results generated by SBFL exhibit more severe tie issues, as certain program entities possess similar structural associations. Consequently, even after PageRank-based graph construction, it remains challenging to distinguish program entities with identical suspiciousness accurately.

Finally, a comparison between Grace and $Grace_G$ reveals that the performance of deep learning-based fault localization techniques, which have promising localization performance, can still be further improved by simply integrating our GBSR strategy.

Summary for RQ1: GBSR exhibits good performance in refining the suspiciousness of various FL techniques (improvement is 189% in SBFL and 68% in MBFL in terms of Top-1), even advanced deep learning-based techniques (improvement is 2.8% in Grace in terms of Top-1), demonstrating its effectiveness and generalizability.

6.2. RQ2: Generalization of GBSR

Table 10 shows the experimental comparison results of the original SBFL and MBFL using six different formulas combined with GBSR. The first and second column in the table represent the fault localization technique and the suspiciousness calculation formula. The third to fifth columns compare the experimental results between the original techniques and the techniques combined with GBSR in Top-1, Top-3, and Top-5. The results show that the average number of the faults localized by SBFL and MBFL on Top-1 is 22.5 and 21.33 lower than those using the GBSR. Note that among the six formulas, GBSR has a significant improvement with Dstar, Wong1, and Hamming. An analysis of the formulas reveals that its suspiciousness calculation relies on the number of failed tests, as shown in Table 1. There is a limited number of failed tests In Defects4J, which significantly reduces the accuracy of fault localization. Accordingly, GBSR that combined with more information can improve the effectiveness of fault localization.

Table 9
The comparisons between $Graec_G$ (GBSR based Grace) and original Grace. The values in bold are the better results of the two techniques for the corresponding criterion, and the highlighted total results of five datasets indicate the improvement of $Graec_G$.

Subject	Tech	Top-1	Top-3	Top-5	MAR	EXAM
Lang	Grace	34	49	**55**	3.66	0.343
	$Graec_G$	**36**	**52**	**55**	**3.58**	**0.340**
Chart	Grace	**14**	**25**	**26**	**8.25**	**0.194**
	$Graec_G$	**14**	24	**26**	8.28	**0.194**
Cli	Grace	**1**	12	**18**	16.13	0.268
	$Graec_G$	**1**	**13**	17	**16.10**	**0.267**
JxPath	Grace	5	**8**	**12**	45.10	0.123
	$Graec_G$	**6**	**8**	**12**	**43.35**	**0.119**
Math	Grace	**49**	**80**	89	**8.36**	**0.202**
	$Graec_G$	**49**	79	**90**	8.38	0.207
Total	Grace	103	174	**200**	16.30	0.226
	$Graec_G$	**106**	**176**	**200**	**15.94**	**0.225**

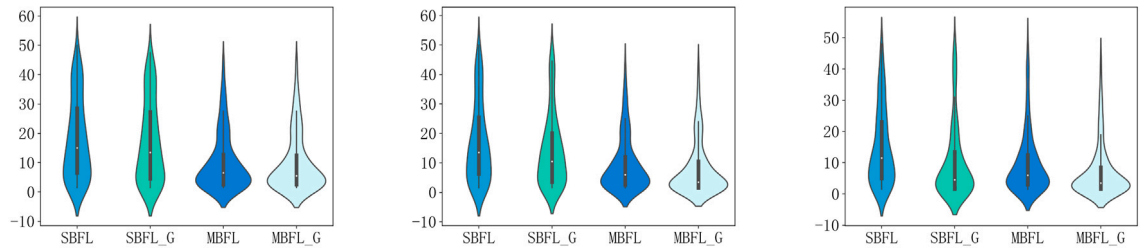
Fig. 4 represents the comparison of traditional fault localization techniques and GBSR under different formulas with respect to MAR. The x-axis represents fault localization techniques, while the y-axis is the result of MAR. The wider parts in the figure represent that a greater number of MAR results fall within that range. For example, in the Dstar formula, the experimental results obtained by the MBFL and $MBFL_G$ techniques are distributed in the range of 0 to 10. However, the results of $MBFL_G$ are concentrated in the lower value of MAR. That is to say, its ranking is more concentrated in the higher level, and the fault localization effect is better. Clearly, the technique combining GBSR in all six formulas performed better on MAR than the original SBFL and MBFL. Fig. 5 illustrates the comparison between traditional fault localization techniques and GBSR across different formulas for $EXAM$. Integrating GBSR into all six formulas yields better $EXAM$ performance compared to the original SBFL and MBFL techniques. Particularly, GBSR demonstrates more pronounced improvement when combined with different MBFL formulas than SBFL, similar to the trend observed with MAR.

Figs. 4 and 5 show that GBSR effectively improves the results of traditional fault localization for all six formulas, consistent with the experimental results shown in Table 10.

Summary for RQ2: GBSR can improve fault localization accuracy across various formulas (improvement averaged 58% in terms of Top-1), demonstrating the effectiveness of GBSR remains unaffected by the choice of formulas.

6.3. RQ3: Impact analysis

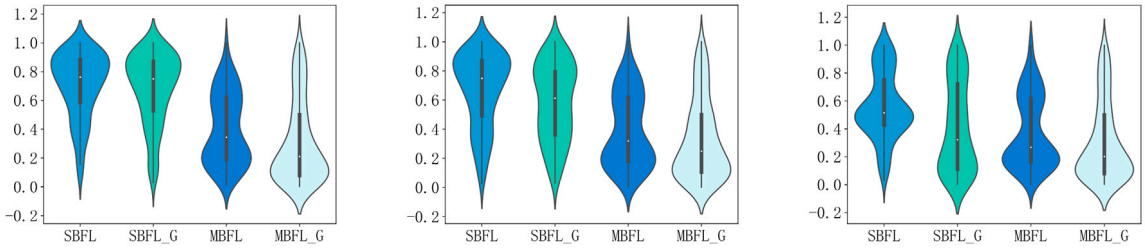
To demonstrate the effectiveness of GBSR under different parameter configurations, we analyze its impact on traditional MBFL. Fig. 6



(a) Tarantula (b) Jaccard (c) Dstar

(d) Wong1 (e) Hamming (f) Hamann

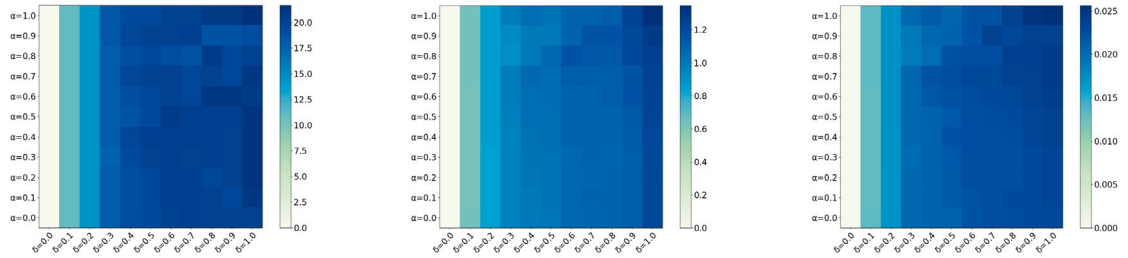
Fig. 4. Experimental results of MAR under different formulas.



(a) Tarantula (b) Jaccard (c) Dstar

(d) Wong1 (e) Hamming (f) Hamann

Fig. 5. Experimental results of $EXAM$ under different formulas.



(a) Top-1 (b) MAR (c) EXAM

Fig. 6. Experimental results of Top-1, MAR, and $EXAM$.

Table 10
Results of the GBSR in combination with different suspiciousness calculation formulas. The results covered by the green shading indicate improvements over the original techniques.

Tech	Formula	Top-1		Top-3		Top-5	
		Original	GBSR	Original	GBSR	Original	GBSR
SBFL	Tarantula	13	**16**	42	**50**	56	**60**
	Jaccard	17	17	45	**58**	60	**84**
	Dstar^2	19	**55**	58	**107**	84	**130**
	Wong1	19	**56**	58	**108**	84	**131**
	Hamming	19	**55**	58	**107**	84	**131**
	Hamann	37	**54**	93	**117**	135	**158**
	Average	20.83	**43.33**	59.0	**91.17**	84.33	**115.0**
MBFL	Tarantula	32	**38**	79	**85**	118	**120**
	Jaccard	37	**64**	85	**118**	121	**139**
	Dstar^2	38	**63**	98	**126**	136	**151**
	Wong1	38	**61**	98	**126**	136	**151**
	Hamming	38	**63**	98	**126**	136	**151**
	Hamann	39	**61**	98	**130**	117	**146**
	Average	37	**58.33**	92.67	**118.5**	127.33	**143**

presents a heatmap visualizing the enhancements achieved by integrating GBSR with MBFL compared to the original MBFL across metrics such as Top-1, MAR, and $EXAM$. Each cell in the heatmap signifies the improvement of GBSR on MBFL under the respective configuration, with darker shades indicating greater improvements. For instance, the improvement effect of GBSR on traditional MBFL becomes increasingly prominent in Top-1 with the increment of δ. However, the variation in Top-1 is not as conspicuous when considering the change in α. Across the evaluation metrics MAR and $EXAM$, the enhancement effect of GBSR increases with the increment of δ, similar to the trend observed in Top-1. Furthermore, during the process of increasing α, both MAR and $EXAM$ show some improvement in effectiveness. This clearly indicates that considering the impact of passed tests on the importance of different program entities is more beneficial than neglecting the effect of passed tests, resulting in superior localization performance.

It is worth noting that when the value of the parameter δ is zero, GBSR shows no improvement. This is because, in Eq. (3), the PageRank process follows the rules of random walks and disregards the transition matrix M when $\delta = 0$. Thus all program entities are equally weighted, and refining suspiciousness through Eq. (6) has no impact. Furthermore, we observed that under different parameter settings, combining GBSR with MBFL consistently outperformed the original MBFL in terms of suspiciousness rankings. Especially, the effect is superior when α equals 1 and δ is greater than 0.8. This demonstrates the effectiveness of GBSR in fault localization.

Summary for RQ3: Regardless of how the parameters were tuned during the calculation of program entity weights, GBSR consistently demonstrates a positive influence on the performance of fault localization.

6.4. RQ4: Time cost

Table 11 shows the comparison of time cost between GBSR and PRMA, including the construction and computation costs of the graph, as well as the mutation costs. PRMA is an improved MBFL technique, which utilizes coverage information and code results to provide additional weighted suspiciousness for MBFL to alleviate the tie issues. To ensure a fair comparison, we applied GBSR on MBFL in RQ4.

In this table, the "Subject" column shows corresponding subjects. The "Graph of PRMA" and "Graph of GBSR" columns describe the average number of vertices and edges of the graph, along with the average time of graph construction and computation per version by PRMA and GBSR, respectively. The "Mutants" column presents the average number of mutants used in the subject and the average execution time of a single mutant.

From Table 11, we observe that even for the most complex experimental subject, the graph construction time for PRMA and GBSR techniques only requires 7.93 s and 18.88 s, respectively. Moreover, during the calculation of node weights, the average computation times for PRMA and GBSR are only 0.65 and 7.01 s, respectively. Overall, the cost of generating and computing the graph is acceptable. Furthermore, PRMA and GBSR use the same set of mutants in our experiments, meaning that they have consistent mutant execution costs. Although the cost of executing mutations is relatively high, it is still can be ignored in the traditional MBFL technique.

Summary for RQ4: The computational overhead incurred by GBSR during the phases of graph construction and weight computation is considered to be acceptable or even negligible. Although the mutation execution is time-consuming, GBSR's robust performance across various FL techniques (SBFL, MBFL, and Grace) demonstrates its potential in FL.

7. Threats to validity

7.1. Internal validity

This threat is related to the choice of experimental parameters. We conducted experiments with 121 different parameter pairs using various values of α and δ, which confirmed that GBSR improves fault localization under different parameter conditions. Furthermore, the presence of bugs during the experimental process posed a significant threat. To address this issue, we implemented strict project management and sharing measures throughout the entire experiment. We repeatedly verified algorithm logic to prevent the occurrence of bugs.

7.2. Construct validity

This threat is related to the evaluation metrics selected for the experiment. For construct validity, we used different metrics widely used in fault location studies (Wu et al., 2023; de Oliveira et al., 2018; Wang et al., 2020), such as Top-n, MAR, and $EXAM$, to evaluate the effectiveness of fault location.

7.3. External validity

The threats to external validity are related to the scale of the experiment in this paper. For external validity, we evaluated the effectiveness of our experimental results extensively. Besides, we selected 5 subjects with different sizes from Defects4J, which included 243 faulty programs and 337 faults for testing. Moreover, Defects4J has been widely used in previous FL researches (Li et al., 2020a; Gong et al., 2015; Wang et al., 2022).

8. Related work

Traditional FL techniques only utilize available information (code, coverage, or killed information) in the form of binary vectors, ignoring the structural relationships between different elements. Besides, program entities that contain varying information may not be treated equitably, as not all information is equally effective in pinpointing the actual faulty location. In fact, the entity that is associated with more program entities has higher importance (Zhang et al., 2017).

Table 11
The comparison of construction and computation costs of the graph between GBSR and PRMA, as well as the mutation costs.

Subject	Graph of PRMA				Graph of GBSR				Mutants	
	#Vertexes	#Edges	Constructing time(s)	Computing time(s)	#Vertexes	#Edges	Constructing time(s)	Computing time(s)	Average number	Executing time(s)
Lang	186	3181.2	0.11	0.07	319	4115.8	3.1	0.05	133.2	13.34
Chart	1310	49 157	2.99	0.53	1834	51 567.5	18.71	16.25	525.3	7.53
Cli	523	17 593.4	0.53	0.17	729	20 061.2	0.61	0.13	206.2	11.13
JxPath	2069	261 592.2	3.14	0.31	3971	334 980.2	12.03	9.16	1902.5	2.72
Math	1641	98 982	7.93	2.18	2628	106 377.7	18.88	9.47	987.9	36.47
Avg	1145.8	86 101.16	2.94	0.65	1896.2	103 420.5	10.666	7.012	751.02	14.238

In particular, when multiple program entities have the same suspiciousness, it becomes challenging to differentiate between them and assign accurate rankings to identify the exact faulty program entities. Therefore, considering the importance of different program entities will effectively alleviate this situation. For example, Lei et al. (2022) proposed a fault localization technique based on semantic features, Feature-FL, which integrates feature diversity from the perspective of program features into the suspiciousness evaluation. Unlike Feature-FL, GBSR is a strategy of refining suspiciousness without altering the existing FL techniques, making it highly applicable to any FL technique.

Zhang et al. introduced a technique named PRFL (Zhang et al., 2017), which considers that entities associated with more suspicious program entities are more likely to be suspicious. PRFL leverages the inter-method invocation relationships and coverage information to assign different importance to program entities, thus enhancing the performance of traditional SBFL. PRFL suggests that program entities connected with more important failed tests (which cover a smaller number of program entities) may be more suspicious. Additionally, program entities connected with more suspicious program entities may also be more suspicious since they may have propagated the error states to the connected entities. However, PRFL does not consider the influence of other information on fault localization, as it is limited to the impact of internal information.

PRMA (Yan et al., 2023), introduced by Yan et al. utilizes information about test coverage of program entities, thereby improving the fault localization accuracy of traditional MBFL. In the process of calculating entity weights, PRMA does not consider the information obtained from mutation analysis. Instead, it adopts a similar approach to PRFL in constructing the information graph, leading to inadequate utilization of information. Furthermore, when adjusting suspiciousness using computed weight information, PRMA simply employs a straightforward addition approach. It fails to consider that when there are significant differences in suspicion levels among different entities, combining the calculated weights may not effectively mitigate these differences, thus resulting in suboptimal performance. In our experimental process, we considered this by first normalizing the calculated suspiciousness, thereby enhancing the versatility of the PRMA across different formulas.

Different from PRFL and PRMA, we utilize readily available and already obtained broader feature information. For instance, we consider information derived from mutation analysis (e.g., the correlation between entities and generated mutants) also influences the importance of program entities.

Furthermore, we considered the differential impact of different tests on the fault localization process, taking into account both passed and failed tests separately. We did not overlook the influence of passed tests on the importance of program entities. More importantly, our approach is not limited to a single fault localization technique or the calculation of a specific suspiciousness formula. Instead, it is a more widely applicable general technique.

9. Conclusion

Traditional fault localization techniques have been applied to solve practical problems. However, they do not perform well when dealing with program entities with similar execution information. In this work, we propose GBSR, a suspiciousness refinement strategy. In particular, it integrates multiple information, including code structure, coverage, and mutation analysis information. GBSR abstracts them into a graphical structure to make full use of the available information. Subsequently, GBSR calculates the weight of each node in the graph (i.e., the program entity) and uses the weight information to refine the suspiciousness.

We perform an extensive experimental evaluation of the GBSR strategy using 243 real software fault programs from the Defects4J. The results show that GBSR significantly improves the accuracy of fault localization. In particular, when combined with traditional SBFL and MBFL techniques, the Top-1 rankings increase by 36 and 26, respectively. Additionally, GBSR can improve the performance of learning fault localization techniques such as Grace, improving the Top-1 ranking by 2.8%. Further investigation indicates our approach is general for various FL techniques, and employing different formulas does not influence the effectiveness of GBSR.

In the future, we will incorporate higher-order mutants into our approach to address the challenges of multiple-fault localization. Moreover, we will explore time-saving and effectiveness-enhancing improvements in our research.

CRediT authorship contribution statement

Zheng Li: Writing – review & editing, Supervision, Resources, Funding acquisition. **Mingyu Li:** Writing – original draft, Software, Methodology, Formal analysis, Data curation. **Shumei Wu:** Writing – review & editing, Supervision, Methodology. **Shunqing Xu:** Software, Data curation. **Xiang Chen:** Writing – review & editing, Supervision. **Yong Liu:** Writing – review & editing, Supervision, Funding acquisition.

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

Data will be made available on request.

Acknowledgments

The work described in this paper is supported by the National Natural Science Foundation of China under Grant No. 61902015 and 61872026.

References

Abreu, R., Zoeteweij, P., 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.

An, G., Yoon, J., Yoo, S., 2021. Searching for multi-fault programs in defects4j. In: International Symposium on Search Based Software Engineering. Springer, pp. 153–158.

Basili, V., 1989. Software development: A paradigm for the future. In: [1989] Proceedings of the Thirteenth Annual International Computer Software & Applications Conference. pp. 471–485.

Briand, L.C., Labiche, Y., Liu, X., 2007. Using machine learning to support debugging with tarantula. In: The 18th IEEE International Symposium on Software Reliability. ISSRE'07, IEEE, pp. 137–146.

Campos, J., Riboira, A., Perez, A., Abreu, R., 2012. Gzoltar: An eclipse plug-in for testing and debugging. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 378–381.

Chen, M.Y., Kiciman, E., Fratkin, E., Fox, A., Brewer, E., 2002. Pinpoint: Problem determination in large, dynamic internet services. In: Proceedings International Conference on Dependable Systems and Networks. IEEE, pp. 595–604.

Chen, X., Shen, Y., Cui, Z., Ju, X., 2017. Applying feature selection to software defect prediction using multi-objective optimization. In: 2017 IEEE 41st Annual Computer Software and Applications Conference, Vol. 2. COMPSAC, IEEE, pp. 54–59.

Chen, X., Zhang, D., Zhao, Y., Cui, Z., Ni, C., 2019. Software defect number prediction: Unsupervised vs supervised methods. Inf. Softw. Technol. 106, 161–181.

Ching, W.-K., Ng, M.K., 2006. Markov chains. In: Models, Algorithms and Applications. Springer.

Dao, T., Wang, M., Meng, N., 2021. Exploring the triggering modes of spectrum-based fault localization: An industrial case. In: 2021 14th IEEE Conference on Software Testing, Verification and Validation. ICST, IEEE, pp. 406–416.

de Oliveira, A.A.L., Camilo-Junior, C.G., de Andrade Freitas, E.N., Vincenzi, A.M.R., 2018. FTMES: A failed-test-oriented mutant execution strategy for mutation-based fault localization. In: 2018 IEEE 29th International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 155–165.

Glass, R., Vessey, I., Ramesh, V., 2002. Research in software engineering: An analysis of the literature. Inf. Softw. Technol. 44 (8), 491–506.

Gong, P., Zhao, R., Li, Z., 2015. Faster mutation-based fault localization with a novel mutation execution strategy. In: 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops. ICSTW, IEEE, pp. 1–10.

Hong, S., Lee, B., Kwak, T., Jeon, Y., Ko, B., Kim, Y., Kim, M., 2015. Mutation-based fault localization for real-world multilingual programs (T). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 464–475.

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.

Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. pp. 273–282.

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. 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.

Just, R., Jalali, D., Ernst, M.D., 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 437–440.

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.

Langville, A.N., Meyer, C.D., 2004. Deeper inside pagerank. Internet Math. 1 (3), 335–380.

Laurent, T., Papadakis, M., Kintis, M., Henard, C., Le Traon, Y., Ventresque, A., 2017. Assessing and improving the mutation testing practice of pit. In: 2017 IEEE International Conference on Software Testing, Verification and Validation. ICST, IEEE, pp. 430–435.

LeCun, Y., Bengio, Y., Hinton, G., 2015. Deep learning. Nature 521 (7553), 436–444.

Lei, Y., Xie, H., Zhang, T., Yan, M., Xu, Z., Sun, C., 2022. Feature-fl: Feature-based fault localization. IEEE Trans. Reliab. 71 (1), 264–283.

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

Li, Z., Wang, H., Liu, Y., 2020a. Hmer: A hybrid mutation execution reduction approach for mutation-based fault localization. J. Syst. Softw. 168, 110661.

Li, Z., Wu, S., Liu, Y., Shen, J., Wu, Y., Zhang, Z., Chen, X., 2023. VsusFL: Variable-suspiciousness-based fault localization for novice programs. J. Syst. Softw. 205, 111822.

Li, Z., Yu, D., Wu, Y., Liu, Y., 2020b. Using fine-grained test cases for improving novice program fault localization. In: 2020 IEEE 44th Annual Computers, Software, and Applications Conference. COMPSAC, IEEE, pp. 178–183.

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

Lin, B., Cassee, N., Serebrenik, A., Bavota, G., Novielli, N., Lanza, M., 2022. Opinion mining for software development: A systematic literature review. ACM Trans. Softw. Eng. Methodol. 31 (3).

Liu, Y., Li, Z., Wang, L., Hu, Z., Zhao, R., 2017. Statement-oriented mutant reduction strategy for mutation based fault localization. In: 2017 IEEE International Conference on Software Quality, Reliability and Security. QRS, IEEE, pp. 126–137.

Liu, Y., Li, Z., Zhao, R., Gong, P., 2018. An optimal mutation execution strategy for cost reduction of mutation-based fault localization. Inform. Sci. 422, 572–596.

Lou, Y., Ghanbari, A., Li, X., Zhang, L., Zhang, H., Hao, D., Zhang, L., 2020. 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. 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: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 664–676.

Martinez, M., Durieux, T., Sommerard, R., Xuan, J., Monperrus, M., 2017. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. Empir. Softw. Eng. 22, 1936–1964.

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.

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

Page, L., Brin, S., Motwani, R., Winograd, T., 1998. The Pagerank Citation Ranking: Bring Order to the Web. Technical report, Stanford University.

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M., 2019. Mutation testing advances: An analysis and survey. In: Advances in Computers, vol. 112, Elsevier, pp. 275–378.

Papadakis, M., Le Traon, Y., 2012. Using mutants to locate "unknown" faults. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE, pp. 691–700.

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

Partenza, G., Amburgey, T., Deng, L., Dehlinger, J., Chakraborty, S., 2021. Automatic identification of vulnerable code: Investigations with an ast-based neural network. In: 2021 IEEE 45th Annual Computers, Software, and Applications Conference. COMPSAC, IEEE, pp. 1475–1482.

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, pp. 609–620.

Planning, S., 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing 1. National Institute of Standards and Technology.

Qian, J., Ju, X., Chen, X., 2023. GNet4FL: Effective fault localization via graph convolutional neural network. Autom. Softw. Eng. 30 (2), 16.

Sarhan, Q.I., Beszédes, Á., 2022. A survey of challenges in spectrum-based software fault localization. IEEE Access 10, 10618–10639.

Sasaki, Y., Higo, Y., Matsumoto, S., Kusumoto, S., 2020. SBFL-suitability: A software characteristic for fault localization. In: 2020 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 702–706.

Sohn, J., Yoo, S., 2017. Fluccs: Using code and change metrics to improve fault localization. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 273–283.

Sullivan, G.M., Feinn, R., 2012. Using effect size—or why the P value is not enough. J. Grad. Med. Educ. 4 (3), 279–282.

Wang, H., Du, B., He, J., Liu, Y., Chen, X., 2020. Ietcr: An information entropy based test case reduction strategy for mutation-based fault localization. IEEE Access 8, 124297–124310.

Wang, H., Li, Z., Liu, Y., Chen, X., Paul, D., Cai, Y., Fan, L., 2022. Can higher-order mutants improve the performance of mutation-based fault localization? IEEE Trans. Reliab. 71 (2), 1157–1173.

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.

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, 979–998.

Wu, S., Li, Z., Liu, Y., Chen, X., Li, M., 2023. GMBFL: Optimizing mutation-based fault localization via graph representation. In: 2023 IEEE International Conference on Software Maintenance and Evolution. ICSME, pp. 245–257.

Xia, F., Sun, K., Yu, S., Aziz, A., Wan, L., Pan, S., Liu, H., 2021. Graph learning: A survey. IEEE Trans. Artif. Intell. 2 (2), 109–127.

Xie, X., Chen, T.Y., Kuo, F.-C., Xu, B., 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. Softw. Eng. Methodol. 22 (4), 1–40.

Xu, X., Debroy, V., Eric Wong, W., Guo, D., 2011. Ties within fault localization rankings: Exposing and addressing the problem. Int. J. Softw. Eng. Knowl. Eng. 21 (06), 803–827.
Yan, Y., Jiang, S., Zhang, Y., Zhang, C., 2023. An effective fault localization approach based on PageRank and mutation analysis. J. Syst. Softw. 204, 111799.
Zhang, M., Li, X., Zhang, L., Khurshid, S., 2017. Boosting spectrum-based fault localization using pagerank. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 261–272.
Zhang, Y., Santelices, R., 2016. Prioritized static slicing and its application to fault localization. J. Syst. Softw. 114, 38–53.

Zheng Li is a full professor of Computer Science in the department of computer science at Beijing University of Chemical Technology. He obtained his Ph.D. from King's College London, CREST centre in 2009 under the supervision of Mark Harman. He is the author of over 80 publications, co-guest editor for 3 journal special issues and has served on 20 programme committees. He has worked on program testing, source code analysis and manipulation, and intelligent software engineering.

Mingyu Li received the B.S. degree in the computer science and technology from Beijing University of Chemical Technology, China in 2022. He is currently working toward the master'sdegree in the Beijing University of Chemical Technology, Beijing, China. His research interests include fault localization and software testing.

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

Shunqing Xu received the B.S. degree in Computer Science and Technology from the College of Modern Science and Technology, Hebei Agricultural University, Hebei, China in 2022. He is currently working toward the master's degree in the Beijing University of Chemical Technology. 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 his M.Sc., and Ph.D. degrees in computer software and theory from Nanjing University, China in 2008 and 2011 respectively. He is currently an Associate Professor at the School of Information Science and Technology, Nantong University, Nantong, China. He has authored or co-authored more than 120 papers in refereed journals or conferences (such as TSE, TOSEM, EMSE, JSS, IST, ICSE, ESEC/FSE, ASE). His research interests include software testing and maintenance, software repository mining, and empirical software engineering. He received two ACM SIGSOFT distinguished paper awards in ICSE 2021 and ICPC 2023. He is the editorial board member of Information and Software Technology.

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 30 papers in referred journals or conferences, such as Journal of Systems and Software, IEEE Transactions on Reliability, Software Testing Verification and Reliability, Information Sciences, ICSME, ASE, ISSRE, QRS, SATE, COMPSAC, etc. He is a member of CCF in China, IEEE, and ACM.