This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

An Empirical Study of Boosting Spectrum-based Fault Localization via PageRank

Mengshi Zhang, Yaoxian Li, Xia Li, Lingchao Chen, Yuqun Zhang, Lingming Zhang, Sarfraz Khurshid

Abstract—Manual debugging is notoriously tedious and time-consuming. Therefore, various automated fault localization techniques have been proposed to help with manual debugging. Among the existing fault localization techniques, spectrum-based fault localization (SBFL) is one of the most widely studied techniques due to being lightweight. The focus of the existing SBFL techniques is to consider how to differentiate program entities (i.e., one dimension in program spectra); indeed, this focus is aligned with the ultimate goal of finding the faulty lines of code. Our key insight is to enhance the existing SBFL techniques by additionally considering how to differentiate tests (i.e., the other dimension in program spectra), which, to the best of our knowledge, has not been studied in prior work. We present our basic approach, PRFL, a lightweight technique that boosts SBFL by differentiating tests using PageRank algorithm. Specifically, given the original program spectrum information, PRFL uses PageRank to *recompute* the spectrum by considering the contributions of different tests. Next, traditional SBFL techniques are applied on the recomputed spectrum to achieve more effective fault localization. On top of PRFL, we explore PRFL+ and PRFL_{MA}, two variants which extend PRFL by optimizing its components and integrating Method-level Aggregation technique, respectively. Though being simple and lightweight, PRFL has been demonstrated to outperform state-of-the-art SBFL techniques significantly (e.g., ranking 39.2% / 82.3% more real/artificial faults within Top-1 compared with the most effective traditional SBFL technique) with low overhead (e.g., around 6 minutes average extra overhead on real faults) on 395 real faults from 6 Defects4J projects and 96925 artificial (i.e., mutation) faults from 240 GitHub projects. To further validate PRFL's effectiveness, we compare PRFL with multiple recent proposed fault localization techniques (e.g., Multric, Metallaxis and MBFL-hybrid-avg), and the experimental results show that PRFL outperforms them as well. Furthermore, we study the performance of PRFL_{MA}, and the experimental results present it can locate 137 real faults (73.4% / 24.5% more compared with the most effective SBFL/PRFL technique) and 35058 artificial faults (159.6% / 28.1% more than SBFL/PRFL technique) at Top-1. At last, we study the generalizability of PRFL on another benchmark Bugs.jar, and the result shows PRFL can help locate around 30% more faults at Top 1.

Index Terms—Software testing, Automated debugging, Spectrum-based fault localization, SBFL, PageRank analysis.

1 Introduction

Software debugging is an expensive and painful process that costs developers a lot of time and effort. For example, it has been reported that debugging can take up to 80% of the total software cost [1], [2]. Thus, there is a pressing need for automated debugging techniques. In the last two decades, various fault localization approaches have been proposed to help developers locate the root causes of failures, e.g., spectrum-based [3], [4], [5], [6], [7], [8], slicing-based [9], [10], machine-learning-based [11], [12], and mutation-based [13], [14], [15], [16] techniques. The recent survey by Wong et al. [2] shows more details about various fault localization approaches.

Among the existing fault localization approaches, spectrum-based fault localization (SBFL), is one of the most widely studied fault localization techniques in the literature [6], [7], [8], [17]. Despite that SBFL is a particularly lightweight approach, it has been shown to be competitive compared to other approaches [18]. SBFL techniques take as input a set of passing and failing tests, and analyze program execution traces (spectra) of successful and failed executions. The execution traces record the program entities (such as statements, basic blocks, and methods) executed by each test. Intuitively, a program entity covered by more failing tests but less passing tests is more likely to be faulty. Hence, SBFL applies a ranking formula to compute suspiciousness scores for each entity based on the program spectra. Suspiciousness scores reflect how likely it is for each program entity to be faulty, and can be used to sort program entities. Then, developers can follow the suspiciousness rank list (from the beginning to the end of the list) to manually inspect source code to diagnose the actual root causes of failures. Recently, SBFL techniques have also been utilized by various automated program repair techniques to localize potential patch positions [19], [20], [21], [22], [23], [24].

The advantage of SBFL is quite obvious – it is an extremely lightweight approach that is scalable and applicable for large-scale programs. An ideal fault localization technique would always rank the faulty program entities at the top. However, in practice, although various SBFL techniques have been proposed (such as Jaccard/Ochiai [4], Op2 [6], and Tarantula [3]), no technique can always perform the best – the developers usually have to check various false-positive faults before finding the real one(s). We believe the current form of spectrum analysis is a key reason that limits the effectiveness of all existing SBFL techniques. Although different existing SBFL techniques use different formulas for

- *Mengshi Zhang and Sarfraz Khurshid are with the Department of Electrical and Computer Engineering, The University of Texas at Austin, USA*
- *Yaoxian Li and Yuqun Zhang are with the Department of Computer Science and Engineering, Southern University of Science and Technology, China*
- *Xia Li, Lingchao Chen and Lingming Zhang are with the Department of Computer Science, The University of Texas at Dallas, USA*
- *Corresponding author: Yuqun Zhang*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

suspiciousness computation, they all only consider how to differentiate *program entities (i.e., one dimension in program spectra)*. Our key insight is that a richer form of spectrum analysis that additionally considers how to differentiate *tests (i.e., the other dimension in program spectra)*, which, to the best of our knowledge, has not been studied in previous work, can provide more effective fault localization. For instance, consider two tests `t1` and `t2` that are both failing tests where `t1` covers 100 program entities and `t2` only covers one. By our intuition, `t2` can be much more helpful than `t1` in fault localization since `t2` has a much smaller search space to localize the fault(s). However, the traditional SBFL techniques ignore this useful information and consider `t1` and `t2` as making the same contribution on SBFL, e.g., a program entity executed by `t1` or `t2` once will be treated the same regardless of the number of entities covered by the tests.

To overcome the limitations of the existing SBFL techniques, we utilize the existing program spectra more effectively by explicitly considering the contributions of different tests. Based on our insight, we present PRFL [25], a lightweight PageRank-based technique that boosts SBFL by considering the additional test information via PageRank algorithm [26]. PRFL captures the connections between tests and program entities (e.g., the traditional spectrum) as well as the connections among program entities (e.g., the static call graphs) via bytecode instrumentation and analysis. Then, PageRank is used to recompute the program spectra: (1) program entities connected with more important failing tests (which cover smaller number of program entities) may be more suspicious, and (2) 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. Finally, PRFL employs existing SBFL ranking formulas to compute the final suspiciousness score for each program entity. We have used our PRFL prototype to localize the faulty methods for 395 real faults in Defects4J [27] benchmark. Since mutation faults have also been shown to be suitable for software testing experimentation [28], [29], to further validate the effectiveness of the proposed approach, we applied it to localize 96925 artificial mutation faults generated from 240 GitHub Java projects. The experimental results demonstrate that our technique can outperform state-of-the-art SBFL techniques significantly (e.g., ranking 39%/103% more real/artificial faults within Top-1) with negligible overhead (e.g., around 6 minutes average overhead on real faults).

To enhance our prior conference paper [25], we extend the evaluation in three aspects to further validate the effectiveness of our approach. First, we compare PRFL with a recently proposed learning-based SBFL technique (i.e., Multric [12]). Second, we compare PRFL with two recent proposed mutation-based fault localization (MBFL) techniques (i.e., Metallaxis [30], and MBFL-hybrid-avg [18]). Moreover, we study a simple integration of PRFL and MBFL, and compare its performance with PRFL and MBFL. Third, we study the performance of PRFL using three different link analysis techniques (i.e., standard PageRank [26], HITS [31] and SALSA [32]). The experimental results demonstrates that PRFL still outperforms the compared and integrated approaches.

On the other hand, to improve the efficacy of PRFL, we further extend the study and propose PRFL+. To be more specific, first, in PRFL, the static call graph analysis is applied to reflect the connections among program entities for simplicity. However, it might not be accurate to reflect the actual runtime connections. Hence, we study the effectiveness of PRFL by using dynamic call graphs to collect the run-time information. Second, PRFL originally assigns the same weight to all passing tests though their test scopes (i.e., numbers of covered program entities) are different. We extend the work by modifying the weights of passing tests to investigate their influence. Third, it is observed that some tests and methods make negligible contribution for fault localization in PRFL. To reduce the computational complexity of PRFL, we add a coverage-refinement module to remove the irrelevant methods and tests prior to PageRank analysis.

Furthermore, we propose an aggregation and PageRank-based fault localization technique, PRFL_{MA}, which applies Method-level Aggregation [33] to PRFL. The Method-level Aggregation first derives the program spectra of all statements within a method and applies SBFL formula to calculate their suspiciousness scores. Next, for each method, it chooses the most suspicious statement, and uses its coverage to substitute the coverage of given method. Finally, PRFL_{MA} executes PRFL on the updated coverage to rank all the methods. We also evaluate PRFL_{MA} on the Defects4J benchmark. The experimental results show that PRFL_{MA} successfully locates 73.4%/24.5% more real faults than state-of-the-art SBFL/PRFL techniques, and 159.6%/28.1% more artificial faults than the state-of-the-art SBFL/PRFL techniques. Moreover, to study the generalizability of PRFL, we evaluate our approach on Bugs.jar benchmark [34], and the result shows PRFL can rank 30% more faults as Top 1.

Overall, we make the following contributions [25]:

- **Novel Idea.** We propose a novel idea that considers the different contributions of different tests to further boost SBFL.
- **Lightweight Technique.** We implement the proposed idea as a lightweight fault localization technique, PRFL, that uses PageRank to consider the weights of different tests to enhance SBFL.
- **Extensions.** We study our PRFL technique by replacing static call graphs with dynamic call graphs, adjusting the modeled weights of passing tests, and reducing the size of tests and methods. Moreover, we propose PRFL_{MA} which utilizes statement-level information to improve the fault localization accuracy of PRFL.
- **Practical Tool.** We implement all the proposed techniques (PRFL and PRFL_{MA}) as a practical Maven Plugin, intelliFL, which is publicly available on Maven Central Repository, and supports Java programs with JUnit tests developed under a variety of JDK and JUnit versions.
- **Evaluation.** We evaluate our PRFL and its extensions on both real and artificial faults. First, we evaluate our approach on 395 real faults from all 6 projects in the Defects4J benchmark. To reduce the threats to external validity, we further evaluate PRFL and PRFL_{MA} on 96925 mutation faults of 240 GitHub

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

projects. The experimental results demonstrate that PRFL and PRFL_{MA} can outperform the state-of-the-art SBFL techniques significantly (e.g., ranking 39% and 73% more faults within the Top-1 on Defects4J benchmark, respectively). Moreover, we further compare PRFL to the recent proposed fault localization techniques, study the integration of PRFL and MBFL, and investigate the impact of different link analysis approach on PRFL. The experimental results show that PRFL still outperforms the compared and integrated approaches. At last, we study the generalizability of PRFL on another benchmark Bugs.jar [34], and the result shows PRFL can help locate around 30% more faults at Top 1.

2 Background

2.1 Spectrum-Based Fault Localization

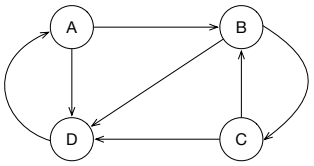
Spectrum-based fault localization techniques (SBFL) [3], [4], [6], [35] help developers identify the locations of faulty program entities (such as statements, basic blocks, and methods) based on observations of failing and passing test executions. An SBFL technique sorts all program entities by their suspiciousness scores and returns a rank list for manual checking. If a program entity is more likely to be faulty, it will be assigned a higher priority in the suspiciousness list. Therefore, an ideal SBFL technique should always rank the faulty entity with high suspiciousness score, which can significantly speed up the debugging process for finding the root causes of test failures. To compute suspiciousness scores of program entities, a SBFL technique firstly runs tests on the target program and records the program spectrum of each failing or passing test, i.e., the run-time profiles about which program entities are executed by each test. Then, based on the program spectra and test outcomes, various statistics can be extracted for suspiciousness computation, e.g., tuple (e_f, e_p, n_f, n_p), where e_f and e_p are the numbers of failing and passing tests executing the program entity e, while n_f and n_p are the numbers of failing and passing tests that do not execute e. Based on such tuples, various SBFL formulas have been proposed. The common intuition of these formulas is that a program entity executed by more failing tests and less passing tests is more likely to be faulty. This paper considers 8 well-studied SBFL techniques – Tarantula, Statistical Bug Isolation (SBI), Ochiai, Jaccard, Ochiai2, Kulczynski, Op2, and Dstar2 [3], [4], [6], [35], [36]. Tarantula, SBI, Ochiai and Jaccard [10], [12] are the most widely-used techniques for the evaluation of fault localization. Ochiai2 [6] is an extension version of Ochiai, which considers the impact of non-executed or passing test cases. Op2 [6] is the optimal SBFL technique for single-fault program, whereas Kulczynski and Dstar2 belong to the formula family Dstar [35], which is shown to be more effective than 38 other SBFL techniques. All their formulas are listed in Table 1.

2.2 Link Ranking Algorithms

Ranking plays an important role in information retrieval systems. In particular, due to the rapid growth of the World Wide Web, an efficient search engine is expected to rank and recommend web pages corresponding to users' preference to significantly save users' time and effort to find the web pages they are interested in. A straightforward example is that we can utilize word frequency to rank web pages according to their relevance to user's query. However, this approach may rank more web pages with low authority to the top since they have stronger text relevance. One solution to address this issue is to use information of the Web structure instead of text similarity only. In the recent two decades, several such link ranking algorithms were invented to efficiently extract the structural information [37], [38]. Generally, the link ranking algorithms can be classified into query independent algorithms (e.g., PageRank [26]) and query dependent algorithms (e.g., HITS [31] and SALSA [32]) that are specifically introduced as follows.

TABLE 1: Spectrum-based Fault Localization Techniques and Definitions

Tech	Defn	Tech	Defn
Tarantula	$\frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f}+\frac{e_p}{e_p+n_p}}$	SBI	$1-\frac{e_p}{e_p+e_f}$
Ochiai	$\frac{e_f}{\sqrt{(e_f+e_p)(e_f+n_f)}}$	Jaccard	$\frac{e_f}{e_f+e_p+n_f}$
Ochiai2	$\frac{e_f n_p}{\sqrt{(e_f+e_p)(n_f+n_p)(e_f+n_p)(n_f+e_p)}}$	Kulczynski	$\frac{e_f}{n_f+e_p}$
Op2	$e_f-\frac{e_p}{e_p+n_p+1}$	Dstar2	$\frac{e_f^2}{e_p+n_f}$



	A	B	C	D
A	0	0	0	1
B	0.5	0	0.5	0
C	0	0.5	0	0
D	0.5	0.5	0.5	0

Fig. 1: A Small Network and Its Transition matrix

2.2.1 PageRank

PageRank [26] is proposed by Larry Page and Sergey Brin for improving search quality and speed. PageRank views the World Wide Web as a set of linked nodes and ranks them based on their importance. The intuition behind PageRank is, for each node, if it is linked by important nodes, it should be more important than the ones linked by uninfluential nodes. Figure 1 presents a simple directed graph to describe a small network with four web pages, denoted by node A, B, C and D. The edges between two nodes denote that the starting node contains a hyperlink pointing to the ending node.

By our observation, the number of edges pointing to D is larger than others, so it should be more important than others. On the other hand, A is pointed by D and thus is also an important node according to the assumption of PageRank. Then, B is in turn pointed by A, and also should be assigned a high score to show the importance. Formally, the websites are described by a directed graph $G = \langle V, E \rangle$ with n nodes and m edges. Let $\boldsymbol{P}$ be the transition matrix of n by n elements. Then, each matrix element, P_{ij}, denotes the probability of transitioning from node j to i and its value is $\frac{1}{\text{Outbound Link Number of Node } j}$, as illustrated in Figure 1.

According to our intuition, the PageRank score of node i depends on the PageRank scores of the nodes with edges pointing to node i. Therefore, the PageRank score of node i can be computed as:

$$PR_i = \sum_{\forall j, j \to i} \frac{PR_j}{\text{Outbound Link Num of Node } j} \qquad (1)$$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

In order to make the equation more compact, we use PageRank vector $\vec{x}$ to present the PageRank score for each node and $\vec{x}$ is the solution of the eigenvalue equation:

$$\vec{x} = \boldsymbol{P} \cdot \vec{x} \tag{2}$$

In some cases, a node may have no outbound links and its PageRank score cannot be distributed to others. Considering these special nodes, an additional teleportation vector $\vec{v}$ weighted by the damping parameter d is attached to Equation(2):

$$\vec{x} = d \cdot \boldsymbol{P}\vec{x} + (1-d) \cdot \vec{v} \tag{3}$$

where $\vec{v}$ is a positive vector and $\sum v_i$ is 1. When the network scale grows, it is harder to find the exact solution for the above equation in a reasonable time. Therefore, Page et al. [26] introduced an iterative approach to get the approximate solution. The equation for the k^{th} iteration is defined as:

$$\vec{x}^{(k)} = d \cdot \boldsymbol{P}\vec{x}^{(k-1)} + (1-d) \cdot \vec{v} \tag{4}$$

and the initial value of $\vec{x}$ can be set as $\vec{v}$ or $\vec{0}$. For the example in Figure 1, if we use damping coefficient $d = 0.85$, vector $\vec{v} = [\frac{1}{n}, \frac{1}{n}, ..., \frac{1}{n}]^T$, and the initial PageRank vector $\vec{x}^{(0)} = \vec{v}$, after 25 iterations, the PageRank scores of nodes A, B, C and D are 0.3134, 0.2278, 0.1343 and 0.3246, respectively. These scores indicate the importance of each node. Recall that D is pointed by all others and becomes the most important node. A is the only node pointed by D, hence it is the second important node in the network. A and C together have two outbound links pointing to B, whose importance is lower than A. C's score is the lowest since it only has one inbound link from B.

Not only can PageRank rank web pages, but it also has been widely applied to various other domains. Gleich [39] surveyed the diversity of applications of PageRank and concluded that PageRank can be applied to Chemistry, Biology and Bioinformatics, Neuroscience, Bibliometrics, Databases and Knowledge Information Systems, Recommender Systems, Social Networks Web, i.e., twelve domains in total. Recently, PageRank-based techniques have also been proposed to analyze software systems. Chepelianskii [40] used PageRank to analyze function importance for Linux kernel. Kim et al. [41] proposed MonitorRank, a PageRank-based approach to find the root causes of anomalies in service-oriented architectures. Bhattacharya et al. [42] proposed the notion of NodeRank based on PageRank to measure the importance of nodes on a static graph for software analysis and fault prediction. Later on, Mirshokraie et al. [43] proposed the notion of FunctionRank, a dynamic variant of PageRank, for ranking functions in terms of their relative importance, for mutation testing.

2.2.2 HITS and SALSA

HITS is proposed by J.Kleinberg [31]. Different to PageRank, HITS does not only use the connections between webpages, it also considers the correlation between webpage and the topic which users provide for query. This insight implies a webpage plays two roles in the context of query. First, a webpage can be an *authority* if it can provide useful information about the topic. Second, the webpage can be a *hub* if it can provide more links to good authorities on the topic. The HITS algorithm is designed to iteratively compute the authority and hub score of each webpage.

Specifically, HITS first applies the traditional text matching technique to filter out a bunch of webpages which are irrelevant to the topic, and it gets a directed and topic-related graph G. In the next step, HITS constructs an adjacent matrix A based on G, and the iterative functions of HITS can be presented as follows:

$$\vec{a}^{(k)} = \phi(A^T\vec{h}^{(k-1)}), \qquad \vec{h}^{(k)} = \phi(A\vec{a}^{(k)}) \tag{5}$$

where $\vec{a}^{(k)}$ and $\vec{h}^{(k)}$ denote the vector of authority and hub scores in the k^{th} iteration, and $\phi(\vec{x})$ is normalization function that $\phi(\vec{x}) = \frac{\vec{x}}{\|\vec{x}\|_2}$. Typically, the initial values of $\vec{a}$ and $\vec{h}$ can be selected as:

$$\vec{a}^{(0)} = \vec{h}^{(0)} = [\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}, ..., \frac{1}{\sqrt{n}}]^T \tag{6}$$

where n is the number of nodes of G.

SALSA (Stochastic Approach for Link Structure Analysis) is proposed by Lempel and Moran [32], which integrates the authority and hub idea from HITS, and the random walk idea from PageRank. Similar to HITS, SALSA also computes the authority and hub score for each webpage, and the iterative functions of SALSA are updated as:

$$\vec{a}^{(k)} = W_c^T W_r \vec{a}^{(k-1)}, \qquad \vec{h}^{(k)} = W_r W_c^T \vec{h}^{(k-1)} \tag{7}$$

where W_c and W_r are the matrices generated from A by dividing each entry of A by the sums of its column and row, respectively.[1]

In this paper, we integrate SBFL with PageRank algorithm to boost SBFL since (1) SBFL does not utilize any text or code similarity between tests and methods for fault localization, and PageRank algorithm is query-independent as well; (2) HITS and SALSA introduce two attributes (authority and hub) of each node, but PageRank only has one attribute (importance) of node, which is more compact than HITS and SALSA. To the best of our knowledge, this work is the first to apply the PageRank algorithm for fault localization.

2.3 Method-level Aggregation

Method-level Aggregation (MA) is a technique developed by Sohn et al. [33]. Instead of using method coverage to calculate SBFL scores, MA uses coverage of all the associated statements within a method and calculates the corresponding SBFL scores for all of them. Next, for each method, the statement-level scores are ranked and the highest SBFL score is identified as the SBFL score of the associated method. MA has been demonstrated to be advanced as follows:

- **Applicability:** Method-level Aggregation is to aggregate statement coverage into method level, and thus can be applied to any SBFL technique.
- **Effectiveness:** In previous work [33], empirical evaluation of this technique was applied to existing SBFL

1. Due to space limit, we cannot introduce the details of HITS and SALSA in the paper, and we encourage interested readers to get more details in [38].

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering


```
class Code{
  static int m1(int x) {
    int y = Math.abs(x);
    if ( y % 2 == 1)
      int s = 1;//buggy
    else
      int s = 1;
    return s;
  }

  static int m2(int x) {
    int s = x + 1;
    return s;
  }
}
```

```
public void t1() {
  int a = Code.m1(-2);
  int b = Code.m2(a);
  assertEquals(2, b);
}
public void t2() {
  int a = Code.m1(2);
  assertEquals(1, a);
}
public void t3() {
  int a = Code.m1(3);
  int b = Code.m2(a);
  assertEquals(0, c);
}
public void t4() {
  int a = Code.m2(5);
  assertEquals(6, a);
}
```

Fig. 2: Example Code Snippet for Method-level Aggregation

formulas, which shows the formulas with Method-level Aggregation can rank about 42% more faults at the top.

Figure 2 shows an example code snippet to illustrate the benefits of MA. In this code snippet, `m1` is faulty, because `m1` should be a parity-check code, the output `s` will return `1` if the input is an even number, and return `-1` otherwise. We assume that there are 4 tests (`t1`-`t4`), and `t1` is failing. Therefore the spectrum tuple (e_f, e_p, n_f, n_p) of both `m1` and `m2` is $(1, 2, 0, 1)$, resulting in $Ochiai = \frac{e_f}{\sqrt{(e_f+e_p)(e_f+n_f)}} = \frac{1}{\sqrt{1(1+2)}} = 0.578$ and $OP2 = e_f - \frac{e_p}{e_p+n_p+1} = 1 - \frac{2}{2+1+1} = 0.50$. The code defect exists in `m1`, so `m1` is expected to have a higher SBFL score than `m2`. However, since `m1` and `m2` share the same spectrum, the traditional SBFL techniques assign the same score for both methods and make it impossible to differ `m1` from `m2` regarding their suspiciousness to be faulty.

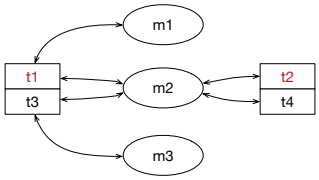
In order to improve SBFL in the similar scenarios similar to Figure 2, we adapt MA to select the statement with the highest SBFL score within each method. In general, MA utilizes the fine-grained statement-level information to help with precise method-level fault localization [33]. When applying MA to the example, we use the traditional SBFL techniques (Ochiai and Op2) to assign SBFL scores for all statements in `m1` and `m2`, and the faulty statement has the highest SBFL score (the Ochiai and Op2 scores are both 1.0). Then, according to the fine-grained statement-level analysis, the SBFL score of `m1` is now higher than that of `m2` due to the high SBFL scores for the statements within `m1`.

Note that PRFL only computes method-level fault localization, making it impossible to directly apply MA after PRFL due to the lack of statement-level fault localization results. Therefore, in this paper, instead of directly using the highest statement SBFL score as the SBFL score of the associated method, MA is adapted such that for each method the spectrum of the statement with the highest SBFL score substitutes the method's spectrum. For instance, (e_f, e_p, n_f, n_p) of `m1` becomes $(1, 0, 0, 3)$ which is the same as the faulty statement and correctly reflects the suspiciousness of program methods. In this way, the adjusted more precise method spectrum is used for more advanced PRFL.

```
class Code{
  static int m1(int
      x){
    if (x >= 0)
      return x;
    else
      return -x;
  }
  static int m2(int
      x){
    if (x > 1)//buggy
      return x;
    else
      return 0;
  }
  static int m3(int
      x){
    return x * x;
  }
}
```

```
public void t1() {
  int a = Code.m1(-2);
  int b = Code.m2(a);
  int c = Code.m3(b);
  assertEquals(0, c);
}
public void t2() {
  int a = Code.m2(5);
  assertEquals(0, a);
}
public void t3() {
  int a = Code.m2(15);
  int b = Code.m3(a);
  int c = Code.m1(b);
  assertEquals(225, c);
}
public void t4() {
  int a = Code.m2(30);
  assertEquals(30, a);
}
```

Fig. 3: Example Code and Corresponding Test Suite



$$\left[\begin{array}{ccccc} 0 & 0 & 0 & \frac{1}{3} & 0 \\ 0 & 0 & 0 & \frac{1}{3} & 1 \\ 0 & 0 & 0 & \frac{1}{3} & 0 \\ \hline 1 & \frac{1}{2} & 1 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 \end{array}\right]$$

Fig. 4: Test Coverage Graph of `Code` and the Transition Matrix of Failing Tests. (Red means the tests that failed.)

3 Motivating Example

SBFL is designed based on program execution statistics, which include both test coverage and test outcomes. Execution statistics can be treated as the information source of SBFL's analysis, so that they determine the upper bound of SBFL's accuracy. When execution statistics is built, SBFL would distribute them to construct program spectra, then apply various rank formulas to compute suspiciousness score for each program entity. Program spectrum is a practical way to present execution statistics, however, it is risky since it loses useful information during construction. Here an example will be analyzed to show how program spectra affect the accuracy of fault localization.

It can be shown from Figure 3 that in example class `Code`, method `m2` is faulty since its conditional expression should be `(x > 10)` instead of `(x > 1)`. This fault leads `t1` and `t2` to fail. Based on the spectrum information in the left half of Table 2, the traditional Tarantula technique would compute all the suspiciousness scores of `m1`, `m2` and `m3` as the same, i.e., 0.5. This result is no better than random guess, and thus it is not quite helpful for fault localization. However, when we observe the detailed test coverage shown in Figure 4, we can directly find that `m2` is faulty since `t2` fails and only covers `m2`. This example shows that different tests have different capabilities to locate faults, and one limitation of the original spectrum information is that it only focuses on computing how many failing and passing tests cover the program entities but ignores the test

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

differences. This observation inspires us that if tests can be weighted based on their capabilities in localizing potential faults, the SBFL will be more accurate.

Here we just analyze failing tests. By our intuition, the test weight should be impacted by the test capability and the covered program entities. Firstly, if the failing test covers very few program entities, this test has a very small scope and it is more capable to infer faulty entities. Therefore its weight should be high. On the other side, if its covered entities are more likely to be faulty, it in turn also should get a higher weight. Similarly, if a program entity is covered by more highly weighted tests, it should also be more likely to contain faults.

All the analysis above is constructed on the bi-directional test coverage graph, which is a kind of network. Hence test weight analysis can be solved by PageRank. Note that the failing and passing tests cover different set of entities, PageRank analysis will be executed twice to generate the scores of entity importance for failing and passing tests. We term these scores *faultiness* and *successfulness* scores, respectively. For example, when computing the failing test weights, we use vector $\vec{x} = [m_1, m_2, m_3, t_1, t_2]^T$ to present the node values where m_1, m_2 and m_3 present the faultiness scores of `m1`, `m2` and `m3` and t_1, t_2 show the test weights of `t1` and `t2`. The test capabilities can be presented by teleportation vector $\vec{v} = [0, 0, 0, w_1, w_2]^T$, where the first three 0s denote the three corresponding source methods and w_1 and w_2 are the weights of `t1` and `t2`. They can be computed by $w_i = \frac{c_i^{-1}}{\sum c_j^{-1}}$, where c_i is defined as the number of program entities covered by the ith test. For this example, c_1 and c_2 are 3 and 1 respectively and $\vec{v}$ is $[0, 0, 0, 0.25, 0.75]^T$. The construction of transition matrix has been introduced in Section 2.2 and the matrix $\boldsymbol{P}$ can be found in Figure 4. Assume that the damping factor d is 0.7, $\vec{x}^{(0)}$ is $\vec{0}$, based on Equation(4), we can get $\vec{x}$ as $[0.061, 0.290, 0.061, 0.262, 0.326]^T$, where m_2 is larger than m_1 and m_3, indicating that `m2` is highly connected with failed tests and thus is more likely to be faulty, while w_1 is less than w_2, indicating that `t2` is more effective to help with fault localization. This result reflects that PageRank analysis computes faultiness score for each method and distributes weights for tests in tandem. In the next step, we only utilize faultiness scores to construct weighted spectra since they already include the information from test weights. The weighted spectra can be computed by the normalized faultiness score $\hat{m}_i = \frac{m_i}{max(\{m_j\})}$. In this example, only failing tests are considered, so only e_{fi} and n_{fi} need to be updated as $\hat{e}_{fi} = \hat{m}_i \cdot N_f$ and $\hat{n}_{fi} = N_f - \hat{e}_{fi}$, where N_f is the total number of failing tests. The passing tests can be analyzed in the similar way, whose details can be found in Section 4.1.2. The right half of Table 2 shows the weighted spectrum information and updated Tarantula scores for each `Code` method. According to the table, PRFL boosts Tarantula to rank `m2` as the first, demonstrating the effectiveness of PageRank for fault localization.

Actually, although PRFL can help with Tarantula with the above example, some other formulas, e.g., Ochiai, actually can also rank the faulty method precisely. Therefore, we further show another example. Suppose method `m1` is also faulty by changing Line 3 to `if (x` $\geq$ `5)`, and also `t4` is

TABLE 2: Original and Weighted Spectra of `Code`. (T denotes Tarantula score)

Program Entity	Original Spectrum Info					Weighted Spectrum Info				
	e_f	e_p	n_f	n_p	T	e_f	e_p	n_f	n_p	T
m1	1	1	1	1	0.5	0.42	1	1.58	1	0.30
m2	2	2	0	0	0.5	2	2	0	0	**0.50**
m3	1	1	1	1	0.5	0.42	1	1.58	1	0.30

modified as:

```
public void t4() {
    int a = Code.m1(4) + 5;
    int b = Code.m3(a);
    assertEquals(81, b);
}
```

Since now both `m1` and `m2` are faulty, tests `t1`, `t2` and `t4` fail and only `t3` pass. This result leads all three methods to share the same traditional spectrum information, i.e., (e_f, e_p, n_f, n_p)=$(2, 1, 1, 0)$ – no matter which SBFL formula is applied, all the methods will be ranked with the same suspiciousness. However, as we analyzed before, `m2` is clearly a fault since it is the only method covered by the failed `t2`. Based on our PRFL idea, we can get vector $\vec{x}$ as $[m_1, m_2, m_3, t_1, t_2, t_4]^T$ = $[0.099, 0.213, 0.099, 0.198, 0.238, 0.151]^T$. This outcome shows that firstly, `m2` is more likely to be faulty than `m1` and `m3` since 0.213 is greater 0.099. Secondly, as we expected, `m1` and `m3` has the same score(0.099) and the reason is that both of them are covered by the same tests. Thirdly, `t2` is the most important (e.g., with the highest weight 0.238) test among `t1`, `t2` and `t4` since it only covers `m2`. Moreover, `t1` is more important than `t4` since it covers more faulty methods. This multiple-fault example further demonstrates the effectiveness of PageRank for fault localization.

4 Approach

In this section, we illustrate the approach of our study. An overall framework of our approach is depicted in Figure 5. Specifically, in Section 4.1, we propose our basic approach, PRFL, which consists of three major phases: *preparation*, *PageRank analysis* and *ranking*. In the *preparation* phase (Section 4.1.1), PRFL first collects execution traces and test results from the faulty program by running tests and further applies static analysis to construct the connections between program methods, i.e., the call graph. In the *PageRank analysis* phase (Section 4.1.2), PRFL constructs the connection matrix between tests and methods as well as among source methods, and generates the weighted spectrum information for each tests. In the final *ranking* phase (Section 4.1.3), PRFL takes as input the updated weighted spectra and uses existing ranking formulas to localize faulty methods.

In Section 4.2, we propose PRFL+, which further enhances PRFL on three major parts: *call graph refinement*, *passing test weight adjustment* as well as *test and method reduction*. Considering that not all methods in static call graphs are executed during runtime, in the *call graph refinement* part (Section 4.2.1), we use dynamic rather than static call graphs to collect the runtime connections of methods. In the *passing test weight adjustment* part (Section 4.2.2), we take the differences of the capabilities of passing tests into consideration by adjusting the their weights prior to PageRank analysis. In the *test and method reduction* part (Section 4.2.3), since some methods and tests make negligible impacts on the PageRank

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

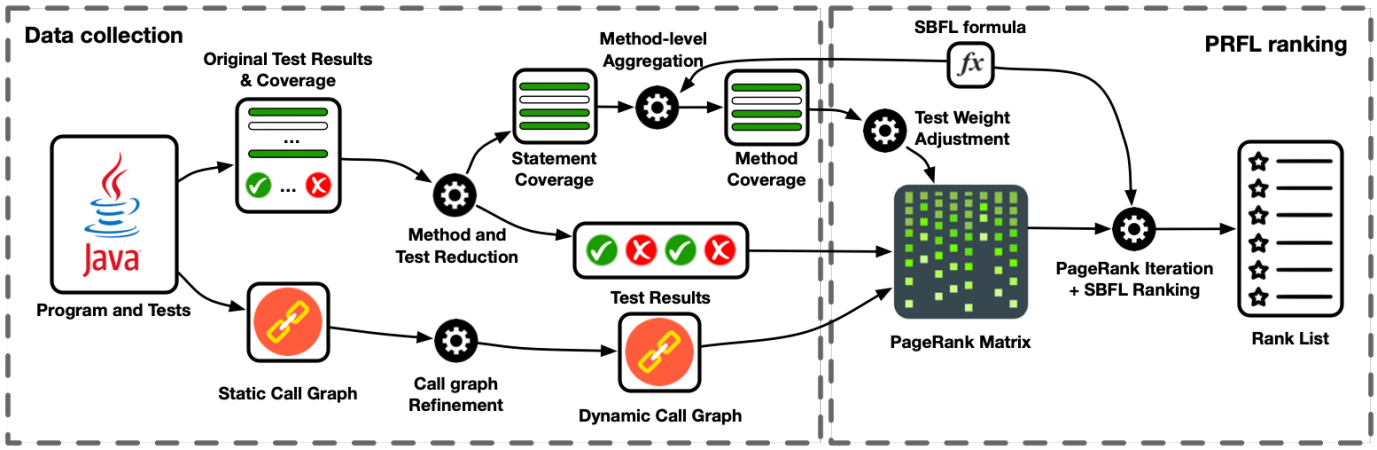


Fig. 5: Framework of the Proposed Approach

analysis, we propose two approaches to identify and remove such methods and tests to make PRFL more efficient.

Furthermore, in Section 4.3, we propose an aggregation and PageRank-based fault localization technique, namely PRFL_{MA}, which adapts Method-level Aggregation (MA) [33] to utilize the detailed statement coverage information in order to boost PRFL. Specifically, instead of applying SBFL scores directly, PRFL_{MA} firstly identifies the statement with the highest suspiciousness score within each method, then uses its coverage as the method coverage for PRFL. Finally, in Section 4.4, we analyze the time and space complexity of PRFL theoretically.

4.1 Basic Approach: PRFL

4.1.1 Preparation Phase: Static and Dynamic Analysis

The *preparation* phase in PRFL is designed to collect graph data from both dynamic execution and source code for the PageRank analysis. In this phase, PRFL constructs test coverage graph via dynamic analysis. When each test is running, the dynamic analysis performs code instrumentation to automatically record which methods are executed by the test. Note that different methods may be covered by the same failing and passing tests, which means they cannot be differentiated only by the coverage graph; hence, more information should be mined to alleviate this issue. Our heuristic is that if two methods have the same test coverage (e.g., the same method and test connections), the connections between the tied methods and other methods can help break the tie (e.g., a method connected with more fault-prone methods may also be fault-prone since it may have propagated the error states to the connected methods). Therefore, for each method, PRFL further applies static analysis to extract the static call graph to obtain the connections among source methods. The constructed call graph will then be combined with test coverage for the PageRank analysis.

4.1.2 Analysis Phase: PageRank Propagation

Here we first recall the original PageRank equation (Equation(3)). The PageRank equation consists of three elements, i.e., transition matrix $\mathbf{P}$, damping factor d, and teleportation vector $\vec{v}$. To be specific, damping factor d and teleportation vector $\vec{v}$ are parameters and transition matrix $\mathbf{P}$ is constructed based on coverage and call graph.

Transition Matrix Construction

As we have illustrated in Section 3, given the test coverage information (i.e., the connections between tests and methods), the transition matrices can be partitioned as:

$$\boldsymbol{P} = \left[\begin{array}{c|c} \boldsymbol{0} & \boldsymbol{P}_{TM} \\ \hline \boldsymbol{P}_{MT} & \boldsymbol{0} \end{array}\right] \tag{8}$$

where $\boldsymbol{P}_{MT}$ and $\boldsymbol{P}_{TM}$ denote the transition matrix between methods and tests (based on the test coverage graph). Note that since the test coverage graph is bipartite, the sub-matrices on the top left and bottom right are zero-matrices.

The matrix $\boldsymbol{P}$ can present the differences between methods based on test coverage; however, some methods may be covered by the same failing and passing tests and they cannot be differentiated only by $\boldsymbol{P}$. To alleviate this issue, we utilize the Class Hierarchy Analysis (CHA) call graph algorithm [44] to further distinguish these methods. The transition matrix of call graph is constructed in a different way. Firstly, we convert the call graph to adjacent matrix $\mathbf{A}$ as follows: suppose method m_i invokes method m_j, one edge would be added from m_i to m_j with weight 1. Also, m_j will return to m_i when finished and this return relation should also be considered. Heuristically, the calling edges may be more important than the return edges, thus we assign smaller weight, δ, on the return edge from m_j to m_i, i.e., A_{ij} and A_{ji} can be computed as $A_{ij} = 1$ and $A_{ji} = \delta$. Note that when m_j and m_i invoke each other (e.g., due to recursion), both A_{ij} and A_{ji} are equal to $1 + \delta$. Finally, $\mathbf{A}$ will be column-normalized to $\hat{\boldsymbol{A}}$ by:

$$\hat{A}_{ij} = \frac{A_{ij}}{\sum_k A_{ik}} \tag{9}$$

and the method-to-method matrix (i.e., the transition matrix of call graph), $\boldsymbol{P}_{MM}$, can be computed as $\boldsymbol{P}_{MM} = \hat{\boldsymbol{A}}^T$.

Teleportation Vector Design

As we have discussed before, test weight is not only impacted by the faultiness or successfulness likelihood of covered methods, but also by test capabilities. Hence, PRFL uses different teleportation vectors to present the impact of test capabilities. Teleportation vector $\vec{v}$ is composed by two sub-vectors: $\vec{v} = \left[\vec{v}_m^T, \vec{v}_t^T\right]^T$, where $\vec{v}_m$ and $\vec{v}_t$ denote the teleportation vector of methods and tests respectively. For both failing and passing tests, $\vec{v}_m$ is $\vec{0}$. For failing tests, $\vec{v}_t$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

is $[w_1, w_2, ..., w_m]^T$, where $w_i = \frac{c_i^{-1}}{\sum_j c_j^{-1}}$ and c_i denotes the number of methods covered by test i. This setting is based on the property that if a test fails, it covers at least one faulty method so that a failing test with smaller scope is more capable to locate the faults. However, a passing test does not satisfy this property since it may cover a faulty method whose fault is not triggered. Therefore, the capability of a passing test does not depend on the number of covered methods and all test capabilities share the same weight. Due to the normalization of weight w_i, $\vec{v}_t$ for passing tests is defined as $[\frac{1}{t}, \frac{1}{t}, ..., \frac{1}{t}]^T$, where t denotes the number of the passing tests.

Constrained PageRank Algorithm

In this work, we extend the standard PageRank algorithm [26] to analyze the integration of test coverage and call graph. The standard PageRank algorithm is applied to a graph whose nodes and edges are all congeneric. However, in our application scenario, this prerequisite is not satisfied since the edges in the test coverage graph present the connections between methods and tests, which are not congeneric with the ones in call graph. Thus, the standard PageRank algorithm may be underperformed because of edge variance.

To overcome this issue, we propose a constrained PageRank algorithm to differentiate the edges based on their connection types. Our intuition is that the test coverage graph makes the main contribution for fault localization, so the edges in test coverage graph deserve more weights; on the other hand, the edges in call graph should be set less weights. We use parameter α to tune the weight of call graph, and the constrained transition matrix is:

$$\boldsymbol{P} = \left[\begin{array}{c|c} \alpha \boldsymbol{P_{MM}} & \boldsymbol{P_{TM}} \\ \hline \boldsymbol{P_{MT}} & \boldsymbol{0} \end{array}\right] \tag{10}$$

The PageRank vector $\vec{x}$ in Equation(3) can be decomposed as $\vec{x} = \left[\vec{x}_m^T, \vec{x}_t^T\right]^T$, where $\vec{x}_m$ denotes the method faultiness or successfulness scores and and $\vec{x}_t$ denotes the weights of tests. Then, the iterative Equation(3) can be updated as:

$$\vec{y}_m^{(k)} = d \cdot (\alpha \boldsymbol{P}_{MM} \cdot \vec{x}_m^{(k)} + \boldsymbol{P}_{TM} \cdot \vec{x}_t^{(k)}) \tag{11}$$

$$\vec{y}_t^{(k)} = d \cdot \boldsymbol{P}_{MT} \cdot \vec{x}_m^{(k)} + (1-d) \cdot \vec{v}_t \tag{12}$$

$$\vec{x}_m^{(k+1)} = \frac{\vec{y}_m^{(k)}}{max(\vec{y}_m^{(k)})} \tag{13}$$

$$\vec{x}_t^{(k+1)} = \frac{\vec{y}_t^{(k)}}{max(\vec{y}_t^{(k)})} \tag{14}$$

where the both initial settings of $\vec{x}_m$ and $\vec{x}_t$ are $\vec{0}$.

4.1.3 Ranking Phase: Weighted-Spectrum-based Fault Localization

The weighted spectrum can be constructed using faultiness, successfulness score and failing, passing test number. In the second phase , the *PageRank Analysis* is executed twice to generate vector $\vec{x}_{mf}$ and $\vec{x}_{ms}$, which consist of faultiness and successfulness scores of all covered methods. For each method m_i, its faultiness score s_{fi} and successfulness score s_{si} are extracted from $\vec{x}_{mf}$ and $\vec{x}_{ms}$ respectively, and the weighted spectrum can be computed as:

$$\begin{array}{ll} \hat{e}_{fi} = s_{fi} \cdot N_f, & \hat{n}_{fi} = N_f - \hat{e}_{fi} \\ \hat{e}_{pi} = s_{si} \cdot N_p, & \hat{n}_{pi} = N_p - \hat{e}_{pi} \end{array} \tag{15}$$

where N_f and N_p denote the total number of failing and passing tests, respectively. PRFL then applies SBFL formulas to compute suspiciousness scores on weighted spectra and ranks all method. The weighted spectra include information not only from test coverage, but also from test capabilities and call graph information. Therefore, the weighted spectra are more accurate to reflect the method faultiness and successfulness, and can consequently boost the effectiveness of SBFL techniques.

4.2 Extended Approach 1: PRFL+

4.2.1 PRFL+ Call Graph Refinement

Generally, call graph represents the method invocation relations in a given project, which can help developers efficiently understand the code structure. In the literature, various static analysis techniques [44], [45], [46] have been proposed to capture static call graphs for Java programs to obtain the connections among source code methods.

In our previous work [25], we analyzed source code and directly applied static call graph analysis to build the connections of program entities (i.e. methods). However, this setting could count more connections than necessary since some methods in static call graphs might not be executed during test execution. In this work, we explore the efficacy of PRFL by replacing static call graphs with dynamic call graphs traced during runtime for PageRank analysis to study the impact of call graph types.

4.2.2 PRFL+ Passing Test Weight Adjustment

The original PRFL only differentiates failing tests by the number of their covered methods to explore the effects on fault localization, while the effects of differentiated passing tests are yet to be studied. In particular, recall that in section 4.1.2, we design the teleportation vectors to present the impact of test capabilities in PRFL. Specifically, the teleportation vector $\vec{v} = \begin{bmatrix} \vec{v}_m^T & \vec{v}_t^T \end{bmatrix}^T$, where $\vec{v}_m$ and $\vec{v}_t$ represent the teleportation vectors of methods and tests respectively. For passing tests, $\vec{v}_m$ is designed to be $\vec{0}$, and the weight of $\vec{v}_t$ is defined as $[\frac{1}{t}, \frac{1}{t}, ..., \frac{1}{t}]^T$ after normalization, where t denotes the number of the passing tests. As a result, all passing test capabilities are assigned with the same teleportation weights based on this assumption that the capability of a passing test is not influenced by the number of its covered methods.

In this section, we study the effects of adjusting the teleportation weights $\vec{v}_t$ for the passing tests. In Section 4.1.2, we design the teleportation weights of failing tests based on the property that a failing test covering very few program entities will have a smaller scope, which is more helpful to infer faulty entities. Here, we make similar assumption that the teleportation weights of passing tests should follow the same principle: if the passing test covers very few

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

program entities, these entities would more likely be fault-free, and this indicates the passing test is more important and should be assigned a higher weight. The reason is that such passing tests would have failed with high probability if the element is faulty since it is easy for any fault within the covered element to propagate to the test output due to the small number of covered elements. Therefore, we design the teleportation weights of passing tests $\vec{v}_t$ as $[w_1, w_2, ..., w_m]^T$, where $w_i = \frac{c_i^{-1}}{\sum c_j^{-1}}$ and c_i denotes the number of methods covered by test i, which are the same as failing tests.

4.2.3 PRFL+ Test and Method Reduction

Recall that in the *PageRank analysis* phase, the faultiness and successfulness scores of all suspicious methods (i.e., the methods covered by failing test cases) and their spectra are computed based on the construction of the method-to-test transition matrix $\boldsymbol{P}_{MT}$ and test-to-method transition matrix $\boldsymbol{P}_{TM}$, whose computational costs strongly depend on the matrix dimensions. Furthermore, note that not all tests and methods contribute to computing their corresponding teleportation vectors and thus impact negligibly on modeling PRFL. These tests and methods are listed as follows:

- Methods: which are covered only by passing tests
- Tests: which cover the methods that are not covered by any failing tests

Accordingly, we propose two approaches: test-based reduction and method-based reduction, to identify such tests and methods and remove them prior to modeling PRFL for the purpose of reducing its computational cost.

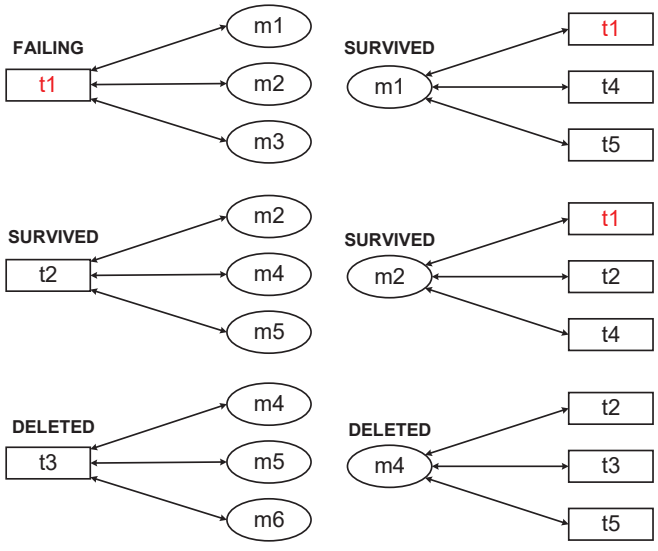


Fig. 6: Test-based Reduction (a) and Method-based Reduction (b)

Test-based Reduction

The test-based reduction approach removes the elements in $\boldsymbol{P}_{TM}$ and $\boldsymbol{P}_{MT}$ when the corresponding tests contain no failing-test-covered methods. For instance, in Figure 6a, `t2` and `t3` are passing tests. `t1` is a failing test (in red) that covers the methods `m1`, `m2`, and `m3`. Since the passing test `t2` also covers `m2` that is covered by `t1`, the test-based reduction approach retains all the `t2`-related elements (`t2`–`m2`, `t2`–`m4`, `t2`–`m5`). On the other hand, the passing test `t3` covers `m4`, `m5`, and `m6`, and none of which is covered by `t1`. Therefore, all the `t3`-related elements (`t3`–`m4`, `t3`–`m5`, `t3`–`m6`) would be deleted from $\boldsymbol{P}_{TM}$, and dually, `m4`–`t3`, `m5`–`t3` and `m6`–`t3` would also be deleted from $\boldsymbol{P}_{MT}$.

Method-based Reduction

Similarly, the method-based reduction approach removes the elements in $\boldsymbol{P}_{MT}$ and $\boldsymbol{P}_{TM}$ when the corresponding methods are only covered by passing tests. For instance, in Figure 6b, the test `t1` is failed (in red) and the tests `t2`-`t5` are passed. According to the method-based reduction approach, since `m1` and `m2` are covered by the failing test `t1`, both of them will be retained for PRFL analysis. On the other side, since `m4` is covered by all the passing tests `t2`, `t3`, and `t5`, all the `m4`-related elements (`m4`-`t2`, `m4`-`t3`, `m4`-`t5`) will be deleted from $\boldsymbol{P}_{MT}$, and dually, `t2`-`m4`, `t3`-`m4`, `t5`-`m4` will also be deleted from $\boldsymbol{P}_{TM}$.

4.3 Extended Approach 2: PRFL$_{MA}$

In Section 2.3, it is observed that the traditional SBFL techniques cannot differentiate the faultiness of `m1` and `m2` only using method coverage. In order to improve the accuracy of fault localization, we further propose PRFL$_{MA}$, a novel technique which integrates Method-level Aggregation and PRFL.

In general, PRFL$_{MA}$ includes the following steps:

- Step 1: PRFL$_{MA}$ extracts the spectrum (e_f, e_p, n_f, n_p) for all statements and applies SBFL formula (e.g., Ochiai2) to compute the suspiciousness scores of all statements.
- Step 2: For each method, PRFL$_{MA}$ chooses the most suspicious statement covered by the given method, and uses the statement coverage to replace the method coverage.
- Step 3: PRFL$_{MA}$ executes PRFL on the updated coverage to rank all the methods.

TABLE 3: The Process of PRFL$_{MA}$

C	Subject	t1	t2	t3	t4	Och.2	Agg.
m1	$static\ int\ m1(int\ x)\ \{$	1	1	1	0	0.08	1*
s1	$y\ = Math.abs(x);$	1	1	1	0	0.08	0.08
s2	$if(y\ \%\ 2 == 1)$	1	0	0	0	1*	1*
s3	$int\ s = 1; //buggy$	1	0	0	0	1*	1*
s4	$else$	0	1	1	0	0	0
s5	$int\ s = y;$	0	1	1	0	0	0
s6	$return\ s;\}$	1	1	1	0	0.08	0.08
m2	$static\ int\ m2(int\ x)\ \{$	1	0	1	1	0.08	0.08
s7	$int\ s = x + 1;$	1	0	1	1	0.08	0.08
s8	$return\ s;\}$	1	0	1	1	0.08	0.08
Test case outcome		f	p	p	p		
Number of candidates		4				m1/2	m1

We use Table 3 as an example to illustrate the process of PRFL$_{MA}$. Columns `t1`–`t4` demonstrate the coverage information of four test cases on `m1`, `m2`, and their statements. Specifically, `1` and `0` represent the method/statement being covered and uncovered by each test, respectively. For instance, `m1` is covered by `t1`, `t2`, and `t3`. In addition, line `s4` in `m1`, "else{" is covered by `t2` and `t3`. Column `Och.2` presents the results of traditional Ochiai2 scores of the two methods and their statements. It can be shown

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

that `m1` and `m2` are assigned with the same SBFL scores and hence cannot be properly ranked because they have the same spectra. On the other hand, the SBFL scores of statements are different. The reason is that some methods are partially executed by tests, leading to different SBFL scores among the statements. Column `Agg.` shows that the statements `s2` and `s3` are assigned with the highest SBFL score within `m1` after applying Ochiai2. Next the spectrum and SBFL scores of `s2` or `s3` are assigned to `m1`. In other words, originally, `m1` is executed by the tests `t1`–`t3`, and `s2` and `s3` are executed by `t1`. After applying MA, `m1` is perceived to be executed by `t1`, leading to a Method-level Aggregation of the statement-level spectra. As a result, `m1` is ranked higher than `m2`.

4.4 Computational Complexity

Suppose the numbers of failing and passing tests are T_f and T_p, and the numbers of their covered methods are M_f and M_p respectively. In general, the program quality is relatively high and only a few of tests are failed, hence, we can make a mild assumption that $T_p \gg T_f$ and $M_p \gg M_f$. Therefore, the complexity of our approaches is derived mostly based on the complexity of the passing tests. Recall that in the Equation 11 and 12, vector $\vec{x}_m$ and $\vec{x}_t$ denote the PageRank scores of methods and tests, and their sizes are M_p and T_p. The dimensions of matrix P_{MM}, P_{TM} and P_{MT} are $M_P \cdot M_P$, $M_P \cdot T_P$ and $T_P \cdot M_P$, respectively. Hence, the time complexity of computing Equation 11 and 12 depends on the complexity of the multiplication of matrix and vector. Specifically, the time complexity of Equation 11 is $O(M_P^2 + M_P \cdot T_P)$ and the time complexity of Equation 12 is $O(M_P \cdot T_P)$. Therefore, the total time complexity of PRFL is $O(M_P^2 + M_P \cdot T_P)$. On the other hand, compared to SBFL, we need more space to store these matrices, therefore, the space complexity is $O(M_P^2 + M_P \cdot T_P)$ as well.

5 Experimental Setup

Our experimental study aims to answer the following research questions:

- **RQ1:** How does PRFL compare with traditional SBFL techniques in term of effectiveness and efficiency?
- **RQ2:** How do different configurations impact the effectiveness of PRFL?
- **RQ3:** How do different numbers of faults impact the effectiveness of PRFL?
- **RQ4:** How does PRFL compare with the recent proposed learning and spectrum-based fault localization technique?
- **RQ5:** How does PRFL compare with the recent proposed mutation-based fault localization techniques?
- **RQ6:** How does PRFL perform using different link analysis algorithms?
- **RQ7:** How does PRFL+ perform on localizing real faults?
- **RQ8:** How does PRFL_{MA} perform on localizing real faults?
- **RQ9:** How do PRFL and PRFL_{MA} perform on a large number of artificial faults?
- **RQ10:** How statistically significant is the improvement of the proposed techniques?

TABLE 4: Subject Statistics

ID	Program	#Faults	LoC	#Tests
Chart	JFreeChart	26	96K	2,205
Closure	Closure Compiler	133	90K	7,927
Lang	Commons Lang	65	22K	2,245
Math	Commons Math	106	85K	3,602
Time	Joda-Time	27	28K	4,130
Mockito	Mockito	38	23k	1,366
Real-Fault Total	6 Projects	395	344k	21,475
Mutation-Fault Total	240 Projects	96925	2565K	19556

5.1 Subjects

Real-Fault Subjects We use Defects4J [27] and Bugs.jar [34] as the benchmarks for our evaluations.

Defects4J [27] is a mature real fault dataset for testing experiments, and has been widely used in software testing research [18], [29], [47], [48], [49]. Defects4J(v1.2.0, update in December 2017) includes 395 real faults from 6 open-source projects: JFreeChart, Google Closure Compiler, Apache Commons Lang, Apache Commons Math, Joda-Time and Mockito. For each fault, Defects4J provides the faulty program, the fixed program with minimum code change and the failing tests. We identify the faulty methods in the following ways. Firstly, we compare the modified source files to collect code changes. If all changes are located in a single method, we label such method as a faulty method. However, in some other cases, the program changes are distributed in multiple methods which may not all be faulty. To precisely identify the actual fault-triggering methods, we then manually apply all the possible combinations of the modified methods to get the minimum change set that can pass all tests. Note that we used all the 395 Defects4J faults except the faults not within method bodies. Table 4 (except the last row) shows the statistics of the Defects4J subjects – Column 1 presents the subject IDs that will be used in the remaining text; Column 2 presents the full name of the subjects; Column 3 presents the number of faults for each subject; finally, Columns 4 and 5 present the LoC (i.e., Lines of Code) and test number information of the most recent version of each subject in Defects4J.

Bugs.jar [34] is a recently proposed real fault dataset, which includes 1158 faults and patches extracted from 8 open-source Java projects: Accumulo, Camel, Apache Commons Math, Flink, Jackrabbit Oak, Log4J2 Logging, Maven, Wicket. In our experiments, we successfully built and applied 560 out of 1158 bugs (Accumulo 56/98, Apache Commons Math 89/147, Flink 69/70, Jackrabbit Oak 238/278, Log4J2 Logging 65/81, Wicket 43/289) to evaluate our approach.

Artificial-Fault Dataset Although Defects4J and Bugs.jar are suitable for evaluating testing techniques, their projects and faults are rather limited, posing threats to validity. Meanwhile, mutation faults have been shown to be suitable for software testing experimentation [28], [29]. Therefore, we further use the PIT mutation testing tool [50] (with all its 16 supported mutation operators) to generate artificial faults for evaluating PRFL. To be specific, we start from the first 3804 most popular Java projects from GitHub [51]. 1855 projects of those were built successfully with Maven and passed all tests. Then, 786 projects were further removed since PIT crashed or could not terminate within our time limit, i.e., 2 hours. Therefore, finally, we have 240 projects with mutation faults, ranging from 112 to 313016 lines of

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

code. The last row of Table 4 presents the statistics of the mutation faults used in our study.

5.2 Implementation and Supporting Tools/Platform

Data Preparation We use ASM bytecode analysis framework [52] together with JavaAgent [53] to perform on-the-fly code instrumentation to capture the test coverage for each test. Furthermore, we also implement the static Class Hierarchy Analysis (CHA) call graph algorithm [44] based on ASM framework. Note that we ignore all the 3rd party libraries and Java internal libraries during the call graph analysis for time efficiency.

Data Analysis We use Numpy [54], one of the most popular scientific computing package in Python, to implement and evaluate PRFL and other traditional SBFL techniques. PRFL applies an iterative algorithm to compute the faultiness and successfulness scores. For our application scenario, the test number is relative small, which leads the sizes of transition matrices $\boldsymbol{P}_{TM}$ and $\boldsymbol{P}_{MT}$ limited. On the other hand, since the method invocations are not frequent, the transition matrix $\boldsymbol{P}_{MM}$ is sparse. This property makes PRFL execute fast, and for all following experiments, PRFL is iterated 25 times for both failing and passing tests.

Tool Supports Currently all the proposed techniques (including PRFL and PRFL_{MA}) have been implemented and integrated in our intelliFL tool[2]. intelliFL has been implemented as a practical Maven Plugin for debugging Java programs with JUnit tests, and now is also publicly available in Maven Central Repository [55]. intelliFL currently supports both single-module and multi-module Maven projects, as well as unit (via Maven Surefire Plugin [56]) and integration (via Maven Failsafe Plugin [57]) tests. Furthermore, intelliFL supports tests developed under both JUnit 3 and 4, and can also support source code developed under JDK versions 7 to 9.

Platform All our experiments were performed on an Intel(R) Xeon E5-4610 v4 CPU(1.80 GHz) with Ubuntu Linux 16.04.

5.3 Evaluation Metrics

We use the absolute wasted effort (AWE) and Top-N, two widely used metrics [10], [12], [47] to evaluate the effectiveness of the studied fault localization techniques. Note that all our metrics do not consider test code.

AWE: Given a faulty program and a ranking formula (such as Tarantula), AWE is defined as the ranking number of the faulty method. However, in some cases, there are more than one method sharing the same SBFL score with faulty method, and AWE is defined as the average ranking of all the tied methods. The AWE is computed as:

$$AWE(b) = |\{m|susp(m) > susp(b)\}|+ \\ |\{m|susp(m) = susp(b)\}|/2 + 1/2 \quad (16)$$

where b is the faulty method and m is any candidate method except b and $|\{\cdot\}|$ is the cardinality of a set. The range of AWE is from 1 to the total number of methods. A smaller AWE means the fault localization is more effective and the ideal value is 1.

2. http://www.intellifl.org/

Top-N: This metric counts the number of successfully localized faulty methods within the top-N (N=1, 3, 5) ranked results. If the faulty methods share the same score, we use the average position to present fault location. Higher Top-N denotes more effective fault localization. Note that this metric can be quite important in practice since developers usually only inspect top-ranked elements, e.g., over 70% developers only check Top-5 ranked elements [58].

6 Result Analysis

6.1 RQ1: PRFL's Overall Effectiveness and Efficiency

TABLE 5: Results of SBFL and PRFL on All Defects4J Faults

	Tech	Top-1		Top-3		Top-5		AWE		
		S	P	S	P	S	P	S	P	Impr.
Chart	Tarantula	7	12	20	21	22	24	14.92	13.63	8.63%
	SBI	7	11	20	21	22	24	14.92	13.63	8.63%
	Ochiai	6	11	17	20	19	24	11.83	10.21	13.66%
	Jaccard	6	11	17	20	20	24	12.12	10.60	12.54%
	Ochiai2	6	12	17	21	21	24	12.15	11.40	6.17%
	Kulczynski	6	11	17	20	20	24	12.12	10.60	12.54%
	Dstar2	5	10	16	21	19	24	13.65	9.83	28.03%
	Op2	5	7	14	17	16	20	66.19	61.37	7.29%
Lang	Tarantula	21	27	45	50	57	57	5.81	5.26	9.40%
	SBI	21	27	45	50	57	57	5.81	5.26	9.40%
	Ochiai	22	30	44	51	56	58	5.35	4.76	10.94%
	Jaccard	22	28	44	50	56	57	5.39	4.88	9.42%
	Ochiai2	21	28	45	50	55	57	5.35	5.33	0.29%
	Kulczynski	22	28	44	50	56	57	5.39	4.88	9.42%
	Dstar2	23	30	45	50	55	58	5.33	4.65	12.70%
	Op2	23	29	45	48	56	56	5.48	4.61	15.99%
Math	Tarantula	24	32	61	65	73	78	8.89	7.36	17.20%
	SBI	24	32	61	65	73	78	8.87	7.36	17.07%
	Ochiai	24	33	60	65	73	83	9.25	6.92	25.23%
	Jaccard	24	32	61	66	73	79	8.87	7.33	17.39%
	Ochiai2	24	31	61	67	73	79	8.88	7.35	17.16%
	Kulczynski	24	32	61	66	73	79	8.87	7.33	17.44%
	Dstar2	24	33	60	65	73	83	9.40	6.91	26.46%
	Op2	23	28	54	61	65	76	10.66	8.35	21.64%
Time	Tarantula	5	7	11	14	16	16	25.37	23.30	8.18%
	SBI	5	7	11	14	16	16	25.37	23.30	8.18%
	Ochiai	6	7	11	13	18	18	22.93	19.93	13.09%
	Jaccard	5	7	9	12	18	17	25.41	23.44	7.73%
	Ochiai2	5	7	11	14	16	17	25.44	23.33	8.30%
	Kulczynski	5	7	9	12	18	17	25.41	23.44	7.73%
	Dstar2	6	7	11	12	12	12	25.22	21.48	14.83%
	Op2	8	4	12	11	14	13	62.67	60.19	3.96%
Closure	Tarantula	12	13	26	33	36	45	120.30	87.01	27.68%
	SBI	12	13	26	33	36	45	120.30	87.02	27.66%
	Ochiai	14	20	29	43	39	59	108.70	75.74	30.33%
	Jaccard	13	14	27	34	37	47	119.40	86.29	27.73%
	Ochiai2	13	13	26	33	37	46	120.01	86.83	27.64%
	Kulczynski	13	14	27	34	37	47	119.40	86.29	27.73%
	Dstar2	14	21	28	42	39	59	108.23	74.77	30.91%
	Op2	17	10	33	32	42	43	119.26	87.76	26.41%
Mockito	Tarantula	6	7	18	22	25	31	39.51	34.89	11.69%
	SBI	6	7	18	22	25	31	39.51	34.89	11.69%
	Ochiai	7	9	18	20	25	32	49.91	36.66	26.55%
	Jaccard	6	7	18	22	25	30	39.32	34.42	12.45%
	Ochiai2	6	7	18	22	25	30	39.46	35.24	10.70%
	Kulczynski	6	7	18	22	25	30	39.32	34.42	12.45%
	Dstar2	7	8	17	19	23	33	57.96	44.34	23.50%
	Op2	6	5	14	16	19	23	155.43	154.03	0.91%
Overall	Tarantula	75	98	181	205	229	251	35.80	28.58	20.18%
	SBI	75	97	181	205	229	251	35.80	28.58	20.17%
	Ochiai	79	110	179	212	230	274	34.66	25.70	25.85%
	Jaccard	76	99	176	204	229	254	35.08	27.83	20.68%
	Ochiai2	75	98	178	207	227	253	35.21	28.25	19.78%
	Kulczynski	76	99	176	204	229	254	35.08	27.83	20.68%
	Dstar2	79	109	177	209	221	269	36.63	27.00	26.30%
	Op2	82	83	172	185	212	231	69.95	62.72	10.34%

Column **S** and **P** indicate SBFL and PRFL respectively

6.1.1 Effectiveness of PRFL

To answer this RQ, we present the experimental results of PRFL using the default configuration (d=0.7, α=0.001 and δ=1.0) on all the real faults from the Defects4J dataset. Table 5 presents the overall results. In the table, different columns present different effectiveness metrics. (for each metric, Column `S` represents the traditional spectrum-based techniques while Column `P` represents our PRFL)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

TABLE 6: Fault Localization Overheads

Sub	COV	CG	DP	Analysis	Ranking	Total
Chart	35.18	66.71	6.49	1.33	0.01	109.72
Closure	231.73	431.71	888.87	6.61	0.01	1558.94
Lang	23.85	22.26	0.81	0.38	0.01	47.31
Math	268.32	106.21	8.33	1.47	0.01	384.34
Mockito	45.37	38.93	4.30	0.15	0.01	88.76
Time	21.56	25.72	19.39	1.32	0.01	68.00
Avg.	104.33	115.23	154.70	1.87	0.01	376.17

and different rows present the subjects and fault localization formulas used. Also, the bottom portion of the table presents the overall results for all Defects4J subjects, e.g., the total Top-N values and the average AWE values. From Table 5, we can have the following observations. First, overall Ochiai and Dstar2 (marked in gray) are the two most effective SBFL techniques for all the faults in Defects4J. Also, both of them are able to localize 109+ faults within Top-1 and 209+ faults within Top-3. Second, in general, PRFL is able to boost all the studied traditional SBFL techniques. For example, the overall Top-1/3/5 and AWE values of all traditional techniques are all outperformed by the corresponding PRFL techniques. Third, interestingly, PRFL tends to boost more effective traditional SBFL techniques even more. For example, PRFL is able to boost the number of faulty methods ranked as Top-1 by Ochiai from 79 to 110 (i.e., **39% more**), a higher improvement than the other inferior techniques.

Table 7 presents the performance of PRFL and SBFL on Bugs.jar dataset. Similarly, as presented in Table 5, we can observe that overall Ochiai and Dstar2 are still the two most effective SBFL techniques for all the faults in Bugs.jar. Specifically, by applying PRFL , Ochiai can locate 47 (30%+), 139 (27.5%+), and 187 (18.4%+) faults, and Dstar2 can locate 48 (26.3%+), 141 (22.6%+), and 190 (15.8%+) faults within Top 1, 3 and 5. However, we found the improvements of AWE on Bugs.jar is not globally consistent to the observation on Defects4J. For example, overall PRFL can improve more than 8% AWE using Ochiai and Dstar2, but it increases around 13% AWE on Math.

6.1.2 Efficiency of PRFL

We record the overhead of our PRFL technique. Table 6 presents the average overhead results for all versions of each subject from Defects4J. In Table 6, each column presents the time(seconds) spent in each phase of PRFL while the last column presents the total overhead; each row presents the overhead for each subject while the last row presents the average results for all the subjects. Column `COV`, Column `CG` and Column `Analysis` respectively represent coverage collection, call graph construction, and PageRank analysis costs. Column `DP` represents time of processing data, which includes reading data (statement coverage and call graph) and constructing matrices for PageRank analysis. Based on the table, the PRFL technique is lightweight and can finish within around 6 minutes for the studied subject on average. Furthermore, the most time-consuming phase is `DP` while the overheads of other PRFL phases (i.e., the call graph and PageRank analysis time) are comparably low, e.g., less than 10 seconds for `CG` and less than 2 seconds for `Analysis`. The reason is that `DP` is implemented in Python, which is relatively slow and also does not enable multi-threading.

Furthermore, we investigate the memory consumption of PRFL. In Table 8, the column "F-matrix" and "P-matrix" present the allocated memory of PageRank matrices of

TABLE 7: Results of SBFL and PRFL on All Bugs.jar Faults

	Tech	Top-1		Top-3		Top-5		AWE		
		S	P	S	P	S	P	S	P	Impr.
Accumulo	Tarantula	2	2	16	23	20	24	228.66	232.94	-1.87%
	SBI	2	2	16	18	20	22	228.66	233.07	-1.93%
	Ochiai	3	6	14	28	19	32	230.25	229.68	0.25%
	Jaccard	3	6	15	28	19	33	228.95	231.00	-0.89%
	Ochiai2	3	7	13	26	19	28	228.06	231.70	-1.59%
	Kulczynski	3	6	15	28	19	33	228.95	231.00	-0.89%
	Dstar2	4	7	18	27	21	33	232.41	230.57	0.79%
	Op2	5	6	20	23	20	26	245.24	236.45	3.58%
Math	Tarantula	9	12	29	34	34	35	18.01	19.13	-6.21%
	SBI	9	12	29	34	34	35	18.01	19.13	-6.21%
	Ochiai	11	14	29	34	34	35	18.90	21.46	-13.49%
	Jaccard	11	14	29	34	34	35	18.12	19.11	-5.42%
	Ochiai2	11	13	29	34	34	35	18.01	19.11	-6.08%
	Kulczynski	11	14	29	34	34	35	18.12	19.11	-5.42%
	Dstar2	11	14	29	34	34	35	18.96	21.47	-13.25%
	Op2	9	12	19	28	25	30	22.09	28.75	-30.14%
Flink	Tarantula	3	5	20	20	33	31	62.35	53.06	14.90%
	SBI	3	5	20	20	33	31	62.35	53.06	14.90%
	Ochiai	3	4	17	20	28	34	75.32	58.36	22.51%
	Jaccard	3	6	18	22	25	31	70.64	53.80	23.83%
	Ochiai2	4	6	19	22	27	32	65.15	52.90	18.80%
	Kulczynski	3	6	18	22	25	31	70.64	53.80	23.83%
	Dstar2	3	4	17	19	28	31	82.89	62.37	24.76%
	Op2	2	3	15	19	24	28	96.73	79.12	18.20%
Oak	Tarantula	8	8	21	21	36	36	148.54	141.45	4.77%
	SBI	8	8	21	21	36	36	148.54	141.41	4.80%
	Ochiai	10	10	23	22	38	37	140.66	140.10	0.40%
	Jaccard	10	10	23	22	38	37	149.50	142.65	4.58%
	Ochiai2	10	10	23	22	37	36	148.28	141.26	4.73%
	Kulczynski	10	10	23	22	38	37	149.50	142.69	4.55%
	Dstar2	10	11	24	25	40	39	140.31	140.37	-0.04%
	Op2	10	11	25	25	35	35	206.58	206.00	0.28%
Log4j2	Tarantula	2	2	12	13	20	21	145.11	123.64	14.80%
	SBI	2	2	12	13	20	21	145.11	123.58	14.84%
	Ochiai	3	6	12	17	20	23	154.55	133.10	13.88%
	Jaccard	3	6	12	19	20	24	156.99	132.03	15.90%
	Ochiai2	3	5	12	16	21	24	150.45	124.75	17.08%
	Kulczynski	3	6	12	19	20	24	156.99	132.03	15.90%
	Dstar2	4	5	13	18	20	24	163.12	138.77	14.93%
	Op2	3	5	12	18	15	21	202.07	165.82	17.94%
Wicket	Tarantula	5	5	12	12	17	17	145.07	133.12	8.23%
	SBI	5	5	12	12	17	17	145.07	133.20	8.18%
	Ochiai	6	7	14	18	19	26	249.55	215.63	13.59%
	Jaccard	6	7	14	18	19	26	178.07	144.38	18.92%
	Ochiai2	6	7	15	16	20	24	163.85	170.91	-4.31%
	Kulczynski	6	7	14	18	19	26	178.07	144.38	18.92%
	Dstar2	6	7	14	18	21	28	267.15	230.00	13.91%
	Op2	6	6	14	17	19	23	311.37	273.23	12.25%
Overall	Tarantula	29	34	110	123	160	164	124.62	117.22	5.94%
	SBI	29	34	110	118	160	162	124.62	117.24	5.92%
	Ochiai	36	47	109	139	158	187	144.87	133.05	8.16%
	Jaccard	36	49	111	143	155	186	133.71	120.49	9.89%
	Ochiai2	37	48	111	136	158	179	128.97	123.44	4.29%
	Kulczynski	36	49	111	143	155	186	133.71	120.50	9.88%
	Dstar2	38	48	115	141	164	190	150.81	137.26	8.98%
	Op2	35	43	105	130	138	163	180.68	164.90	8.74%

Column **S** and **P** indicate SBFL and PRFL respectively

TABLE 8: Average Memory Consumption of PRFL

Proj.	F-matrix	P-matrix
Chart	0.037	248.618
Closure	0.075	1616.352
Lang	0.002	112.479
Math	0.005	382.071
Time	0.007	207.31
Mockito	0.023	42.011

Unit: MB

failing and passing tests respectively. It can be found that these results are consistent to our assumption that there are much more passing tests than failing tests. Moreover, we can observe that the memory overhead is less than 2 Gigabyte which can be easily handled by a commonly-configured personal computer.

6.2 RQ2: Configuration Impacts

In this section, we extend our experiments with different configurations to investigate the influence of internal factors of PRFL so as to learn how to make PRFL achieve better performance. Figure 7 presents the impacts of different damping factors (i.e., d) on the effectiveness of PRFL using the default α=0.001 and δ=1.0. In the figure, the x axis presents various damping factor values, while the y axis

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

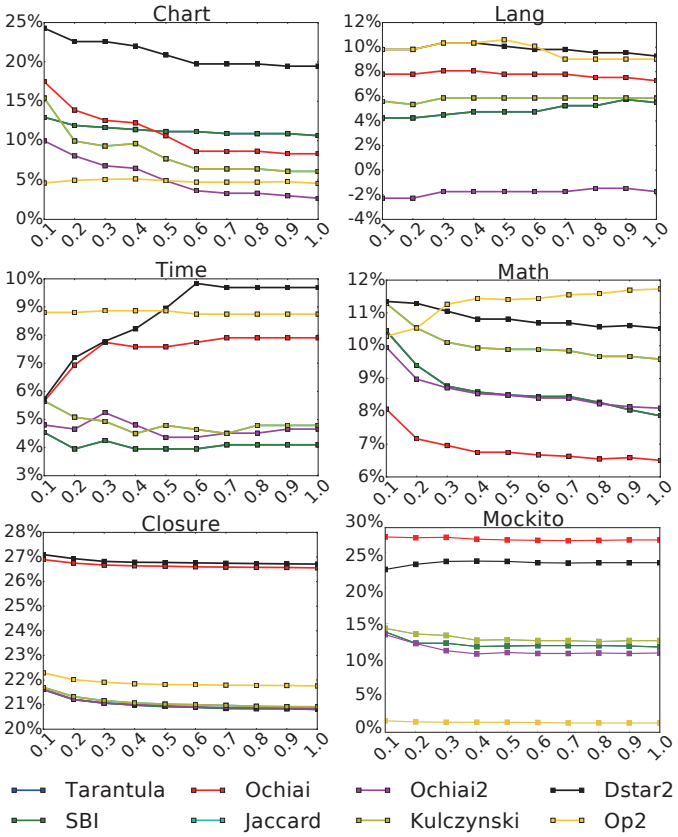


Fig. 7: Impact of Damping Factor

presents the AWE improvements of PRFL techniques over the original pure SBFL techniques (different formulas are represented using different lines). From the figure, we have the following observations. First, the damping factor does not impact the PRFL effectiveness much. For example, for all the formulas on all the subjects, the largest improvement difference among different damping factors is only 4%. Second, for the majority cases, when the damping factor increases, the improvement rates slightly decrease. This observation is as expected. The reason is that when damping factor increases, the test capabilities will be distributed a smaller weight, causing it to make less contributions in localizing the faults.

Figure 8 shows the impact of the call graph weights (i.e., α) using the default d=0.7 and δ=1.0. Similar to Figure 7, the x axis presents different call graph weights, the y axis presents the improvement rates of PRFL over pure SBFL techniques (different line represents different formulas). From the figure, we have the following observations. First, on all the subjects, the improvement rates of PRFL dramatically increase at the very beginning, but then slowly increase or even decrease for some formulas. One major reason is that when the call graph weight is 0, PRFL degrades to use only the test coverage information and cannot differentiate the methods with the same test coverage, hence the improvement rates are relative low. Note that even without call graph information (i.e., α=0), PRFL is still able to outperform the traditional SBFL techniques for the majority cases. Second, the call graph weight has different impacts on different formulas. For example, the improvement rate of PRFL over Op2 eventually decreases dramatically for subjects Time and Closure, while the improvement rates

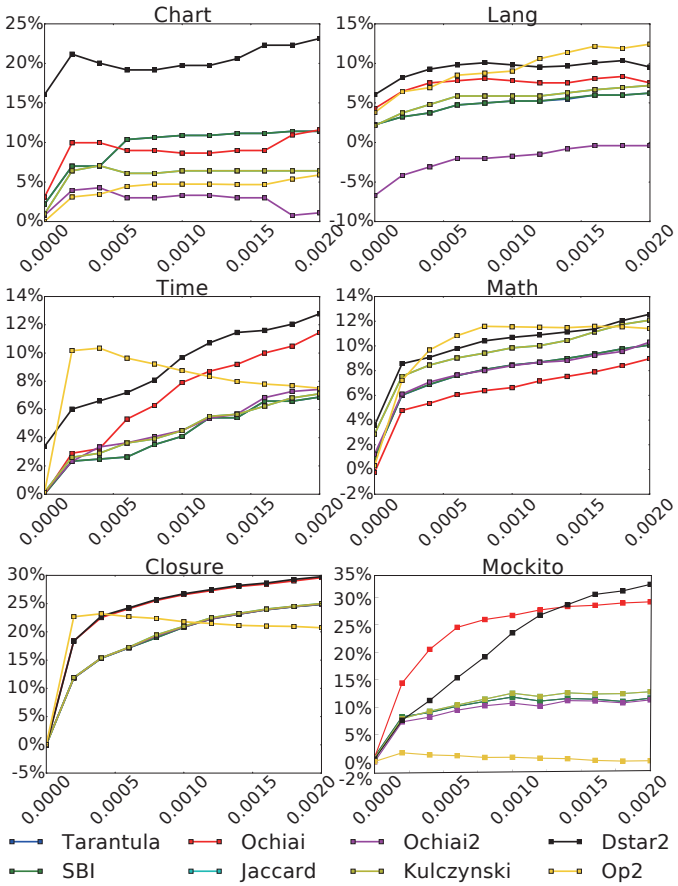


Fig. 8: Impact of Call Graph Weight

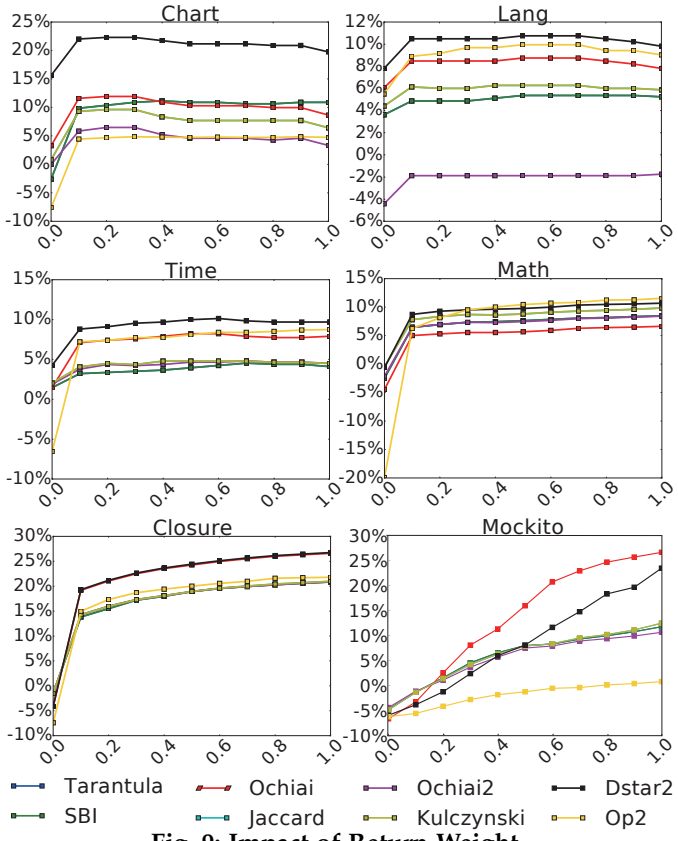


Fig. 9: Impact of Return Weight

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

keep stable or increasing for the other formulas on the most subjects. Also, the impact of call graph weight is similar for Tarantula and SBI, as well as Jaccard and Kulczynski due to the similarities in the formula definition.

Figure 9 shows the impact of the return edge weights (i.e., δ) using d=0.7 and α=0.001. Similar with Figure 7 and 8 , the x axis presents different return edge weights, while the y axis presents the improvement rates of PRFL over the traditional SBFL techniques. From this figure, it is observed that the improvement rates of PRFL (especially on Op2) on all the subjects are low and sometimes even below zero when δ=0.0. The potential reason is that when $\delta = 0.0$, all return edges are ignored in call graphs, and the invoked methods are assigned with too much faultiness or successfulness scores. Especially, faultiness score plays a leading role in Op2 and can make callee methods more suspicious than the caller methods, thus decreasing the effectiveness of PRFL. Furthermore, interestingly, it can also be observed that the return edge weight does not impact the PRFL effectiveness much when $\delta \geqslant 0.1$, e.g., the largest improvement difference among different return edge weights is no more than 5% for all the formulas on all the subjects.

6.3 RQ3: Impact of Fault Number

As mentioned by existing work [59], over 82% of faults are single-faults, where the systems can be repaired by only eliminating one fault. This observation inspires us that it is plausible to apply simpler and near-optimal fault localization in most cases. In our experiment, we obtain similar observation that around 80.0% of subjects are single-fault subjects (309 out of 395 subjects in Defects4J v1.2), which also validates the prevalence of single-fault faults in software development. Hence, we classify the program faults as single-fault and multi-location-faults. Specifically, for method-level fault localization, single-fault is defined as all bug fixes located in a single method. In contrast, multi-location-faults refers to bug fixes in multiple methods. In our evaluation, we define localizing multi-location-faults as multiple tasks of single-fault localization where an ideal technique is expected to rank all faulty methods on the top.

The real faults from Defects4J include both single-fault and multi-location-faults. Table 9 presents the experimental results, where the left half presents the results on single-fault versions while the right half presents the results on the multi-location-faults versions. From the table, we have the following findings. First, we find that the traditional techniques perform differently on single-fault and multi-location-faults versions. For example, Dstar2 and Op2 (marked in gray in the left half) are the two overall most effective techniques for single-fault versions, while Tarantula and SBI (marked in gray in the right half) are the two overall most effective techniques for multi-location-faults versions. In particular, Op2 performs the best on single-fault versions (also confirmed by prior work [35]), e.g., with the highest Top-1 value (i.e., 68) and the lowest AWE value (i.e., 24.17), but performs the worst on multi-location-faults versions, e.g., with the lowest Top-1 value (i.e., 14) and the highest AWE value (i.e., 63.98). The major reason might be that Op2 is specifically designed and also shown to be optimal for single-fault programs [6], but it cannot perform well for multi-location-faults programs. Second, we find that despite the fact that various techniques perform differently on single-fault or multi-location-faults programs, PRFL is able to boost almost all the studied techniques similarly on both single-fault and multi-location-faults fault programs. For example, the Top-1 value improvement for Tarantula is 20.7% (from 58 to 70) on single-fault programs and 64.7% (from 17 to 28) on multi-location-faults programs. Finally, PRFL is also able to boost the originally effective techniques (e.g., Dstar2, SBI, and Tarantula) significantly. For example, the AWE improvement for Dstar2 (i.e., 28.62%) is the highest for single-fault versions. For multi-location-faults versions, the AWE improvement for two optimal techniques, Tarantula and SBI, are 23.45% and 23.41% respectively.

6.4 RQ4: Comparison of PRFL and Recent Proposed SBFL Technique

We compare PRFL with **Multric** [12], a recent proposed spectrum-based fault localization technique. Multric [12] applies pairwise learning-to-rank algorithm to ensemble multiple SBFL formulae in order to improve the accuracy of SBFL techniques. To be specific, in the experiment, we first select 34 SBFL formulae [16] to compute the suspiciousness scores for generating feature vectors for each method.

Then, we adopted LIBSVM[3], a widely-used library for support vector machines, and XGBoost[4], an widely-used optimized distributed gradient boosting library, to investigate the effectiveness. More specifically, we used RankSVM with linear kernel from LIBSVM (default settings), and LambdaRank from XGBoost (booster = `gbtree`, max_depth = 60, num_round = 100, colsample_bytree=0.85, and eta =0.5). Moreover, we performed a leave-one-out cross validation [16], [47] not only across each version of the six projects, but also across whole six projects. For the total N bugs, we can separate them into two groups: (1) one group only includes one bug is the test data for predicting its rank; (2) and the other group with remaining N - 1 bugs as the training data is to build the ranking model.

Table 10 presents the results of Multric, Ochiai, and PRFL. From the table, we can observe that Multric can totally locate 80/188/236 faults within Top-1/3/5, which performs slightly better than Ochiai. However, after applying PRFL, Ochiai is effectively boosted and it can rank more faults within Top-1/3/5. We can learn from the results that Multric can generate better formula to compute suspiciousness scores. However, the program spectrum is still the bottleneck which limits its performance. On the other hand, the improvement from PRFL suggests the benefit of using PageRank for extracting more information from coverage and call graphs.

6.5 RQ5: Comparison of PRFL, MBFL and the integrated approach

We first compare PRFL with two recent proposed mutation-based fault localization (MBFL) techniques: **Metallaxis** [30] and **MBFL-hybrid-avg** [18]. Then, we study a simple integration of PRFL and MBFL and present its performance.

3. https://www.csie.ntu.edu.tw/ cjlin/libsvm
4. https://github.com/dmlc/xgboost

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

TABLE 9: Overall fault Localization Results on Single-fault and Multi-location-faults of Defects4J

Tech	Single-fault versions									Multi-location-faults versions								
	Top-1		Top-3		Top-5		AWE			Top-1		Top-3		Top-5		AWE		
	S	P	S	P	S	P	S	P	Impr.	S	P	S	P	S	P	S	P	Impr.
Tarantula	58	70	122	135	150	161	30.81	24.20	21.44%	17	28	59	70	79	90	27.28	20.88	23.45%
SBI	58	70	122	135	150	161	30.81	24.20	21.44%	17	27	59	70	79	90	27.28	20.89	23.41%
Ochiai	63	82	128	144	156	178	26.41	19.00	28.06%	16	28	51	68	74	96	26.22	20.43	22.08%
Jaccard	60	73	123	137	153	163	29.42	23.02	21.74%	16	26	53	67	76	91	27.27	20.66	24.26%
Ochiai2	59	71	122	136	151	162	29.83	23.64	20.76%	16	27	56	71	76	91	27.22	20.83	23.47%
Kulczynski	60	73	123	137	153	163	29.42	23.02	21.74%	16	26	53	67	76	91	27.28	20.66	24.27%
Dstar2	64	82	129	146	156	179	26.35	18.81	28.62%	15	27	48	63	65	90	27.19	20.58	24.31%
Op2	68	67	136	142	162	174	24.17	19.75	18.30%	14	16	36	43	50	57	63.98	53.23	16.81%

Column **S** and **P** indicate SBFL and PRFL respectively

TABLE 10: Effectiveness of Ochiai, Multric and PRFL

Tech.	Proj.	Top1	Top3	Top5	AWE
Ochiai	Chart	6	17	19	11.83
	Lang	22	44	56	5.35
	Math	24	60	73	9.25
	Time	6	11	18	22.93
	Mockito	7	18	25	49.91
	Closure	14	29	39	108.70
	Overall	79	179	230	34.66
Multric	Chart	7	18	21	10.88
	Lang	23	47	59	6.9
	Math	21	57	69	25.17
	Time	6	14	14	39.11
	Mockito	6	14	24	48.55
	Closure	17	38	49	134.44
	Overall	80	188	236	44.18
PRFL-Ochiai	Chart	11	20	24	10.21
	Lang	30	50	58	4.76
	Math	33	65	83	6.92
	Time	7	13	18	19.93
	Mockito	9	20	32	36.66
	Closure	20	43	59	75.74
	Overall	110	212	274	25.70

Metallaxis [30] is the first mutation-based fault localization technique. Metallaxis supposes that a mutant leads to different failure outputs/messages for the failing tests, and the mutated program entity may have high impact on the failing tests, which means it is more likely to be the root cause of the test failures. MBFL-hybrid-avg is an improved mutation-based fault localization technique which averages each statement's MBFL suspiciousness with the suspiciousness calculated by a SBFL technique, and it performs better than other proposed techniques in [18].

In the experiment, we choose Ochiai as the basic SBFL technique for it is the most effective one on PRFL, and we customize MBFL-hybrid-avg as follows: first, to reduce the time complexity, we only mutate the methods covered by failing tests following the original work [18]; second, instead of statement-level fault localization, we average the MBFL scores of all the statements within each method and use it to present the method suspiciousness. Since project `Closure` is much larger than other projects and its associated mutation testing is very time consuming, we ignore `Closure` and apply the approaches on the other five projects of Defects4J benchmark. Moreover, we choose Metallaxis as the basic MBFL technique of MBFL-hybrid-avg, and we use the following formula to compute Metallaxis score:

$$s(e) = \max_{m \in M(e)} \left(\frac{|T_f^{(m)}(e)|}{\sqrt{(|T_f^{(m)}(e)| + |T_p^{(m)}(e)|) \cdot |T_f|}} \right) \quad (17)$$

where $M(e)$ denotes the set of all mutants of method e, $|T_f^{(m)}(e)|$ and $|T_p^{(m)}(e)|$ denote the number of the failing and passing tests impacted by mutant m, and $|T_f|$ denotes the total number of failing tests.

Furthermore, we study a simple integration of PRFL and Metallaxis, PR-MBFL, which combines the passing and failing test weights to the native Metallaxis. Specifically, the PR-MBFL score can be computed using the following formula:

$$s(e) = \max_{m \in M(e)} \left(\frac{\sum s_f^{(m)}(e)}{\sqrt{(\sum s_f^{(m)}(e) + \sum s_p^{(m)}(e)) \cdot \sum s_f}} \right) \quad (18)$$

Similar to Formula 17, $\sum s_f^{(m)}(e)$ and $\sum s_p^{(m)}(e)$ present the sum of the weights of the failing and passing tests impacted by mutant m, $\sum s_f$ denotes the sum of the weights of all failing tests, and the test weight s_f and s_p can be extracted from Equation 14.

We present the experimental results of PRFL, Metallaxis, MBFL-hybrid-avg (abbreviated as MB-hy-avg) and PR-MBFL in Table 11. It can be observed that, first, PRFL performs better than other techniques on all subjects. Second, MBFL-hybrid-avg locates less faults with in Top-1/3/5 than Metallaxis for the method-level fault localization. Third, PR-MBFL can locate more faults at Top-1, but it does not perform very on Top-3/5 compared to Metallaxis.

TABLE 11: Overall Results of the Comparison of PRFL, MBFL and Their Combination

Tech.	Proj.	Top 1	Top 3	Top 5	AWE
Metallaxis	Chart	7	19	22	11.88
	Lang	31	55	60	3.12
	Math	19	69	84	6.77
	Time	6	11	15	18.20
	Mockito	9	19	27	52.42
	Overall	72	173	208	15.40
PR-MBFL	Chart	8	20	21	18.16
	Lang	24	48	57	4.22
	Math	20	55	68	10.53
	Time	4	9	12	18.52
	Mockito	6	13	25	52.58
	Overall	62	145	183	17.33
MB-hy-avg	Chart	7	15	22	14.00
	Lang	31	54	60	3.38
	Math	24	63	80	7.24
	Time	7	12	14	18.17
	Mockito	9	20	27	56.05
	Overall	78	164	203	16.47
PRFL	Chart	11	20	24	10.21
	Lang	30	50	58	4.76
	Math	33	65	83	6.92
	Time	7	13	18	19.93
	Mockito	9	20	32	36.66
	Overall	90	168	215	13.08

6.6 RQ6: Comparison of the constrained PageRank and other link analysis algorithms

We compare the accuracy of PRFL using different link analysis algorithms: the standard PageRank (STPR), the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

TABLE 12: Overall Results of the Comparison of Link Analysis Algorithms

	Tech	STPR				PRFL			
		Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE
Overall	Tarantula	63	135	171	50.45	98	204	251	28.58
	SBI	63	135	171	50.45	97	204	251	28.58
	Ochiai	62	130	160	48.11	110	211	274	25.70
	Jaccard	62	128	160	53.00	99	203	254	27.83
	Ochiai2	61	134	164	50.72	98	206	253	28.25
	Kulczynski	62	128	160	53.00	99	203	254	27.83
	Dstar2	61	126	157	53.29	109	208	269	27.00
	Op2	62	122	160	58.38	83	184	231	62.72

	Tech	HITS				SALSA			
		Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE
Overall	Tarantula	72	174	222	31.26	21	81	101	118.65
	SBI	72	174	222	31.26	21	81	101	118.65
	Ochiai	81	189	239	28.03	21	81	101	118.65
	Jaccard	74	178	226	30.65	21	81	101	118.65
	Ochiai2	74	178	225	31.04	21	81	101	118.65
	Kulczynski	74	178	226	30.65	21	81	101	118.65
	Dstar2	81	186	235	28.83	21	81	101	118.65
	Op2	72	161	198	62.33	21	81	101	118.65

constrained PageRank (PRFL), HITS, and SALSA. To implement these approaches, we first construct graphs with the nodes presenting the methods/tests and edges presenting the method invocation and test coverage. Note that in this graph all nodes and edges are congeneric. Next, we generate adjacent matrices based on the graphs and implement the corresponding algorithms introduced in [38]. For HITS and SALSA, every node has authority and hub scores, and in this experiment, we only choose the authority score to present the node importance. Table 12 shows the results of the approaches with different link analysis algorithms, and we can observe that the constrained PageRank outperforms the standard PageRank, which validates the necessity of edge differentiation. On the other hand, HITS and SALSA perform poorly compared to the constrained PageRank. Such results inspire us that it is necessary for the the vanilla link analysis algorithms to be customized for fault localization. Interestingly, SALSA computes the same score on all the formalae. We study the intermediate results and find that all the methods share the same faultiness and successfulness scores, because the graphs and adjacent matrices of failing and passing tests are symmetric, and SALSA sets the same score to all nodes.

6.7 RQ7: Effectiveness of PRFL+

We study the effecivesness of PRFL+ by analyzing the impact of dynamic call graph, passing test weight adjustment, and test and method reduction, respectively.

6.7.1 Impact of dynamic call graph

In this section, we design experiments in order to verify whether adopting dynamic call graphs would impact on Top-N rank, that is presented in Table 13. The results show that using dynamic call graphs instead of static call graphs makes no obvious changes on Top-N. For instance, in row Ochiai, PRFL with static call graphs captures 110 faults while PRFL with dynamic call graphs captures 109 faults. On the other hand, PRFL can perform slightly better on Tarantula using dynamic call graphs, which can localize 2 more faults on Top-3 (205 vs. 206) and Top-5 (251 vs. 253). We suspect the reason to be two-fold: (1) the majority of static call graph edges and nodes are also within the dynamic call graphs, (2) call graphs are used to fine-tune the final fault localization results and may not impact the results significantly.

TABLE 13: Overall Results of Dynamic Call Graph

Tech	Top-1		Top-3		Top-5	
	S	D	S	D	S	D
Tarantula	98	98	205	206	251	253
SBI	97	97	205	206	251	251
Ochiai	110	109	212	212	265	265
Jaccard	99	99	204	204	254	254
Ochiai2	98	98	207	207	253	253
Kulczynski	99	99	204	204	254	254
Dstar2	109	109	209	209	269	269
Op2	83	83	185	185	231	231

Column **S** and **D** indicate static and dynamic call graph

6.7.2 Impact of Passing Test Weight Adjustment

In this section, we extend our experiments to explore the performance of passing test weight adjustment. Table 14 shows the fault localization results of PRFL without passing test weight adjustment. Specifically, the columns represent the evaluated metrics (i.e., Top-1/3/5, and AWE). For each metric, Column `P` represents PRFL without passing test weight adjustment and Column `WA` represents PRFL with passing test weight adjustment. The rows represent the subjects with the eight fault localization formulas. The overall results of the six Defects4J subjects are displayed in the bottom row of the table.

TABLE 14: Results of Passing Test Weight Adjustment

	Tech	Top-1		Top-3		Top-5		AWE		
		P	WA	P	WA	P	WA	P	WA	impr.
Chart	Tarantula	12	12	21	21	24	24	13.63	13.67	-0.28%
	SBI	11	11	21	21	24	24	13.63	13.67	-0.28%
	Ochiai	11	11	20	20	24	24	10.21	10.21	0.00%
	Jaccard	11	11	20	20	24	24	10.60	10.60	0.00%
	Ochiai2	12	12	21	21	24	24	11.40	11.37	0.34%
	Kulczynski	11	11	20	20	24	24	10.60	10.60	0.00%
	Dstar2	10	10	21	21	24	24	9.83	9.83	0.00%
	Op2	7	7	17	17	20	20	61.37	61.37	0.00%
Lang	Tarantula	27	27	50	50	57	57	5.26	5.25	0.29%
	SBI	27	27	50	50	57	57	5.26	5.25	0.29%
	Ochiai	30	30	51	51	58	58	4.76	4.76	0.00%
	Jaccard	28	28	50	50	57	57	4.88	4.87	0.31%
	Ochiai2	28	28	50	50	57	57	5.33	5.32	0.29%
	Kulczynski	28	28	50	50	57	57	4.88	4.87	0.31%
	Dstar2	30	30	50	50	58	58	4.65	4.64	0.33%
	Op2	29	29	48	48	56	56	4.61	4.59	0.33%
Math	Tarantula	32	32	65	65	78	79	7.36	7.28	1.03%
	SBI	32	32	65	65	78	79	7.36	7.28	1.03%
	Ochiai	33	33	65	66	83	83	6.92	6.85	0.95%
	Jaccard	32	32	66	66	79	80	7.33	7.25	1.03%
	Ochiai2	31	31	67	67	79	80	7.35	7.29	0.90%
	Kulczynski	32	32	66	66	79	80	7.33	7.25	1.03%
	Dstar2	33	34	65	65	83	83	6.91	6.84	0.96%
	Op2	28	28	61	61	76	76	8.35	8.32	0.45%
Time	Tarantula	7	7	14	14	16	16	23.30	23.26	0.16%
	SBI	7	7	14	14	16	16	23.30	23.26	0.16%
	Ochiai	7	7	13	13	18	18	19.93	19.96	-0.19%
	Jaccard	7	7	12	12	17	17	23.44	23.33	0.47%
	Ochiai2	7	7	14	14	17	16	23.33	23.22	0.48%
	Kulczynski	7	7	12	12	17	17	23.44	23.33	0.47%
	Dstar2	7	7	12	12	12	12	21.48	21.59	-0.52%
	Op2	4	4	11	11	13	13	60.19	60.22	-0.06%
Closure	Tarantula	13	14	33	34	45	46	87.01	86.81	0.22%
	SBI	13	14	33	34	45	46	87.02	86.83	0.22%
	Ochiai	20	23	43	44	59	59	75.74	75.62	0.16%
	Jaccard	14	15	34	36	47	47	86.29	86.03	0.30%
	Ochiai2	13	14	33	36	46	47	86.83	86.59	0.29%
	Kulczynski	14	15	34	36	47	47	86.29	86.03	0.30%
	Dstar2	21	23	42	44	59	59	74.77	74.63	0.19%
	Op2	10	10	32	32	43	42	87.76	87.74	0.03%
Mockito	Tarantula	7	7	22	21	31	31	34.89	34.74	0.45%
	SBI	7	7	22	21	31	31	34.89	34.74	0.45%
	Ochiai	9	9	20	20	32	32	36.66	36.34	0.86%
	Jaccard	7	7	22	22	30	31	34.42	34.08	0.99%
	Ochiai2	7	7	22	22	30	31	35.24	34.89	0.97%
	Kulczynski	7	7	22	22	30	31	34.42	34.08	0.99%
	Dstar2	8	8	19	19	33	33	44.34	44.05	0.65%
	Op2	5	5	16	16	23	23	154.03	153.87	0.10%
Overall	Tarantula	98	99	205	205	251	253	28.58	28.50	0.04%
	SBI	97	98	205	205	251	253	28.58	28.50	0.04%
	Ochiai	110	113	212	214	274	274	25.70	25.62	0.09%
	Jaccard	99	100	204	206	254	256	27.83	27.69	0.05%
	Ochiai2	98	99	207	210	253	255	28.25	28.11	0.08%
	Kulczynski	99	100	204	206	254	256	27.83	27.69	0.05%
	Dstar2	109	112	209	211	269	269	27.00	26.93	0.06%
	Op2	83	83	185	185	231	230	62.72	62.68	0.00%

Weight Modification effectiveness

Legend deterioration improvement

Column **P** and **WA** indicate PRFL and PRFL with Passing Test Weight Adjustment respectively

From this table, it can be generally observed that passing

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

test weight adjustment do not incur significant differences for PRFL analysis. For instance, for Ochiai, that is the most effective SBFL technique for PRFL analysis, PRFL can locate 110 out of 395 faults within Top-1, while PRFL with passing weight adjustment can locate 113, which slightly improves the performance of PRFL.

TABLE 15: Result Snippet of Project: Math in Version 1

Method Name	Failing Score		Passing Score	
	PRFL	WA	PRFL	WA
*.Fraction:getNumerator()I	0.484206	0.484206	0.015727	0.015912
*.BigFraction:getDenominatorAsInt()I	0.486999	0.486999	0.007583	0.007799
*.BigFraction:<init>(DDII)V	0.511939	0.511939	0.003186	0.003262
*.util.FastMath:floor(D)D	0.978647	0.978647	0.366248	0.366175
*.util.FastMath:abs(J)J	0.487929	0.487929	0.440933	0.439075

We intercept a resulting snippet from Project Math, version 1 in Table 15, where the SBFL scores of PRFL without passing test weight adjustment are displayed. From Table 15, it can be observed that the failing score for each method is not affected since the weights of failing tests are not tuned. On the other hand, after modifying the weights on passing tests, the most scores are changed by lower than 10^{-3}. The SBFL scores can be subsequently computed in PRFL based on weighted spectra. Therefore, we can conclude that passing test weight adjustment has only slight impact on PRFL.

6.7.3 Impact of Test and Method Reduction

The efficacy of reducing tests and methods for PRFL analysis are demonstrated in Table 16 and 17, respectively. In both tables, the columns present the evaluated metrics in Top-N and AWE of PRFL without test and method reduction, and the rows represent the subjects and fault localization formulas. In the bottom rows of both tables, we present the overall results for six Defects4J subjects. Moreover, the performance improvement and deterioration by adopting test and method reduction is marked being dark and light gray, respectively.

Impact of Test-based Reduction. From Table 16, it can be observed that the overall performance of fault localization is only slightly changed. For instance, for PRFL with test-based reduction, Ochiai2 and Op2 deteriorate, and Jaccard, Kulczynski, and Dstar improve at Top-1, within a slight range, compared with the original PRFL. The changes on AWE are mostly limited within 2%.

Impact of Method-based Reduction. From Table 17, it can be shown that similar to the test-based reduction approach, the method-based reduction approach does not incur significant changes on Top-N and AWE.

The Number of Tests and Methods after Pruning. We further explore the average number of remaining tests and methods after pruning, as shown in Table 18. We design a metric, namely *reducing rate*, to present the level of the reducing the number of tests and methods after applying the pruning. Specifically, the *reducing rate* $R = (1 - \frac{F}{I}) \times 100\%$, where I and F are the numbers of the original and remained tests/methods after pruning, respectively.

In Table 18, the left part of the table presents the results of the number of the remaining tests after applying the test-based reduction approach in terms of the six projects. It indicates that the test-based reduction lead to different remaining number of tests among different projects. For instance, for project Lang, the *reducing rate* is as large as

TABLE 16: Results of Test-based Reduction

	Tech	Top-1		Top-3		Top-5		AWE		
		P	TR	P	TR	P	TR	P	TS	impr.
Chart	Tarantula	12	12	21	20	24	24	13.63	13.67	-0.28%
	SBI	11	11	21	20	24	24	13.63	13.67	-0.28%
	Ochiai	11	11	20	20	24	24	10.21	10.37	-1.51%
	Jaccard	11	11	20	20	24	23	10.60	10.90	-2.90%
	Ochiai2	12	11	21	20	24	25	11.40	11.13	2.36%
	Kulczynski	11	11	20	20	24	23	10.60	10.90	-2.90%
	Dstar2	10	9	21	20	24	22	9.83	11.17	-13.70%
	Op2	7	6	17	17	20	20	61.37	61.40	-0.06%
Lang	Tarantula	27	27	50	50	57	57	5.26	5.28	-0.29%
	SBI	27	27	50	50	57	57	5.26	5.28	-0.29%
	Ochiai	30	29	51	50	58	58	4.76	4.72	0.97%
	Jaccard	28	29	50	50	57	58	4.88	4.78	2.20%
	Ochiai2	28	26	50	50	57	57	5.33	5.08	4.76%
	Kulczynski	28	29	50	50	57	58	4.88	4.78	2.20%
	Dstar2	30	30	50	49	58	56	4.65	4.82	-3.64%
	Op2	29	29	48	48	56	56	4.61	4.73	-2.67%
Math	Tarantula	32	32	65	65	78	79	7.36	7.32	0.51%
	SBI	32	32	65	65	78	79	7.36	7.32	0.51%
	Ochiai	33	34	65	66	83	84	6.92	6.88	0.55%
	Jaccard	32	33	66	67	79	80	7.33	7.27	0.77%
	Ochiai2	31	32	67	68	79	79	7.35	7.30	0.77%
	Kulczynski	32	33	66	67	79	80	7.33	7.27	0.77%
	Dstar2	33	34	65	64	83	84	6.91	6.86	0.68%
	Op2	28	28	61	60	76	76	8.35	8.33	0.23%
Time	Tarantula	7	7	14	14	16	16	23.30	23.30	0.00%
	SBI	7	7	14	14	16	16	23.30	23.30	0.00%
	Ochiai	7	7	13	12	18	17	19.93	20.11	-0.93%
	Jaccard	7	7	12	12	17	17	23.44	23.41	0.16%
	Ochiai2	7	7	14	14	17	17	23.33	23.26	0.32%
	Kulczynski	7	7	12	12	17	17	23.44	23.41	0.16%
	Dstar2	7	7	12	12	12	12	21.48	21.70	-1.03%
	Op2	4	4	11	11	13	13	60.19	60.19	0.00%
Closure	Tarantula	13	13	33	33	45	45	87.01	87.00	0.01%
	SBI	13	13	33	33	45	45	87.02	87.02	0.01%
	Ochiai	20	20	43	43	59	59	75.74	75.73	0.01%
	Jaccard	14	14	34	34	47	47	86.29	86.27	0.03%
	Ochiai2	13	13	33	33	46	46	86.83	86.80	0.04%
	Kulczynski	14	14	34	34	47	47	86.29	86.27	0.03%
	Dstar2	21	22	42	42	59	59	74.77	74.74	0.04%
	Op2	10	10	32	32	43	43	87.76	87.75	0.01%
Mockito	Tarantula	7	7	22	22	31	31	34.89	34.84	0.15%
	SBI	7	7	22	22	31	31	34.89	34.84	0.15%
	Ochiai	9	9	20	20	32	32	36.66	36.58	0.22%
	Jaccard	7	7	22	22	30	30	34.42	34.39	0.08%
	Ochiai2	7	7	22	22	30	30	35.24	35.24	0.00%
	Kulczynski	7	7	22	22	30	30	34.42	34.39	0.08%
	Dstar2	8	8	19	19	33	33	44.34	44.39	-0.12%
	Op2	5	5	16	16	23	23	154.03	154.08	-0.03%
Overall	Tarantula	98	98	205	204	251	252	28.58	28.57	0.03%
	SBI	97	97	205	204	251	252	28.58	28.57	0.03%
	Ochiai	110	110	212	211	274	274	25.70	25.73	-0.11%
	Jaccard	99	101	204	205	254	255	27.83	27.84	-0.03%
	Ochiai2	98	96	207	207	253	254	28.25	28.13	0.41%
	Kulczynski	99	101	204	205	254	255	27.83	27.84	-0.03%
	Dstar2	109	110	209	206	269	266	27.00	27.28	-1.06%
	Op2	83	82	185	184	231	231	62.72	62.75	-0.05%

Test-based Reduction effectiveness

Legend deterioration improvement

Column **P** and **TR** indicate PRFL and PRFL with Test-based Reduction respectively

97%, while the *reducing rate* in project Closure is only 2.71%. Moreover, the right part of the table presents the results of the remaining number of the methods after applying the method-based reduction approach. It can be observed that the *reducing rate* of methods of all the projects are higher than their corresponding *reducing rates* of tests, ranging from 85% to 99%.

Time Efficiency. Reducing the size of the transition matrices is expected to reduce the computational cost of PRFL. Therefore, we collect the computation time for PRFL without applying the test and method reduction approaches to verify how exactly the computational cost can be optimized, as in Table 19. It can be shown that both the test and method reduction approaches result in shorter computation time than PRFL.

Table 19 shows the average computation time in each phase of PRFL, the test-based reduction approach, and the method-based reduction approach. Column `DP`, Column `Analysis` and Column `Ranking` represent data processing of PRFL(i.e., transition matrices construction), PageRank analysis, and suspiciousness computation/ranking, respectively. Since the time of test coverage and call graph col-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

TABLE 17: Results of Method-based Reduction

	Tech	Top-1		Top-3		Top-5		AWE		
		P	MR	P	MR	P	MR	P	MR	impr.
Chart	Tarantula	12	11	21	21	24	24	13.63	13.75	-0.85%
	SBI	11	10	21	21	24	24	13.63	13.75	-0.85%
	Ochiai	11	10	20	21	24	24	10.21	10.17	0.38%
	Jaccard	11	10	20	20	24	23	10.60	10.98	-3.63%
	Ochiai2	12	10	21	20	24	24	11.40	11.37	0.34%
	Kulczynski	11	10	20	20	24	23	10.60	10.98	-3.63%
	Dstar2	10	8	21	20	24	22	9.83	11.17	-13.70%
	Op2	7	7	17	17	20	20	61.37	61.33	0.06%
Lang	Tarantula	27	26	50	50	57	57	5.26	5.22	0.73%
	SBI	27	26	50	50	57	57	5.26	5.22	0.73%
	Ochiai	30	28	51	49	58	57	4.76	4.75	0.16%
	Jaccard	28	28	50	49	57	57	4.88	4.83	1.10%
	Ochiai2	28	25	50	49	57	57	5.33	5.07	4.91%
	Kulczynski	28	28	50	49	57	57	4.88	4.83	1.10%
	Dstar2	30	29	50	49	58	55	4.65	4.86	-4.46%
	Op2	29	28	48	48	56	56	4.61	4.69	-1.84%
Math	Tarantula	32	34	65	68	78	79	7.36	7.16	2.76%
	SBI	32	34	65	68	78	79	7.36	7.16	2.76%
	Ochiai	33	34	65	65	83	82	6.92	6.87	0.75%
	Jaccard	32	34	66	69	79	80	7.33	7.11	2.90%
	Ochiai2	31	33	67	70	79	79	7.35	7.16	2.69%
	Kulczynski	32	34	66	69	79	80	7.33	7.11	2.90%
	Dstar2	33	34	65	65	83	81	6.91	6.96	-0.75%
	Op2	28	29	61	61	76	73	8.35	8.55	-2.32%
Time	Tarantula	7	6	14	14	16	16	23.30	23.33	-0.16%
	SBI	7	6	14	14	16	16	23.30	23.33	-0.16%
	Ochiai	7	6	13	12	18	17	19.93	20.04	-0.56%
	Jaccard	7	6	12	12	17	17	23.44	23.37	0.32%
	Ochiai2	7	6	14	14	17	17	23.33	23.19	0.63%
	Kulczynski	7	6	12	12	17	17	23.44	23.37	0.32%
	Dstar2	7	6	12	11	12	12	21.48	21.89	-1.90%
	Op2	4	4	11	11	13	13	60.19	60.19	0.00%
Closure	Tarantula	13	14	33	35	45	46	87.01	86.52	0.56%
	SBI	13	14	33	35	45	46	87.02	86.54	0.56%
	Ochiai	20	23	43	44	59	59	75.74	75.65	0.11%
	Jaccard	14	15	34	37	47	47	86.29	85.69	0.70%
	Ochiai2	13	14	33	37	46	46	86.83	86.28	0.64%
	Kulczynski	14	15	34	37	47	47	86.29	85.69	0.70%
	Dstar2	21	25	42	43	59	58	74.77	74.65	0.17%
	Op2	10	10	32	31	43	42	87.76	87.58	0.20%
Mockito	Tarantula	7	7	22	22	31	31	34.89	34.88	0.04%
	SBI	7	7	22	22	31	31	34.89	34.88	0.04%
	Ochiai	9	9	20	20	32	31	36.66	36.67	-0.04%
	Jaccard	7	7	22	22	30	30	34.42	34.36	0.19%
	Ochiai2	7	7	22	22	30	30	35.24	35.22	0.04%
	Kulczynski	7	7	22	22	30	30	34.42	34.36	0.19%
	Dstar2	8	8	19	19	33	32	44.34	44.43	-0.21%
	Op2	5	5	16	16	23	23	154.03	154.01	0.01%
Overall	Tarantula	98	98	205	210	251	253	28.58	28.48	0.34%
	SBI	97	97	205	210	251	253	28.58	28.48	0.34%
	Ochiai	110	110	212	211	274	270	25.70	25.69	0.04%
	Jaccard	99	100	204	209	254	254	27.83	27.72	0.38%
	Ochiai2	98	95	207	212	253	253	28.25	28.05	0.72%
	Kulczynski	99	100	204	209	254	254	27.83	27.72	0.38%
	Dstar2	109	110	209	207	269	260	27.00	27.33	-1.22%
	Op2	83	83	185	184	231	227	62.72	62.72	-0.01%

Method-based Reduction effectiveness

Legend deterioration improvement

Column **P** and **MR** indicate PRFL and PRFL with Method-based Reduction respectively

TABLE 18: Average Test(Left) and Method(Right) Number after Reduction

Test-based Reduction				Method-based Reduction			
Project	Initial	Final	Reducing Rate	Project	Initial	Final	Reducing Rate
Chart	1821	628	65.50%	Chart	4473	103	97.70%
Lang	1856	55	97.00%	Lang	1994	10	99.50%
Math	2871	1064	63.00%	Math	3871	43	98.89%
Time	3924	3432	12.50%	Time	3442	122	96.46%
Closure	7187	6993	2.71%	Closure	7260	780	89.26%
Mockito	1136	1113	2.03%	Mockito	1171	169	85.57%

lection is not affected by test and method reduction, we do not present it in Table 19 and the readers can find it in Table 6(Column `COV` and `CG`). The main differences are presented in Column `DP` and Column `Analysis`. "-" represents the time taken by this phase is negligible (far below 0.01s). It can be shown that both test- and method-based reduction approaches are timely efficient compared with PRFL. Specifically, the projects Chart, Lang, and Math using the test-based reduction approach shorten less than 2.5% computation time of the original PRFL. The method-based reduction approach can shorten more computation time, resulting in around 40% and 27% computation time of the original PRFL for project Closure and Time.

6.8 RQ8: Effectiveness of $PRFL_{MA}$

In order to study the effectiveness of $PRFL_{MA}$, we add an additional experiment to study the performance of MA combined with naive SBFL, namely $SBFL_{MA}$. Table 20 shows the impact of SBFL, PRFL, $SBFL_{MA}$ and $PRFL_{MA}$ for 8 SBFL formulas.

From Table 20, the overall results suggest that $SBFL_{MA}$ performs much better than the naive SBFL, further conforming the study results of prior work [33]. For instance, on Closure, $SBFL_{MA}$ can locate 33 faults within Top-1 while the naive SBFL can only locate 14 faults within Top-1 by using Dstar2.

We also observe that $PRFL_{MA}$ can locate even more faults compared with PRFL and $SBFL_{MA}$. $PRFL_{MA}$ successfully localizes 137 and 133 at Top-1, with Ochiai and Dstar2 as the overall two most effective SBFL techniques. Compared to the state-of-the-art SBFL techniques, $PRFL_{MA}$ can rank 73.4%, 49.2%, and 31.3% more faults at Top-1, Top-3 and Top-5, respectively.

TABLE 19: Computation Time of PRFL, Test-based Reduction and Method-based Reduction

Tech	Sub	DP	Analysis	Ranking	Total	Time Radio
PRFL	Chart	6.49	1.33	0.01	109.72	100.00%
	Closure	888.87	6.61	0.01	1558.95	100.00%
	Lang	0.81	0.38	0.01	47.31	100.00%
	Math	8.33	1.47	0.01	384.34	100.00%
	Mockito	4.30	0.15	0.01	88.76	100.00%
	Time	19.39	1.32	0.01	68.00	100.00%
Test-based	Chart	2.52	0.45	0.01	104.87	95.58%
	Closure	835.75	6.56	0.01	1505.76	96.59%
	Lang	0.01	-	0.01	46.133	97.51%
	Math	3.22	0.50	0.01	378.26	98.42%
	Mockito	4.25	0.14	0.01	88.70	99.93%
	Time	17.4	1.14	0.01	65.83	96.80%
Method-based	Chart	0.01	0.01	0.01	102.09	93.04%
	Closure	279.12	0.46	0.01	943.03	60.49%
	Lang	0.01	-	0.01	46.13	97.51%
	Math	0.18	0.01	0.01	374.73	97.49%
	Mockito	1.49	0.02	0.01	85.82	96.68%
	Time	2.78	0.03	0.01	50.10	73.68%

6.9 RQ9: Impact of Fault Type

So far we have studied the effectiveness of PRFL, $SBFL_{MA}$ and $PRFL_{MA}$ on real faults from the Defects4J dataset. In this section, we further study the effectiveness of above techniques on artificial mutation faults from other projects to reduce the threats to external validity.

Table 21 presents our results on 96925 mutation faults from 240 GitHub projects. In the table, different rows present the results of different fault localization formulas, while different columns present the different metrics used.

According to the table, we find that most effective techniques of SBFL are the same with those for single real faults on Defects4J (shown in Table 9), i.e., Dstar2 and Op2 (marked in gray in the table), demonstrating the consistent result on single real and mutation faults (Note that mutation faults are all single faults since PIT only makes one syntactic modification for each mutant).

The most effective techniques of PRFL on artificial mutation faults are Ochiai and Dstar2, which are the same with those on Defects4J. PRFL can locate 27378, 49253 and 59547 faults within Top-1/3/5. Compared to the state-of-the-art SBFL techniques, PRFL can rank 82.3% (24613 vs. 13505) more faults at Top-1, and reduce the AWE values from 7.59 to 6.82 (10.1% more precise) on Op2.

PRFL performs better than $SBFL_{MA}$ on artificial mutation faults at Top-1, 3 and 5. However, it is worth noting that the AWE values of $SBFL_{MA}$ is lower than PRFL. According to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

TABLE 20: Result of SBFL,PRFL,$SBFL_{MA}$ and $PRFL_{MA}$ on all Defects4J bugs

	Tech	SBFL				PRFL				$SBFL_{MA}$				$PRFL_{MA}$			
		Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE
Chart	Tarantula	7	20	22	14.92	12	21	24	13.63	7	23	25	10.10	12	24	27	9.98
	SBI	7	20	22	14.92	11	21	24	13.63	7	23	25	10.10	11	24	27	9.98
	Ochiai	6	17	19	11.83	11	20	24	10.21	7	19	21	9.37	12	23	26	7.52
	Jaccard	6	17	20	12.12	11	20	24	10.60	7	19	22	9.02	12	22	25	7.37
	Ochiai2	6	17	21	12.15	12	21	24	11.40	7	19	23	8.37	12	24	27	7.02
	Kulczynski	6	17	20	12.12	11	20	24	10.60	7	19	22	9.02	12	22	25	7.37
	Dstar2	5	16	19	13.65	10	21	24	9.83	6	18	21	11.25	11	23	27	7.52
	Op2	5	14	16	66.19	7	17	20	61.37	6	16	19	62.90	7	16	20	60.67
Lang	Tarantula	21	45	57	5.81	27	49	57	5.26	26	49	59	4.28	29	53	61	4.11
	SBI	21	45	57	5.81	27	49	57	5.26	26	49	59	4.28	29	53	61	4.12
	Ochiai	22	44	56	5.35	30	50	58	4.76	28	48	58	3.85	34	54	61	3.55
	Jaccard	22	44	56	5.39	28	49	57	4.88	28	47	58	3.95	32	53	61	3.72
	Ochiai2	21	45	55	5.35	28	49	57	5.33	27	48	57	3.86	32	53	61	3.96
	Kulczynski	22	44	56	5.39	28	49	57	4.88	28	47	58	3.95	32	53	61	3.72
	Dstar2	23	45	55	5.33	30	49	58	4.65	29	49	55	4.02	34	53	61	3.44
	Op2	23	45	56	5.48	29	47	56	4.61	29	46	56	4.15	36	48	56	3.76
Math	Tarantula	24	61	73	8.89	32	65	78	7.36	30	79	92	5.25	38	80	90	5.09
	SBI	24	61	73	8.87	32	65	78	7.36	30	79	92	5.25	37	80	90	5.09
	Ochiai	24	60	73	9.25	33	65	83	6.92	31	74	92	5.17	40	83	90	4.63
	Jaccard	24	61	73	8.87	32	66	79	7.33	32	75	92	5.28	38	79	91	5.11
	Ochiai2	24	61	73	8.88	31	67	79	7.35	30	77	92	5.27	38	81	91	5.09
	Kulczynski	24	61	73	8.87	32	66	79	7.33	32	75	92	5.28	38	79	91	5.11
	Dstar2	24	60	73	9.40	33	65	83	6.91	31	74	91	5.24	40	82	90	4.64
	Op2	23	54	65	10.66	28	61	76	8.35	29	67	80	7.79	33	62	77	7.99
Time	Tarantula	5	11	16	25.37	7	14	16	23.30	6	14	21	15.57	7	17	20	14.30
	SBI	5	11	16	25.37	7	14	16	23.30	6	14	21	15.57	7	17	20	14.30
	Ochiai	6	11	18	22.93	7	13	18	19.93	7	14	18	15.06	8	17	21	13.33
	Jaccard	5	9	18	25.41	7	12	17	23.44	7	13	20	15.83	8	16	21	14.33
	Ochiai2	5	11	16	25.44	7	14	17	23.33	7	15	22	15.43	8	17	21	14.44
	Kulczynski	5	9	18	25.41	7	12	17	23.44	7	13	20	15.83	8	16	21	14.33
	Dstar2	6	11	12	25.22	7	12	12	21.48	7	14	16	18.39	7	14	16	16.00
	Op2	8	12	14	62.67	4	11	13	60.19	7	13	15	63.13	4	11	11	63.56
Closure	Tarantula	12	26	36	120.30	13	33	45	87.01	27	57	66	69.44	28	58	65	69.32
	SBI	12	26	36	120.30	13	33	45	87.02	27	57	66	69.44	28	58	65	69.32
	Ochiai	14	29	39	108.70	20	43	59	75.74	31	64	70	62.17	32	63	69	62.04
	Jaccard	13	27	37	119.40	14	34	47	86.29	30	62	68	68.67	31	61	67	68.54
	Ochiai2	13	26	37	120.01	13	33	46	86.83	30	61	68	69.10	30	60	67	68.94
	Kulczynski	13	27	37	119.40	14	34	47	86.29	30	62	68	68.67	31	61	67	68.54
	Dstar2	14	28	39	108.23	21	42	59	74.77	33	63	70	61.72	32	62	69	61.65
	Op2	17	33	42	119.26	10	32	43	87.76	32	59	61	124.32	32	59	61	124.19
Mockito	Tarantula	6	18	25	39.51	7	22	31	34.89	10	26	29	25.07	12	26	31	22.01
	SBI	6	18	25	39.51	7	22	31	34.89	10	26	29	25.07	12	26	31	22.01
	Ochiai	7	18	25	49.91	9	20	32	36.66	11	26	30	38.12	11	27	35	28.25
	Jaccard	6	18	25	39.32	7	22	30	34.42	10	26	30	25.92	12	26	31	22.36
	Ochiai2	6	18	25	39.46	7	22	30	35.24	10	26	29	25.33	12	26	31	22.51
	Kulczynski	6	18	25	39.32	7	22	30	34.42	10	26	30	25.92	12	26	31	22.36
	Dstar2	7	17	23	57.96	8	19	33	44.34	10	25	28	47.38	9	26	36	37.25
	Op2	6	14	19	155.43	5	16	23	154.03	10	19	23	151.86	7	19	25	151.96
Overall	Tarantula	75	181	229	35.80	98	204	251	28.58	106	248	292	21.62	126	258	294	20.80
	SBI	75	181	229	35.80	97	204	251	28.58	106	248	292	21.62	124	258	294	20.80
	Ochiai	79	179	230	34.66	110	211	274	25.70	115	245	289	22.29	137	267	302	19.89
	Jaccard	76	176	229	35.08	99	203	254	27.83	114	242	290	21.45	133	257	296	20.24
	Ochiai2	75	178	227	35.21	98	206	253	28.25	111	246	291	21.22	132	261	298	20.33
	Kulczynski	76	176	229	35.08	99	203	254	27.83	114	242	290	21.45	133	257	296	20.24
	Dstar2	79	177	221	36.63	109	208	269	27.00	116	243	281	24.67	133	260	299	21.75
	Op2	82	172	212	69.95	83	184	231	62.72	113	220	254	69.02	119	215	250	68.69

Section 4.3, we infer that MA enables the faulty method to have a higher SBFL score than other methods in most cases.

$PRFL_{MA}$ can locate more faults compared with PRFL and $SBFL_{MA}$ on artificial mutation faults, with Ochiai and Ochiai2 as the two most effective SBFL techniques. $PRFL_{MA}$ successfully locates 35058 faults within Top-1, that is 36% of total mutation faults. Compared with PRFL and $SBFL_{MA}$, $PRFL_{MA}$ can further rank 28% (35058 vs. 27231) and 87% (35058 vs. 18703) more faults at Top-1.

6.10 RQ10: Statistical Significance

To further validate the effectiveness of the proposed approaches, we use paired *t*-test [60], [61] to investigate the statistical significance of the experimental results. Specifically, we regard each subject in Defects4J dataset as an individual item, and group its experimental results (e.g. Top 1 scores) of two different techniques (e.g. SBFL and PRFL) as a pair. Next, the paired *t*-test is applied on such pairs to compute the *p*-values to analyze the significance of difference between the two approaches.

We present the results of paired t-test in Table 22. From the table, it can be observed that the *p*-values of most ranking formulae are less than 0.05 of Top 1, 3 and 5 (except Op2), and these results suggest the statistical significance that PRFL and Method-level Aggregation can boost most SBFL techniques in term of the accuracy of fault localization on Top N.

7 THREATS TO VALIDITY

The main threats to the validity of our work are consisted of three parts: internal, external, and construct validity threat.

Threats to internal validity are mainly concerned with uncontrolled factors. These factors may impact the results and reduce their creditability. In this work, the main threat to internal validity is potential defects in the implementation of our own technique and the reimplementation of other

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

TABLE 21: Fault localization Results on Mutation Faults

Tech	SBFL				PRFL				$SBFL_{MA}$				$PRFL_{MA}$			
	Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE	Top1	Top3	Top5	AWE
Tarantula	12026	34891	45460	10.61	23327	45491	55044	9.35	18688	44365	55546	6.00	32561	55053	64482	4.95
SBI	10871	32704	43549	10.76	19728	42150	52263	9.64	16613	41885	52625	6.15	27269	51057	61594	5.23
Ochiai	13113	37500	49096	8.05	27231	48953	59146	6.76	18671	44387	55568	6.00	34960	55460	63181	5.03
Jaccard	13072	37334	48919	8.35	27088	48619	58863	7.07	18671	44387	55568	6.00	34928	55442	63121	5.05
Ochiai2	12976	36804	48166	9.40	26389	47381	57215	8.51	18703	44387	55568	6.00	35058	55544	63263	4.98
Kulczynski	13072	37336	48919	8.35	27088	48619	58863	7.07	18671	44387	55568	6.00	34928	55442	63121	5.05
Dstar2	13179	37667	49329	7.92	27378	49253	59547	6.57	18671	44387	55568	6.00	34642	55249	62873	5.10
Op2	13505	38288	49915	7.59	24613	45940	56298	6.82	18228	43784	54929	6.02	23970	42743	50323	6.08

TABLE 22: Paired *t*-test Results of Different Techniques

Comp.	Tech.	Top 1	Top 3	Top 5	AWE
SBFL vs PRFL	Tarantula	0.0119*	0.0024♦	0.0278*	0.1134
	SBI	0.0131*	0.0024♦	0.0278*	0.1135
	Ochiai	0.0053♦	0.0174*	0.0268*	0.0715
	Jaccard	0.0119*	0.0004◊	0.0230*	0.1111
	Ochiai2	0.0160*	0.0003◊	0.0077♦	0.1219
	Kulczynski	0.0119*	0.0004◊	0.0230*	0.1110
	Dstar2	0.0073♦	0.0204*	0.0197*	0.0590
	Op2	0.4697	0.0798	0.0675	0.0994
PRFL vs $PRFL_{MA}$	Tarantula	0.0494*	0.0300*	0.0324*	0.0171*
	SBI	0.0534	0.0300*	0.0324*	0.0171*
	Ochiai	0.0257*	0.0151*	0.0075♦	0.0151*
	Jaccard	0.0330*	0.0354*	0.0360*	0.0182*
	Ochiai2	0.0363*	0.0343*	0.0303*	0.0153*
	Kulczynski	0.0330*	0.0354*	0.0360*	0.0182*
	Dstar2	0.0356*	0.0216*	0.0041♦	0.0172*
	Op2	0.0688	0.1466	0.1709	0.1876
$SBFL_{MA}$ vs $PRFL_{MA}$	Tarantula	0.0153*	0.0211*	0.3397	0.0767
	SBI	0.0117*	0.0211*	0.3397	0.0771
	Ochiai	0.0265*	0.0265*	0.0683	0.0878
	Jaccard	0.0075♦	0.0321*	0.1146	0.0393*
	Ochiai2	0.0181*	0.0321*	0.1509	0.0500*
	Kulczynski	0.0075♦	0.0321*	0.1146	0.0393*
	Dstar2	0.0764	0.0495*	0.0665	0.0591
	Op2	0.2812	0.2171	0.2580	0.2167

* $p < 0.05$, ♦ $p < 0.01$, ◊ $p < 0.001$

baselines. To reduce this threat, we utilize state-of-the-art tools and frameworks, such as Java Agent, ASM bytecode manipulation framework and Numpy library to build our technique. Moreover, we also reimplement other baseline techniques by following their papers. We carefully review all our code and experiment scripts to ensure their correctness. However, there is always a small possibility of defects, which introduces risk to the result's correctness.

Threats to external validity are mainly concerned with whether the performance of our techniques can still hold in other experimental settings. In our work, the subjects (including tests and faults) may introduce threats to external validity. To reduce these threats, we use the real-word subjects from Defects4J and Bugs.jar dataset to evaluate our approach, respectively. However, the total subject number is relatively small and it may limit the generalization of our technique. We further utilize a bunch of artificial faults to evaluate the effectiveness of our approach. However, the artificial faults are generated by mutation testing tool, and this will make the diversity of artificial faults limited. Therefore, although the experimental results of artificial faults have low variance (due to the large number of mutants), they may be biased. To further reduce the threats, we are planning to collect more real-world subjects to evaluate our technique.

One of the threats to construct validity is the suitability of our evaluation metrics. To reduce this risk, we follow the suggestions of Parnin and Orso [62]. Specifically, first, instead of locating faulty statement, we focus on method-level fault localization since existing work has demonstrated that this is more practical in the general development [58]. For example, suppose a faulty constructor method misses some field-initialization statements, statement-level fault localization even does not have a ground truth since such field-initialization statement can be added anywhere within the constructor, but method-level fault localization can still work. Second, instead of using applying percentage, we use Top-N and AWE, two absolute-rank-based metrics to evaluate the effectiveness of our approaches. However, as Parnin and Orso [62] argued, fault ranking should be more specific under different scenarios, including absolute ranking, search rank, etc. On the other hand, fault localization techniques should also provide more information to developer help them understanding the faults instead of just the locations. Honestly, these two points are not fully considered in our approach. In the further work, we plan to conduct user study to further investigate the effectiveness and practicality of our approach in daily development.

8 Related Work

To the best of our knowledge, this paper is the first extensive study on improving SBFL using PageRank and Method-level Aggregation. We list the related work in fault localization as follows.

Spectrum-Based Fault Localization Various formulas have been proposed for computing suspiciousness scores of program entities based on passing and failing test cases. Jones et al. [3] proposed the first foundational ranking formula, Tarantula, which is based on the intuition that program entries which are frequently executed by failing test cases and infrequently executed by passing test cases are more likely to be faulty. Dallmeier et al. [5] proposed Ample, an Eclipse plug-in for identifying faulty classes in Java software. Abreu et al. [4] designed Ochiai, which is also widely-studied and state-of-the-art ranking formula. Naish et al. [6] proposed the theoretical-best ranking formulas for single faults, Op and Op2, and empirically analyzed the existing ranking formulas on C programs. Yoo [7] generated a group of ranking formulas using genetic programming (GP). Xie et al. [8] summarized existing ranking formulas and theoretically compared them, finding that formulas of two families are optimal, including Op, Op2 and four GP-generated formulas. Lucia et al. [17] investigated the effectiveness of existing work and concluded that there is no best single ranking formula for all cases. Similarly, Steimann et al. [63] studied the threats to the validity in SBFL on ten open-source programs and showed that well-known fault locators do not uniformly perform better.

Machine-Learning-Based Fault Localization Several existing work has applied machine learning techniques to improve the accuracy of SBFL. Abreu et al. [64] proposed Barinel, in which Bayesian reasoning is used to deduce

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2911283, IEEE Transactions on Software Engineering

multiple-fault candidates and their probabilities. Nath et al. [65] proposed TFLMs, a Relational Sum-Product Network model for fault localization. TFLMs can be learned from a corpus of faulty programs and localizes faults in a new context more accurately. Feng et al. [11] proposed Error Flow Graph (EFG), a Bayesian Network to predict fault locations. EFG is constructed from the dynamic dependency graphs of the programs and then standard inference algorithms are employed to compute the probability of each executed statement being faulty. Xuan et al. [12] proposed Multric, which applied RankBoost, a pairwise learning-to-rank algorithm to combine 25 existing formulas. Roychowdhury et al. [66] utilized feature selection for fault localization and Le et al. [67] extended a standard feature selection to identify program entities. Recently, Sohn et al. [33] and Li et al. [16] combined SBFL with source code metrics and mutation information, respectively, for learning-to-rank based fault localization. Perez et al. [68], [69] proposed Q-SFL, which leverages qualitative reasoning to augment the information made available to improve SBFL techniques. Actually, our work can be treated as an unsupervised-learning-based fault localization technique. Different from most existing learning-based work, our work does not require training data which can be hard to collect.

Mutation-Based Fault Localization Besides spectrum-based and machine-learning-based fault localization, there is one category of approach utilizing mutation analysis [30], [70], [71]. Papadakis et al. [15] firstly applied mutation testing to traditional fault localization. Zhang et al. [14] firstly applied mutation testing to localize faults during regression testing. Later on, Moon et al. [13] proposed MUSE, a mutation-based fault localization technique by analyzing mutant impacts on faulty and correct program entities. Hong et al. [72] developed new mutation operators as well as traditional operators to improve fault localization in real-world multilingual programs. There are also empirical studies evaluating mutation-based fault localization techniques [16], [18], [73]. More specifically, recently, Li et al. [16] demonstrated that mutation-based fault localization can be effective for localizing real-world bugs at the method level.

Slicing-Based Fault Localization Slicing technique is also widely used in fault localization [74], [75], [76]. Zhang et al. [77] proposed a forward and a bidirectional dynamic slicing techniques for improving fault localization. Alves et al. [78] used dynamic slicing technique and change-impact analysis to prune irrelative code statements to improve Tarantula [3]. Sinha et al. [79] focused on the fault localization of Java Runtime Exceptions. They combined dynamic analysis and static backward data-flow analysis to detect source statements which lead to exceptions. Xuan et al. [10] proposed to use program slicing to trim test cases into minimal fractions to achieve more precise SBFL. Gupta et al. [80] combined delta debugging which can identify a minimal failure-inducing input with forward and backward dynamic program slicing to narrow down probably faulty code for improving fault localization. Ocariza et al. [81] also proposed an automated technique to improve fault localization for JavaScript code via backward slicing.

Other Fault Localization Techniques Similar to mutation-based fault localization, Jeffrey et al. [82] proposed a value-profile-based approach for ranking program statements according to their likelihood of being faulty. Campos et al. [83] applied entropy theory in fitness function to generate new test cases in order to improve fault localization. Alipour et al. [84] extracted extended invariants such as execution features to improve fault localization. Zhang et al. [85] identified the causes of faults by switching predicates' outcome at runtime and altering the control flow. Yu et al. [86] introduced multiple kinds of spectrum types such as control and data dependences to build fault localization model. Le et al. [87] and Dao et al. [88] combined SBFL with information-retrieval-based fault localization, which recommends a set of program entities with similar contents of bug reports. Perez et al. [89] proposed a metric, DDU, which applies a quantified test-suite's diagnosability to SBFL to improve the effectiveness of fault localization. Gonzalez-Sanchez et al. [90] proposed RAPTOR , a test prioritization algorithm to improve fault localization, based on reducing the similarity between statement execution patterns. Le et al. [47] employed likely invariants and suspiciousness scores to locate faults. In this work, they used Daikon's Invariant Diff [91] tool to mine changes in invariant sets between failing and passing program executions, and applied learning-to-rank algorithm to predict fault locations.

9 Conclusion

Manual debugging remains costly and painful. Researchers have developed various techniques to automate debugging. A particularly well-studied class of techniques is spectrum-based fault localization (SBFL), which help developers infer the positions of faulty program entities. Despite the research progress, current SBFL techniques are not precise enough for practical debugging. In this paper, we propose PRFL, a novel approach to boost the accuracy of spectrum-based fault localization. PRFL uses PageRank algorithm to analyze the importance of each test, and then ranks methods by considering the corresponding test importance. We further extend the studies on PRFL by various aspects for effectiveness and performance optimization and propose PRFL_{MA} that integrates the statement-level coverage. We evaluate our approach on 395 real faults and 96925 mutation faults. The experimental results showed that PRFL and PRFL_{MA} outperforms existing state-of-the-art SBFL techniques significantly (e.g., ranking 39.2%/82.3% and 73.4%/159.6% more real/artificial faults within Top-1 compared with the most effective traditional SBFL technique, respectively), with low overhead.

References

[1] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," *NIST*, vol. 7007, no. 011, 2002.
[2] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *TSE*, vol. 42, no. 8, pp. 707–740, 2016.
[3] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE*. ACM, 2002, pp. 467–477.
[4] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *TAICPART-MUTATION*. IEEE, 2007, pp. 89–98.
[5] V. Dallmeier, C. Lindig, and A. Zeller, "Lightweight bug localization with ample," in *AADEBUG*. ACM, 2005, pp. 99–104.
[6] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *TOSEM*, vol. 20, no. 3, p. 11, 2011.

[7] S. Yoo, "Evolving human competitive spectra-based fault localization techniques," in *SSBSE*. Springer, 2012, pp. 244–258.

[8] X. Xie, F.-C. Kuo, T. Y. Chen, S. Yoo, and M. Harman, "Provably optimal and human-competitive results in sbse for spectrum based fault localization," in *SSBSE*. Springer, 2013, pp. 224–238.

[9] X. Mao, Y. Lei, Z. Dai, Y. Qi, and C. Wang, "Slice-based statistical fault localization," *Journal of Systems and Software*, vol. 89, pp. 51–62, 2014.

[10] J. Xuan and M. Monperrus, "Test case purification for improving fault localization," in *FSE*. ACM, 2014, pp. 52–63.

[11] M. Feng and R. Gupta, "Learning universal probabilistic models for fault localization," in *PASTE*. ACM, 2010, pp. 81–88.

[12] J. Xuan and M. Monperrus, "Learning to combine multiple ranking metrics for fault localization," in *ICSME*. IEEE, 2014, pp. 191–200.

[13] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *ICST*. IEEE, 2014, pp. 153–162.

[14] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *OOPSLA*, 2013, pp. 765–784.

[15] M. Papadakis and Y. Le Traon, "Using mutants to locate "unknown" faults," in *ICST*. IEEE, 2012, pp. 691–700.

[16] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," in *OOPSLA*. ACM, 2017, pp. 92–122.

[17] L. Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of Software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, 2014.

[18] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *ICSE*. IEEE, 2017, pp. 609–620.

[19] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *TSE*, vol. 38, no. 1, pp. 54–72, 2012.

[20] J. Xuan, M. Martinez, F. Demarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *TSE*, vol. 43, no. 1, pp. 34–55, 2017.

[21] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus, "Automatic repair of buggy if conditions and missing preconditions with smt," in *CSTVA*. ACM, 2014, pp. 30–39.

[22] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *FSE*. ACM, 2015, pp. 166–178.

[23] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *ICSE*. ACM, 2014, pp. 254–265.

[24] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, pp. 1–29, 2016.

[25] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *ISSTA*. ACM, 2017, pp. 261–272.

[26] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[27] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *ISSTA*. ACM, 2014, pp. 437–440.

[28] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *ICSE*. ACM, 2005, pp. 402–411.

[29] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *FSE*. ACM, 2014, pp. 654–665.

[30] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *STVR*, vol. 25, no. 5-7, pp. 605–628, 2015.

[31] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 604–632, 1999.

[32] R. Lempel and S. Moran, "The stochastic approach for link-structure analysis (salsa) and the tkc effect1," *Computer Networks*, vol. 33, no. 1-6, pp. 387–401, 2000.

[33] J. Sohn and S. Yoo, "Fluccs: using code and change metrics to improve fault localization," in *ISSTA*. ACM, 2017, pp. 273–283.

[34] R. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. Prasad, "Bugs. jar: a large-scale, diverse dataset of real-world java bugs," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 10–13.

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

[36] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 15–26, 2005.

[37] A. Borodin, G. O. Roberts, J. S. Rosenthal, and P. Tsaparas, "Link analysis ranking: algorithms, theory, and experiments," *ACM Transactions on Internet Technology (TOIT)*, vol. 5, no. 1, pp. 231–297, 2005.

[38] A. Farahat, T. LoFaro, J. C. Miller, G. Rae, and L. A. Ward, "Authority rankings from hits, pagerank, and salsa: Existence, uniqueness, and effect of initialization," *SIAM Journal on Scientific Computing*, vol. 27, no. 4, pp. 1181–1201, 2006.

[39] D. F. Gleich, "Pagerank beyond the web," *SIAM Review*, vol. 57, no. 3, pp. 321–363, 2015.

[40] A. D. Chepelianskii, "Towards physical laws for software architecture," *arXiv preprint arXiv:1003.5455*, 2010.

[41] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture," in *PER*, vol. 41, no. 1. ACM, 2013, pp. 93–104.

[42] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos, "Graph-based analysis and prediction for software evolution," in *ICSE*, 2012, pp. 419–429.

[43] S. Mirshokraie, A. Mesbah, and K. Pattabiraman, "Guided mutation testing for javascript web applications," *TSE*, vol. 41, no. 5, pp. 429–444, 2015.

[44] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *ECOOP*. Springer, 1995, pp. 77–101.

[45] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *TOPLAS*, vol. 23, no. 6, pp. 685–746, 2001.

[46] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan, "An empirical study of static call graph extractors," *TOSEM*, vol. 7, no. 2, pp. 158–191, 1998.

[47] T.-D. B Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *ISSTA*. ACM, 2016, pp. 177–188.

[48] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *ASE*. IEEE, 2015, pp. 201–211.

[49] G. Laghari, A. Murgia, and S. Demeyer, "Fine-tuning spectrum based fault localisation with frequent method item sets," in *ASE*. ACM, 2016, pp. 274–285.

[50] "Pit mutation testing tool," 2018, accessed: 02-10-2018. [Online]. Available: http://pitest.org/

[51] "Github a web-based version control repository and internet hosting service," 2018, accessed: 02-10-2018. [Online]. Available: https://github.com/

[52] "Asm java bytecode manipulation and analysis framework," 2018, accessed: 02-10-2018. [Online]. Available: http://asm.ow2.org/

[53] "Java programming language agents," 2018, accessed: 02-10-2018. [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html

[54] "Numpy package for scientific computing with python," 2018, accessed: 02-10-2018. [Online]. Available: http://www.numpy.org/

[55] "Maven central repository," 2018, accessed: 02-10-2018. [Online]. Available: https://search.maven.org/

[56] "Maven surefire plugin," 2018, accessed: 02-10-2018. [Online]. Available: http://maven.apache.org/surefire/maven-surefire-plugin/

[57] "Maven failsafe plugin," 2018, accessed: 02-10-2018. [Online]. Available: http://maven.apache.org/surefire/maven-failsafe-plugin/

[58] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *ISSTA*, 2016, pp. 165–176.

[59] A. Perez, R. Abreu, and M. d'Amorim, "Prevalence of single-fault fixes and its impact on fault localization," in *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. IEEE, 2017, pp. 12–22.

[60] H. Hsu and P. A. Lachenbruch, "Paired t test," *Wiley encyclopedia of clinical trials*, pp. 1–3, 2007.

[61] B. Rosner, "A generalization of the paired t-test," *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 31, no. 1, pp. 9–13, 1982.

[62] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *ISSTA*. ACM, 2011, pp. 199–209.

[63] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *ISSTA*. ACM, 2013, pp. 314–324.

[64] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "Spectrum-based multiple fault localization," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 88–99.

[65] A. Nath and P. Domingos, "Learning tractable probabilistic models for fault localization," in *AAAI*. AAAI Press, 2016, pp. 1294–1301.

[66] S. Roychowdhury and S. Khurshid, "Software fault localization using feature selection," in *MALETS*. ACM, 2011, pp. 11–18.

[67] T.-D. B. Le, D. Lo, and M. Li, "Constrained feature selection for localizing faults," in *ICSME*. IEEE, 2015, pp. 501–505.

[68] A. Perez, R. Abreu, and I.-T. HASLab, "Leveraging qualitative reasoning to improve sfl." in *IJCAI*, 2018, pp. 1935–1941.

[69] A. Perez and R. Abreu, "A qualitative reasoning approach to spectrum-based fault localization," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 2018, pp. 372–373.

[70] M. Papadakis and Y. Le Traon, "Effective fault localization via mutation analysis: A selective mutation approach," in *SAC*. ACM, 2014, pp. 1293–1300.

[71] M. Papadakis, M. E. Delamaro, and Y. Le Traon, "Proteum/fl: A tool for localizing faults using mutation analysis," in *SCAM*. IEEE, 2013, pp. 94–99.

[72] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim, "Mutation-based fault localization for real-world multilingual programs (t)," in *ASE*. IEEE, 2015, pp. 464–475.

[73] T. T. Chekam, M. Papadakis, and Y. L. Traon, "Assessing and comparing mutation-based fault localization techniques," *arXiv preprint arXiv:1607.05512*, 2016.

[74] X. Zhang, H. He, N. Gupta, and R. Gupta, "Experimental evaluation of using dynamic slices for fault location," in *AADEBUG*. ACM, 2005, pp. 33–42.

[75] B. Hofer and F. Wotawa, "Spectrum enhanced dynamic slicing for better fault localization." in *ECAI*, 2012, pp. 420–425.

[76] X. Zhang, N. Gupta, and R. Gupta, "A study of effectiveness of dynamic slicing in locating real faults," *Empirical Software Engineering*, vol. 12, no. 2, pp. 143–160, 2007.

[77] X. Zhang, N. Gupta, and R. Gupta, "Locating faulty code by multiple points slicing," *Software: Practice and Experience*, vol. 37, no. 9, pp. 935–961, 2007.

[78] E. Alves, M. Gligoric, V. Jagannath, and M. d'Amorim, "Fault-localization using dynamic slicing and change impact analysis," in *ASE*. IEEE, 2011, pp. 520–523.

[79] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault localization and repair for java runtime exceptions," in *ISSTA*. ACM, 2009, pp. 153–164.

[80] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *ASE*. ACM, 2005, pp. 263–272.

[81] F. S. Ocariza Jr, K. Pattabiraman, and A. Mesbah, "Autoflox: An automatic fault localizer for client-side javascript," in *ICST*. IEEE, 2012, pp. 31–40.

[82] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *ISSTA*. ACM, 2008, pp. 167–178.

[83] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim, "Entropy-based test generation for improved fault localization," in *ASE*. IEEE, 2013, pp. 257–267.

[84] M. A. Alipour and A. Groce, "Extended program invariants: applications in testing and fault localization," in *WODA*. ACM, 2012, pp. 7–11.

[85] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *ICSE*. ACM, 2006, pp. 272–281.

[86] K. Yu, M. Lin, Q. Gao, H. Zhang, and X. Zhang, "Locating faults using multiple spectra-specific models," in *SAC*. ACM, 2011, pp. 1404–1410.

[87] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: better together," in *FSE*. ACM, 2015, pp. 579–590.

[88] T. Dao, L. Zhang, and N. Meng, "How does execution information help with information-retrieval based bug localization?" in *ICPC*. IEEE, 2017, pp. 241–250.

[89] A. Perez, R. Abreu, and A. van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 654–664.

[90] A. Gonzalez-Sanchez, R. Abreu, H.-G. Gross, and A. J. van Gemund, "Prioritizing tests for fault localization through ambiguity group reduction," in *Proceedings of the 2011 26th IEEE/ACM international conference on automated software engineering*. IEEE Computer Society, 2011, pp. 83–92.

[91] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.