Use of Control Flow Graphs with Edges Consideration for Fault Localization

ANONYMOUS AUTHOR(S)

In the final stage of software development, debugging, especially fault localization, is one of the most time-consuming processes for both novice and senior software engineers. One way to locate the fault is by testing the code with different inputs and looking for statements that are predominantly covered by failing test cases. This process can be used to then calculate the suspiciousness of individual lines of code.

Previous work introduced Spectrum-Based Fault Localization (SBFL), a visualization technique to help developers identify suspicious lines of code. Here, we build on that work by displaying a Control Flow Graph (CFG) as a visualization technique for fault localization assistance, with the most suspicious line(s) of code highlighted to draw the programmer's attention, and other statements highlighted to indicate suspicious paths through the graph. Further, we include the consideration of edges in the graph while computing the suspiciousness of code lines, which helps localize faults that are in decision points in the CFG.

In this paper, we provide an overview of our CFG-based approach with some examples of visualizations that are easy for developers to understand, as well as an explanation of how we use edges in the graph to assess the suspiciousness of method decision points. We also describe the results of experiments using both 71 various valid real software faults from three different libraries provided by *Defects4J* benchmark and various Java methods from the Apache Commons Math library with mutation analysis, which demonstrates that our technique can be more effective at helping developers localize faults than previous approaches.

ACM Reference Format:
Anonymous Author(s). 2024. Use of Control Flow Graphs with Edges Consideration for Fault Localization. 1, 1 (November 2024), 17 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Locating faults during software development is a time-consuming and costly process that could involve extensive testing by trying many different inputs and analyzing their outputs, which can be very expensive in terms of both time and money [9]. Helping developers narrow their searches by selecting suspicious statements becomes important, where "suspicious" refers to code entities that are more likely to be responsible for observed failing test cases, i.e. those containing the fault [10].

To assist with fault localization, Jones et al. proposed a spectrum-based fault localization approach called Tarantula that displays lines of code in different colors based on the computed suspiciousness of the line [1, 3, 13, 18]. In 2006, Abreu et al. proposed Ochiai which was originally used in molecular biology, which was showed outperforming Tarantula on SBFL [2]. Despite the effectiveness of this both approaches [2, 14], we propose that it may be preferable to display the code as a Control Flow Graph (CFG) so that the programmer can more easily see the path of the code that is likely to contain a fault. The Control Flow Graph can clearly represent the code and indicate the suspiciousness of each line [16]. The line that has the highest level of suspiciousness is most likely to hold the programmer's attention, and the programmer can also see which code precedes and follows the most suspicious line in the path that is most associated with failing tests.

ACM XXXX-XXXX/2024/11-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Moreover, we can take advantage of the structure of the CFG by considering the suspiciousness of edges in addition to nodes/statements, which may improve the probability that the line with the fault will be top-ranked. We call this new modified approach "+Edges", specifically "Tarantula+Edges" for the modification for Tarantula and "Ochiai+Edges" for Ochiai.

The implementation of a Tarantula+Edges and Ochiai+Edges tool is still under development, but in this paper, we focus on describing the motivation for the approach and an empirical evaluation of the benefits of considering CFG edges for fault localization. The main contributions of this paper are:

(1) A presentation of a new technique for providing users with a visualization for fault localization using a Control Flow Graph. This allows users to locate the most suspicious line and also to spot neighboring lines of code for a better understanding of the location of the fault.
(2) A description of the +Edges approach, including Tarantula+Edges and Ochiai+Edges, which aims to improve the accuracy of locating the fault, especially for a piece of code that contains more decision points.
(3) A comparative real Java program analysis of the accuracy and effectiveness of three different techniques for locating the fault: Tarantula, Tarantula+Edges, and the Set-union technique [6].
(4) An empirical evaluation of Ochiai+Edges using 71 valid real word bugs from Defects4J, which shows a better performance compared with the Ochiai approach based on 4 different ranking metrics.

2 RELATED WORK

Different types of fault localization techniques have been proposed, implemented, and examined, such as spectrum-based fault localization (SBFL), model-based diagnosis, statistical-based and machine learning techniques, etc. However, for software debugging, SBFL is considered one of the most prominent techniques due to its low overhead performance cost [10] and ease of use [19][21].

2.1 Control-flow spectra

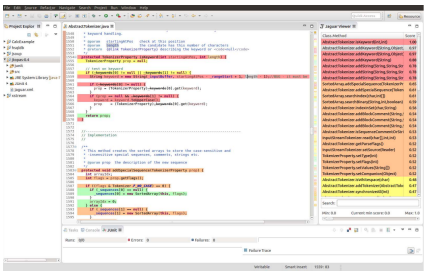
SBFL techniques use control-flow spectra to pinpoint the program entities that are more likely to cause the fault [18]. Control-flow spectra focus on the information that is represented by a graph with nodes and edges, where nodes are basic blocks that are a set of statements that will be executed together and edges are those statements that can transfer the execution flow among blocks [18].

As far back as 1969, Balzer [8] proposed EXDAMS (Extendable Debugging And Monitoring Systems), in which programs could be debugged via backward and forward navigation. This system used the flow of control for visualization assisting with flowback analysis by showing how information flowed in the program to compute a specific value. However, EXDAMS did not implement the visualization to indicate the more suspicious code lines, which is likely to assist users with locating the fault.

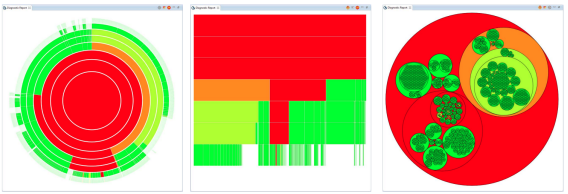
Zhang et al. [24] proposed Capture Propagation which computes the suspiciousness score for each block associating with edges via propagation of the test input data that led to the fault. Others have implemented an approach based on Markov Logic Network from machine learning by considering both dynamic behaviors of a software system and also a static viewpoint that depends on other statements via control flow or data flow [10][23].

2.2 Spectrum-Based Fault Localization

Spectrum-based fault localization refers to a diagnosis approach based on analyzing differences in program spectra, i.e. data collected during a test suite execution [1, 3, 18]. SBFL techniques pinpoint



(a) Example for Coloring in Editors: Jaguar [17]



(b) Example for Sunburst, Vertical Partition, and Bubble Hierarchy [11]

Fig. 1. Combined figure of Jaguar coloring and multiple visualization examples.

program elements that are more likely to contain faults by producing a ranking based on their suspicious score computed [10].

As a spectrum-based fault localization technique, Janssen et al. introduced Zoltar [12], which returns a ranking of likely faulty locations by using produced runtime data. Considering the fact that most automatic diagnosis techniques are predominantly statistical without explicitly considering multiple faults, variations of the tool – Zoltar-M(ultiple fault) and Zoltar-S(ingle fault) – were developed for handling multiple faults methods and single fault methods, respectively[5]. The authors have shown that for larger programs, Zoltar-M clearly outperforms all statistical approaches. However, compared with other techniques, Zoltar-M adds significantly more time/space complexity.

2.3 Visualization

One approach for visually assisting fault localization is the discrete coloring scheme [13][19]. This simply colors the program using three different colors: one showing if a statement is executed by only failing test cases; one if a statement is executed by only passing test cases; and one if the statement is executed by both. However, a potential problem with this scheme is that most lines will be executed by both passing and failing test cases and will all be colored the same, which might not help with fault localization.

Continuous coloring mapping refers to the use of colors and brightness to denote program entities' suspiciousness [19]. For instance, Ribeiro et al. [17] presented the Jaguar fault localization tool for Java programs. As shown in Fig. 1a, they provide the visualization of lists containing suspicious program entities by coloring lines of code in the editor. Furthermore, to improve the user interaction experience with more options and to offer more visualizations, Gouveia et al. [11] proposed Sunburst, Vertical Partition, and Bubble Hierarchy visualizations, shown in Fig 1b.

2.4 Tarantula approach

In *Visualization of Test Information to Assist Fault Localization*[13], Jones et al. provide a new visualization approach that assists fault localization, which they call color mapping. Color mapping uses the results of program testing and analyzes the passing and failing test cases that cover a particular statement. They map the results of the program testing by coloring lines of code as they would appear in a text editor, which provides a global view of these results while still letting the user access information about individual lines of code.

The authors improved the color mapping from the discrete approach to an approach using a much richer mapping, which they called the continuous approach. In the discrete approach of color mapping, there are only three cases of coloring: red if the statement is executed only by failing test

cases, green if the statement is executed only by passing test cases, and yellow if the statement is executed by both failing and passing test cases. However, in the continuous approach, they color the statement s by varying color and brightness, which is computed using the formula shown in Equation (1).

$$\text{color(s)} = \text{color (red)} + \frac{\%\text{passed(s)}}{\%\text{passed(s)} + \%\text{failed(s)}} * \text{color range} \tag{1}$$

However, there are limitations to this approach: coloring the source code as it appears in a text editor might not provide the user with a clear understanding of the distribution and relationship between the individual faulty line and the whole method, i.e. they may not get a sense of a suspicious path that is leading to the failing test. Additionally, as with many such approaches, faults that occur in conditional statements such as if-statements may not lead to a high suspiciousness ranking for that statement since it will be covered by both passing and failing tests, even if the branch the code takes is incorrect as a result of the fault.

2.5 Ochiai approach

The Ochiai approach, introduced in On the Accuracy of Spectrum-based Fault Localization [4], is a SBFL technique that were originally designed for molecular biology [2]. Similar to most SBFL, ochiai introduced a different coefficient for computing the suspiciousness for different code line. This is based on a similarity coefficient originally developed for biological taxonomy and adapted to software fault localization. The Ochiai coefficient is calculated by comparing the execution frequency of a statement in both failing and passing test cases, shown in Equation (2). The main difference between Ochiai and Tarantula is that Ochiais does not take into account program entities that are not executed in passing test cases [19].

$$\text{suspiciousness(s)} = \frac{\text{failed (s)}}{\sqrt{\text{total failed} * (\text{failed (s)} + \text{passed (s)})}} \tag{2}$$

3 APPROACH

To improve upon the Tarantula approach for assisting with fault localization, we propose using a Control Flow Graph (CFG) to visualize the possible locations of the faulty line, which helps the programmer have an overall understanding of the distribution of the suspicious lines with both a global view and local information for individual code lines. Our color mapping is based on the adjusted equation from the color component computation used in the Tarantula approach.

Additionally, we also introduce the consideration of edges in determining the suspiciousness of decision points in the CFG. For an if-statement or any other decision point in the CFG, we separate all test cases into two parts and record the passing and failing results independently, depending on which branches coming out of the decision point are covered. To be more specific, all the test cases that execute the conditional statement and pass will be counted separately from those test cases that execute the conditional statement and return false.

This section describes our approach in more detail and provides examples of the CFGs that represent the suspiciousness levels of different lines of code.

3.1 Control Flow Graph

Figures 2 and 3 show examples of CFGs rendered using our approach. Instead of using varying both color and brightness to achieve color mapping, we use different opacity of the same red color for coloring different blocks to represent their suspiciousness and also change the formats and colors of text, so that the user is more likely to notice the lines that are most suspicious.

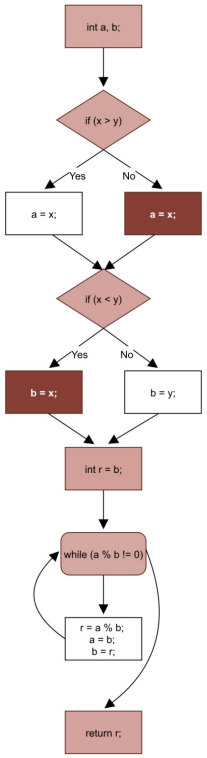


Fig. 2. Control Flow Graph for the method *gcd*
The fault is "a = x;", it should be "a = y;"

Before coloring the method, we need to do enough testing with various inputs that have the result of both passing and failing test cases. In order to compute the suspiciousness of each code line, we modify the formula that the Tarantula approach uses for computing color components. Then, we plug this computed suspicious score into the formula that calculates the opacity of red, in order to color the block of a statement s as shown in Equation 3.

$$\text{opacity}(s) = 1 - \text{suspiciousness}(s) = 1 - \frac{\%\text{passed}(s)}{\%\text{passed}(s) + \%\text{failed}(s)} \tag{3}$$

$$\begin{aligned} \%\text{passed}(s) &= \frac{\text{pass\_test}(s)}{\text{total\_pass}} \\ \%\text{failed}(s) &= \frac{\text{fail\_test}(s)}{\text{total\_fail}} \end{aligned} \tag{4}$$

Here, %passed(s), passing percentage of statement s, represents the percentage of passing test cases that execute statement s ($pass\_test(s)$) out of the total number of passing test cases ($total_pass$), shown in Equation 4; similarly, %failed(s), the failing percentage, indicates the number of failing test cases that execute statement s ($fail\_test(s)$) divided by the total number of failing test cases ($total\_fail$). Therefore, for the statement s, with the opacity calculated from the test cases, we can adjust the opacity of the block with the statement s as text inside.

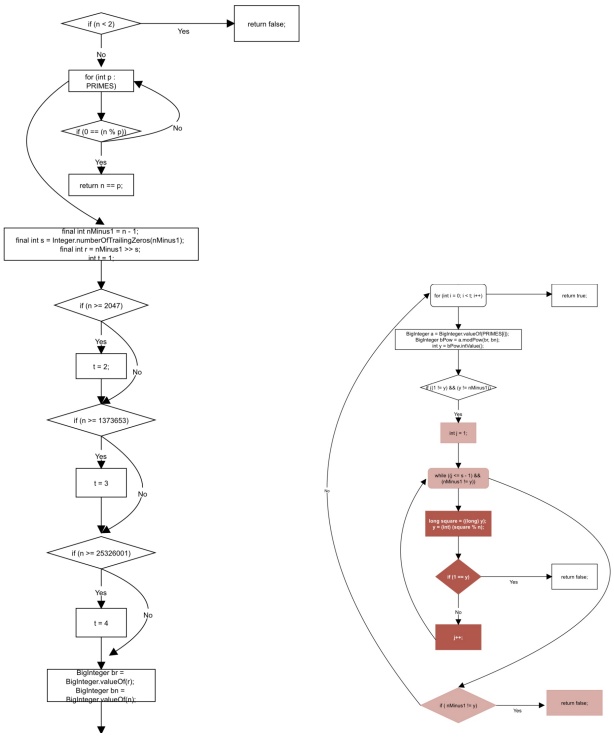


Fig. 3. Control Flow Graph for the method *prime*
"square = ((long) y);" should be "square = ((long) y)*y;"

In Fig. 2, the two boxes with the code $a = x$ and $b = x$ are colored with the highest opacity, to indicate that they are most suspicious; the boxes containing code lines that are less suspicious are colored in a way that does not stand out as much, using the same red color but with the opacity computed by Equation 3 with dynamic testing information.

Moreover, for determining the opacity of the code block, we use the precise relative value generated by Equation 3 instead of using the actual value and segmenting the values into 10 partitions imprecisely, as in the Tarantula approach. That is, within one piece of code, if the maximum opacity value that we have is not 100%, we compute all opacity values that we get from the equation by dividing by the maximum opacity value. This allows the code line that is ranked as most suspicious to be as obvious as possible because, for fault localization, the code line that is ranked as most suspicious, intuitively, is always the first line of code that the programmer will start examining, regardless of the value from the opacity function.

Last, in order to present a clearer visualization of the graph, we only color the nodes/statements that are ranked most suspicious (note that there may be more than one), second-most suspicious, or third-most suspicious, assuming their suspiciousness levels are above zero.

For example, Fig. 4 and 5 show different visualizations for the faulty program example presented in the original Tarantula paper [13]. Whereas the Tarantula approach (Fig. 5) colors some lines that are not covered by failing tests, specifically lines 4, 5, 9, and 10) and uses different colors for lines that have low suspiciousness levels (lines 1, 2, 3, and 13), our approach (Fig. 4) does not color lines

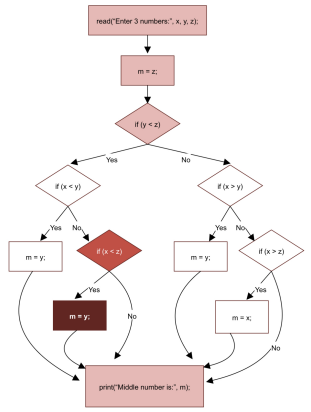


Fig. 4. Control Flow Graph for Faulty Program Example

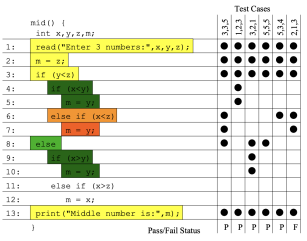


Fig. 5. Straight Line Format for Faulty Program Example [13]

that are not at all suspicious, uses opacity to demonstrate levels of suspiciousness, and more clearly shows the path that is most associated with the failing test.

3.2 Edge Consideration

Using a CFG to represent the code makes it possible to consider the suspiciousness of edges in the graph, specifically branches from decision points, and use those to modify the suspiciousness of conditional statements. The algorithm we described below could be applied both to Tarantula and ochiai, returning the modified version Tarntula+Edges and ochiai+Edges.

In particular, note that a faulty conditional statement would being executed no matter whether it evaluates to True or False. This will result in the inaccuracy of the function %passed(s) and %failed(s) because all the passing test cases will have executed the faulty conditional statement and so will all the failing test cases. Therefore, the result of the suspiciousness of the line would be inaccurate because it is determined by %passed(s) and %failed(s), and this conditional statement will not be highly ranked. Similarly, this will also influence the original ochiai suspiciousness function, shown in Equation 2.

In our modified approach, which we call *Tarantula+Edges* and *ochiai+Edges*, we consider the suspiciousness of branches in the CFG and assign their suspiciousness to the conditional statement from which they originate. That is, for conditional statements, instead of tallying the passing and failing test cases as we do for other code lines, we split this line into two parts: one for all test cases where the conditional statement evaluates to True, and another for those where it evaluates to False. Then, we use the collected information of the number of passing and failing test cases for

Table 1. Code Example #1. The fault is on line 2.

	input	n=100	n=110	n=45
	expected	r=400	r=550	r=90
	output	r=500	r=550	r=90
1	`int r = 0;`	Failed	Passed	Passed
2	`if (n>=100) {`	Failed	Passed	Passed
3	`r = 5n;`	Failed	Passed	
4	`} else if (100 >= n > 50) {`			Passed
5	`r = 4n;`			
6	`} else if (n <= 50){`			Passed
7	`r = 2n;`			Passed
	`}`			
8	`return r;`	Failed	Passed	Passed

Table 2. Code Example #1 with Edge Consideration.

	input	n=100	n=110	n=45
	expected	r=400	r=550	r=90
	output	r=500	r=550	r=90
1	`int r = 0;`	Failed	Passed	Passed
2	`if (n>=100) {`	Failed	Passed	Passed
2.1	**True**	Failed	Passed	
2.2	**False**			Passed
3	`r = 5n;`	Failed	Passed	
4	`} else if (100 >= n > 50) {`			Passed
5	`r = 4n;`			
6	`} else if (n <= 50){`			Passed
7	`r = 2n;`			Passed
	`}`			
8	`return r;`	Failed	Passed	Passed

Table 3. Suspiciousness Ranking for Each Code Line in Table 1

line 3	50%
line 1	33.3%
line 2	33.3%
line 8	33.3%
line 4	0%
line 5	0%
line 6	0%
line 7	0%

both edges to compute the suspiciousness of containing the faults separately and then choose the higher one – either the True branch's value or the value of the False branch – to be the overall value of the suspiciousness for this conditional statement.

For example, in the code shown in Table 1 , the fault is on line 2, and the correct version should be $if(n > 100)$ instead of $if(n >= 100)$. This table shows some input examples for the code with passing and failing test cases. We can use the equation to calculate the suspiciousness for each line.

As shown in Table 3, the line that contains the fault (line 2) is not top-ranked because the tests will execute the conditional statement, no matter whether it returns True or False, and line 2 will not seem particularly associated with failing test cases, even though it is the line with the fault.

However, after considering the two branches coming out of line 2, which we label Edge 2.1 and Edge 2.2 in Table 2, the suspiciousness of line 2 is set to the maximum of the suspiciousness values for each branch. This gives us $\max\{50\%, 33\%\} = 50\%$, which allows line 2 to be ranked as most suspicious.

4 EVALUATION

To evaluate the performance of our proposed two modified approaches, the Tarantula+Edges and the ochiai+Edges, we set up two different evaluations. For evaluating the Tarantula+Edges, we compare among Tarantula, Tarantula+Edges, and Set Union approaches based on the generated test of five methods from the Java Math library. For evaluating the ochiai+Edges approach, we compare that with the original ochiai approach based on 71 valid various real software faults from three different Java libraries from Defects4J.

4.1 Metrics

First, since the performance of the approach is directly proportional to how much it helps users. The better the approach can minimize the amount of work for bug localization, the better the approach is. In other words, the effectiveness of an approach is the amount of code lines that needed to be examined before reaching the faulty code line [22]. Since the number of code lines heavily depends on the length of the source code, we introduce metrics that we used to evaluate the performance among different approaches, which could eliminate the impact of the length of the source code.

(1) **Ranking of Faults (RF):** This metric reports the ranking information of the faults in the suspiciousness ranking. To be more specific, we have **Mean RF** refers to the average of RF for all faults in the same faulty version since a faulty version of the source code might have more than one faulty line. Also **Best RF** refers to the highest ranking of faults among all faults in the same faulty version. If our focus is to help users locate a good starting point

Table 4. Mutations used in experiment

code	mutants generated	all tests passed	all tests failed	usable mutants
sinh	148	62	0	**86**
log	448	274	0	**174**
prime	78	35	0	**43**
pow	224	89	0	**135**
gcd	20	3	12	**5**
cosh	129	63	3	**63**

for examining rather than providing a complete set of all faulty lines, successfully locating one faulty code line is enough. Therefore, this is also a helpful metric.

(2) **Wasted Effort (WE):** This metric counts the number of elements investigated before locating the fault divided by the number of elements in the source code. Similarly, we also have **Mean WE**, which refers to the average of WE for all faulty code lines in the same faulty version. **Best WE** refers to the number of elements investigated before reaching any of the faulty code lines over the total number of code lines.

4.2 Tarantula+Edges Evaluation

4.2.1 Methodology. In order to evaluate the effectiveness of Tarantula+Edges, we compared the different approaches in five methods from the Apache Commons Mathematics Library [7]: hyperbolic sine (*sinh*) and cosine (*cosh*); power (*pow*); logarithm (*log*); and *prime*. We also used an iterative implementation of Euclid's greatest common divisor (*gcd*) algorithm. All software was implemented in Java.

For each of the six methods, we used mutation analysis [20] to systematically insert faults into the code, each version of the method containing exactly one fault. For simplicity, we only used "first order mutants" involving arithmetic, relational, and logical operators.

We then generated test cases for each of the methods. For *pow*, we used 200 random inputs and an additional five in order to achieve 100% statement coverage. For *gcd*, we used 100 random inputs, and for *prime* we used 10,000 random odd inputs. For *log* we used 10 inputs that achieved 100% statement coverage, and for *cosh* and *sinh* we used the values 0-50. The original, unmutated versions of these methods were used to identify the expected output for each test case.

Because the fault localization techniques described here require both passing and failing test cases, we discarded any mutation for which all of our tests passed or for which all of the tests failed, resulting in the usable mutants indicated in Table 4.

For each usable mutation, we then used the test cases and the fault localization techniques to determine whether the technique could correctly identify the line on which the fault occurred. Specifically, each technique ranks the lines according to their suspiciousness, and our evaluation assessed the ranking of the faulty line, and the average number of lines of code required to find the faulty line, as described in the subsections below.

4.2.2 Results.

(1) **Ranking of Faults (RF)**

For our first experiment, we used both Tarantula and Tarantula+Edges to determine the suspiciousness of the lines of code in the mutant versions for the six methods described above, and then compared the ranking of the faulty line to see if it increased as a result of using Tarantula+Edges.

We have compared different percentages of the faulty versions that are ranked as #1 through #4 and below for each approach - either with edge consideration or without. Fig. 6 and Table 5 illustrate the improvements and differences introduced by considering edges

Method	ranked	Tarantula	Tarantula+Edges
sinh	#1	50	54
	#2	5	4
	#3	28	25
	#4 and below	3	3
log	#1	169	169
	#2	5	5
	#3	0	0
	#4 and below	0	0
prime	#1	3	40
	#2	22	3
	#3	14	0
	#4 and below	4	0
pow	#1	120	123
	#2	1	2
	#3	1	0
	#4 and below	13	10
gcd	#1	5	5
	#2	0	0
	#3	0	0
	#4 and below	0	0
cosh	#1	55	55
	#2	3	3
	#3	5	5
	#4 and below	0	0

Table 5. Ranking of faulty lines with the Tarantula approach and the Tarantula+Edges approach

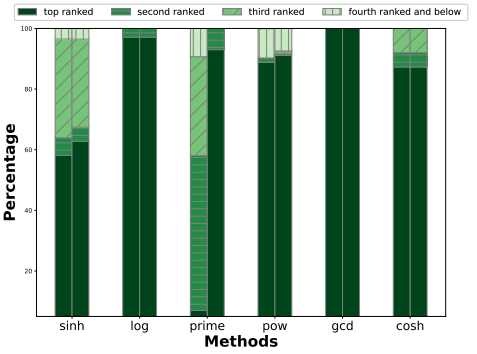


Fig. 6. Ranking of Faults. For each program, rankings provided by Tarantula are shown on the left, and Tarantula+Edges is shown on the right.

for each method compared with the original Tarantula approach, showing the cumulative rankings of the lines that contain the fault as top-ranked, ranked #2, etc. both with and without considering edges.

We observed that for *sinh*, the number of faulty versions for which the faulty line is top-ranked increased from 50 to 54 by applying the Tarantula+Edges approach. Meanwhile, the number of faulty lines ranked #2 and #3 both decreased from 5 to 4 and 28 to 25 respectively. The number of faulty lines that are ranked as #4 remains the same, 3 versions total.

We noticed a similar improvement for both *prime* and *pow* for the Tarantula+Edges approach compared with the Tarantula approach. For *prime*, the number of top-ranked faulty lines increased from 3 to 40; the number of faulty lines ranked as #2 changed from 22 to 3, with all 19 converting to be top-ranked; both of the numbers of being ranked as #3 and #4 and below changed from 14 and 4 to 0. Similarly, for *pow*, with the help of the consideration of edges, the number of faulty lines that were top-ranked increased from 120 to 123, and the #2-ranked lines increased from 1 to 2.

However, we also noticed that the performance of the two different approaches remains the same for the other three methods.

(2) **Mean Wasted Effort (Mean WE)**

In this part of our experiment, we compare the Waster Effort for three different approaches: Tarantula, Tarantula+Edges, and the Set-union approach.

For Tarantula, we measured the number of lines of code that would need to be read as those that have a suspiciousness value greater than or equal to that of the faulty line. For Tarantula-Edges, we used the same metric, but also include any statements (decision points) for which one of its edges (branches) has a suspiciousness value greater than or equal to that of the faulty line.

The computation for the Set-union approach is slightly more complex. The Set-union approach determines a set of potentially faulty lines by removing the union of all statements executed by all passing test cases from the set of statements executed by a single failing test case[14]. The number of lines to be considered before finding the line that contains the fault is determined by the following:

1	`int a = 0;`
2	`int b = 1;`
3	`if (!(a > b)) {`
4	`  return b;`
	`} else {`
5	`  return b + 1;`
	`}`

Table 6. Code Example #2

Methods	Tarantula+Edges	Tarantula	Set Union
sinh	43.6620%	40.8451%	70.8483%
log	59.6330%	58.7156%	40.8310%
prime	71.8750%	75.0000%	80.0145%
pow	46.6667%	43.3333%	77.4198%
gcd	92.3077%	92.3077%	50.7692%
cosh	68.5714%	68.5714%	76.4626%

Table 7. Results of the Experiment: Average Percentage of Code Needed to be Read in Order to Find Fault

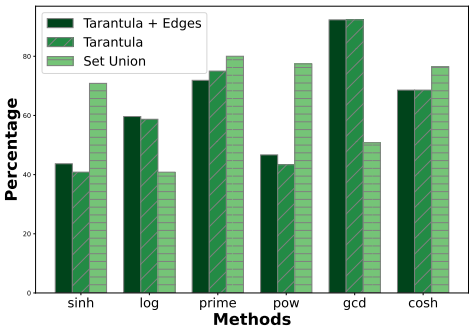


Fig. 7. Results of experiment: Average percentage of code needed to be read in order to find fault

(a) If this technique computes an empty initial set of statements, i.e. every line has been executed by a failing test case, in theory, we need to evaluate all the lines in the method to locate the fault.
(b) If this technique computes a set that includes the faulty line, in theory, the number of lines we need to test is the size of the set.
(c) However, if the technique computes a non-empty set that does *not* include the faulty line, we would need to consider the lines that are contained in the set, and then consider the lines that are one unit distance away (in the CFG) from the set of lines that we evaluated, and so on, until we find the faulty line.

For example, for code showing in Table 6, if the set returned from the Set-union technique is {Line 4}, but line 4 does not contain the fault, we would need to consider the lines that are one unit distance away from it in the CFG, specifically line 3, since after the return statement, there are no more lines after line 4, so we only consider the previous possible operation, which is line 3. If line 3 is not the faulty line, we need to consider the lines that are one unit distance away from the set {Line 3, Line 4}, which includes lines 2 and 5, and so on.

As shown in Table 7 and Fig. 7, for *prime*, the Tarantula+Edges approach is more effective on average than the Tarantula approach, for which 71.8750% and 75.0000% of the code needs to be read before locating the fault, respectively.

For both *gcd* and *cosh*, the Tarantula approach and the Tarantula+Edges approach perform the same, with 92.3077% for *gcd* and 68.5714% for *cosh*. Tarantula slightly outperforms Tarantula+Edges for *sinh*, *log*, and *pow*; analysis of this is provided below. Although both approaches are typically better than Set-union, the Set-union approach does perform better for *gcd* and *log*.

4.3 Ochiai+Edges Evaluation

4.3.1 Methodology. To evaluate the effectiveness of Ochiai+Edges approach, we compare it with the original ochiai approach in 83 buggy versions (71 valid buggy version) total in three different libraries from Defects4J: Commons Math, JFreeChart, and Joda-Time.

Defects4J is a medium size open-source database of real-world faults designed to support controlled testing studies in software research proposed by Just et al [15]. It includes 357 genuine

library	#valid	better Mean RF	better Best RF	mean ranking	+Edges mean ranking
Commons Math	25	0	0	175.5866	175.7586
JFreeChart	22	3	1	331.0530	331.1212
Joda-Time	24	3	2	324.7488	325.0775

Table 8. Overall Performance of the Ranking of Faults

bugs from five popular Java projects, such as JFreeChart and Apache Commons Math. Unlike many databases that rely on synthetic or hand-seeded faults, Defects4J provides actual faults with corresponding fixed versions and test cases that can reveal each fault. This approach offers a realistic basis for assessing testing tools and techniques since these bugs are derived directly from actual development histories and practices.

We first generate the buggy line number provided by Defects4J for each buggy version. Note that the buggy line number might not be valid if the buggy version only include deletions and deletions do not introduce a different path. Among 83 buggy versions, we have 25 valid buggy versions for Commons Math library, 22 valid buggy versions for JFreeChart library, and 24 valid buggy versions for Joda-Time library.

Given the spectra information of code lines for test cases generated by Defects4J, we can calculate the suspicious score for every element. We then generate a ranking output for each of 83 buggy versions via sorting code line by their suspicious score.

For each valid buggy version, we compare its buggy line number with the corresponded ranking information. This process will return the total number of lines of code required to be investigated to find the faulty lines and also the ranking of the faulty lines.

4.3.2 Results.

(1) **Ranking of Faults (RF)** For our evaluation experiment, we used both the ochiai approach and the ochiai+Edges approach to find the suspiciousness of the lines of code in all valid buggy versions described above. Similarly, we compared the ranking of faults computed by ochiai and ochiai+Edges approach to determine if the modified approach is better. The smaller the ranking is, the better the approach is.

We have compared two different RF: the Mean RF and the Best RF. Among all valid buggy versions for three libraries, we notice that there are 3 faulty versions from JFreeChart, and 3 buggy versions from Joda-Time perform with a better Mean RF. Also, in JFreeChart, there is 1 faulty version perform a better Best RF. There are 2 faulty versions perform a better Best RF for Joda-Time library.

Another metric to notice is the *mean ranking* and *+Edges mean ranking*, which are the average of the mean RF values for all faulty versions within the same library. We can see that the mean ranking value for Commons Math is 175.5866 without the edge consideration, and 175.7586 with edge consideration. We observed similar trends for JFreeChart, with 331.0530 without edges consideration and 331.1212 with ochiai+Edges approach, and also for Joda-Time with 324.7488 and 325.0775 respectively.

(2) **Mean Wasted Effort (Mean WE)**

For our evaluation experiment, we not only compute the ranking of faults, but also have the Wasted Effort computed, which is the percentage of the elements needed to be investigated before locating the fault. We compared the WE computed by ochiai and ochiai+Edges approach to determine if the modified approach is better. The smaller the WE is, the better the approach is.

library	#valid	better Mean WE	better Best WE	mean percentage	+Edges mean percentage
Commons Math	25	5	5	10.9413%	10.9629%
JFreeChart	22	3	3	14.3638%	14.3605%
Joda-Time	24	4	6	27.1190%	27.1282%

Table 9. Overall Performance of the Wasted Efforts

Similar to RF, we have compared two different WE: the Mean WE and the Best WE. Among all valid buggy versions for three libraries, we notice that there are 5 faulty versions from Commons Math, 3 faulty versions from JFreeChart, and 4 buggy versions from Joda-Time perform with a better Mean WE. Also, in JFreeChart, there are 5 faulty versions performed a better Best WE. There are 3 faulty versions performed a better Best ME in JFreeChart. And there are 6 faulty versions performed a better Best ME for Joda-Time library. We have 16.90% of the valid faulty versions perform with a better Mean WE, and 19.72% with a better Best WE.

Another metric to notice is the *mean percentage* and *+Edges mean percentage*, which are the average of the mean WE values for all faulty versions within the same library. We can see that the mean percentage value for Commons Math is 10.9413% without the edge consideration, and 10.9629% with edge consideration. We observed a similar trend for Joda-Time, with 27.1190% without edges consideration and 27.1282% with ochiai+Edges approach. For JFreeChart, the mean percentage value decreases a tiny bit from 14.3638% to 14.3605%.

and also for Joda-Time with 324.7488 and 325.0775 respectively.

4.4 Discussion

Based on our experiments, we noticed that the Tarantula+Edges improves the ranking performance for three of the six methods we tested, and the rest remained the same. In order to examine the effect of our Tarantula+Edges on real-life debugging, we also evaluated the number of lines having a higher ranking than the faulty line. We would expect that our Tarantula+Edges approach would be more effective than the original Tarantula approach since the first experiment shows that the inclusion of edges improved the ranking of faulty lines that were conditional statements. However, we found that the Tarantula+Edges approach performs better than the others for only one of the six methods.

Similarly, we also noticed that even though the better Mean RF, Best RF, Mean WE, and Best WE allow the users to locate the fault and start debugging successfully, the mean rankings and also mean percentages for some libraries still introduce a tiny increase.

For Tarantula+Edges and ochiai+Edges, we need to look at more code to locate the faulty line because of the consideration of edges, as not only the faulty line (assuming it is inside the conditional statement), but also the other code lines inside other or the same if-block may return a larger number from the formula we use to compute the suspiciousness.

For most situations (among methods we have tested) the different conditions used in if-statements have a high possibility of having a correlation relationship. As an example, assume that a piece of code checks a condition A in an if-statement, and then later (outside of the corresponding if-block) checks a condition B which is likely to be true if A is true. This means that a path that covers the "True" branch coming out of A is also likely to cover the "True" branch coming out of B. Thus, if the suspiciousness of A increases as a result of taking the "True" branch coming out of it, then the suspiciousness of B will increase as well, as a result of that edge also being taken.

Table 10. Code Example #3. The fault is on line 5.

	input	a = 4	a = 0	a = 5
1	`if (a > 0) {`	Failed	Passed	Failed
1.1	`True`	Failed		Failed
2	`a = a + 1;`	Failed		Failed
	`} else {`			
1.2	`False`		Passed	
3	`a = a - 1;`		Passed	
	`}`			
4	`if (a > 3) {`	Failed	Passed	Failed
4.1	`True`	Failed		Failed
5	**`a = 3;`**	Failed		Failed
	`} else {`			
4.2	`False`		Passed	
6	`a = a - 1;`		Passed	
	`}`			

Table 11. Suspiciousness Rankings for Elements in Code Example #3

Edge 1.1	100%
Line 2	100%
Edge 4.1	100%
Line 5	100%
Line 1	66.67%
Line 4	66.67%
Edge 1.2	0%
Line 3	0%
Edge 4.2	0%
Line 6	0%

Because most of the test methods we use are math methods, most if-else conditions in the code have correlation relationships. Table 10 shows a simple example demonstrating how the consideration of edges will increase the number of lines of code that need to be read. Assume the fault is on line 5, which should be $a = 4$. If $a > 0$ is true on line 1, there is a high possibility that $a > 3$ compared with if $a \not> 0$, and vice versa. Therefore, for the situation where the test input is greater than 3, if the test executes edge 4.1 and line 5, it will also have executed edge 1.1 and line 2. Therefore, the ranking produced after combining all three output sets is shown in Table 11. We can notice that by the Tarantula approach, we only need to read lines 2 and 5; however, by the Tarantula+Edges approach, we would also need to review lines 1 and 4, since branches coming out of those decision points have a high suspiciousness ranking.

Therefore, the result from the Tarantula+Edges approach is likely to contain extra code lines, which are ranked before the faulty line. This is why for those methods, the Tarantula+Edges approach is not better than the original Tarantula or the Set-union approach.

5 LIMITATIONS

The Tarantula+Edges approach and the ochiai+Edges indeed performs well for most of the methods we tested, but the way of computing the suspiciousness of containing the fault for the different code lines may cause some limitations. This approach cannot be used for CFGs that do not have any branches, i.e. it is necessary not to have the identical same path executed when we test the code with different inputs. Similar to most SBFL techniques, Tarantula+Edges will return a group of statements with the same suspiciousness score, i.e. the same ranking if they exhibit the same execution pattern [22]. Not containing edges or having too many parallel lines of code under a conditional statement might cause some difficulties for the user to find the faults effectively since there might be many lines of code ranked higher than or equal to the faulty line.

Furthermore, our experiment only evaluated six Java methods. This limited scope may have constrained the generalizability of our findings, as the selected methods might not fully represent the diversity and complexity of other methods. Future work could consider the types of methods for which the Tarantula+Edges is more effective than Tarantula and other approaches.

6 CONCLUSION

In this paper, we proposed using a Control Flow Graph as a visualization approach for fault localization, which includes highlighting the statements most likely to contain a fault based on the suspiciousness we compute. We also introduced the Tarantula+Edges approach and evaluated its effectiveness by comparing the likelihood of the faulty line being ranked as most suspicious and the percentage of the method we need to investigate in order to locate the fault.

In addition to addressing the limitations described above and completing the implementation of the Tarantula+Edges tool, future work could further investigate the use of CFG edges and paths in localizing faults, as well as the most effective ways of visualizing the annotated CFG in order to help make fault localization and debugging more effective and efficient. We would also consider combining the *+Edges* modification we have so far with data-flow spectrum fault localization.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J.C. van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792. https://doi.org/10.1016/j.jss.2009.06.035 SI: TAIC PART 2007 and MUTATION 2007.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J.c. Van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. 39–46. https://doi.org/10.1109/PRDC.2006.18

[3] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. 89–98. https://doi.org/10.1109/TAIC.PART.2007.13

[4] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. 89–98. https://doi.org/10.1109/TAIC.PART.2007.13

[5] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2011. Simultaneous debugging of software faults. *Journal of Systems and Software* 84, 4 (2011), 573–586. https://doi.org/10.1016/j.jss.2010.11.915 The Ninth International Conference on Quality Software.

[6] H. Agrawal, J.R. Horgan, S. London, and W.E. Wong. 1995. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95*. 143–151. https://doi.org/10.1109/ISSRE.1995.497652

[7] Apache Software Foundation. [n. d.]. Apache Commons Mathematics Library. https://commons.apache.org/proper/commons-math/

[8] R. M. Balzer. 1969. EXDAMS: extendable debugging and monitoring system. In *Proceedings of the May 14-16, 1969, Spring Joint Computer Conference* (Boston, Massachusetts) *(AFIPS '69 (Spring))*. Association for Computing Machinery, New York, NY, USA, 567–580. https://doi.org/10.1145/1476793.1476881

[9] James S. Collofello and Scott N. Woodfield. 1989. Evaluating the effectiveness of reliability-assurance techniques. *Journal of Systems and Software* 9, 3 (1989), 191–195. https://doi.org/10.1016/0164-1212(89)90039-3

[10] Higor A de Souza, Marcos L Chaim, and Fabio Kon. 2016. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347* (2016).

[11] Carlos Gouveia, José Campos, and Rui Abreu. 2013. Using HTML5 visualizations in software fault localization. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. 1–10. https://doi.org/10.1109/VISSOFT.2013.6650539

[12] Tom Janssen, Rui Abreu, and Arjan J.C. van Gemund. 2009. Zoltar: a spectrum-based fault localization tool. In *Proceedings of the 2009 ESEC/FSE Workshop on Software Integration and Evolution @ Runtime* (Amsterdam, The Netherlands) *(SINTER '09)*. Association for Computing Machinery, New York, NY, USA, 23–30. https://doi.org/10.1145/1596495.1596502

[13] J.A. Jones, M.J. Harrold, and J. Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. 467–477. https://doi.org/10.1145/581396.581397

[14] James A. Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (2005). https://api.semanticscholar.org/CorpusID:14937735

[15] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[16] Sean Kauffman, Carlos Moreno, and Sebastian Fischmeister. 2024. Annotating Control-Flow Graphs for Formalized Test Coverage Criteria. arXiv:2407.04144 [cs.SE] https://arxiv.org/abs/2407.04144

[17] Henrique L. Ribeiro, Roberto P. A. de Araujo, Marcos L. Chaim, Higor A. de Souza, and Fabio Kon. 2018. Jaguar: A Spectrum-Based Fault Localization Tool for Real-World Software. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 404–409. https://doi.org/10.1109/ICST.2018.00048

[18] Henrique L. Ribeiro, P. A. Roberto de Araujo, Marcos L. Chaim, Higor A. de Souza, and Fabio Kon. 2019. Evaluating data-flow coverage in spectrum-based fault localization. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–11. https://doi.org/10.1109/ESEM.2019.8870182

[19] Qusay Idrees Sarhan and Árpád Beszédes. 2022. A Survey of Challenges in Spectrum-Based Software Fault Localization. *IEEE Access* 10 (2022), 10618–10639. https://doi.org/10.1109/ACCESS.2022.3144079

[20] P. Thevenod-Fosse, H. Waeselynck, and Y. Crouzet. 1991. An experimental study on software structural testing: deterministic versus random input generation. In *Digest of Papers. Fault-Tolerant Computing: The Twenty-First International Symposium*. IEEE Computer Society, Los Alamitos, CA, USA, 410,411,412,413,414,415,416,417. https://doi.org/10.1109/FTCS.1991.146694

[21] Shailesh Tiwari, K.K. Mishra, Anoj Kumar, and A.K. Misra. 2011. Spectrum-Based Fault Localization in Regression Testing. In *2011 Eighth International Conference on Information Technology: New Generations*. 191–195. https://doi.org/10.1109/ITNG.2011.40

[22] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740. https://doi.org/10.1109/TSE.2016.2521368

[23] Sai Zhang and Congle Zhang. 2014. Software bug localization with markov logic. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 424–427. https://doi.org/10.1145/2591062.2591099

[24] Zhenyu Zhang, W. K. Chan, T. H. Tse, Bo Jiang, and Xinming Wang. 2009. Capturing propagation of infected program states. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) *(ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/1595696.1595705