Use of Control Flow Graphs with Edges Consideration for Fault Localization

Anonymous

Abstract—**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 a Spectrum-Based Fault Localization (SBFL) technique with visualization 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 enhanced eight established fault localization techniques - including Tarantula, Ochiai, Barinel, DStar, Jaccard, O^p, Russell, and Sørensen - through our novel +Edges modification approach. Our experimental evaluation demonstrates that these enhanced techniques yield measurable improvements in fault localization accuracy. The results, validated across 262 real faults from the Defects4J benchmark, show that our approach helps developers identify defects more effectively, with certain techniques like Sørensen achieving performance gains of up to 55.8%.

I. 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 [Collofello and Woodfield(1989)]. 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 [de Souza et al.(2016)].

To assist with fault localization, Jones et al. proposed a fault localization technique called Tarantula that displays lines of code in different colors based on the computed suspiciousness of the line [Jones et al.(2002)], [Ribeiro et al.(2019)], [Abreu et al.(2009b)], [Abreu et al.(2007a)]. In 2006, Abreu et al. proposed Ochiai which was originally used in molecular biology, which was shown to outperform Tarantula [Abreu et al.(2006)]. Despite the effectiveness of both approaches [Jones and Harrold(2005)], [Abreu et al.(2006)], we propose that it may be preferable to display the code as a Control Flow Graph (CFG) so that, for the programmer, it can be easier to 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 [Kauffman et al.(2024)]. 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.

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, which aims to improve the accuracy of locating the fault, especially for a piece of code that contains more decision points.
3) An empirical evaluation of +Edges using 262 valid real-world bugs from Defects4J, which shows a better performance compared with the existing technique based on different ranking metrics.

II. 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 [de Souza et al.(2016)] and ease of use [Sarhan and Beszédes(2022)][Tiwari et al.(2011)].

TABLE I: Fault Localization Techniques and Their Formulas

Technique	Formula	Technique	Formula
Tarantula	$\frac{\frac{n_f(s)}{n_f}}{\frac{n_f(s)}{n_f}+\frac{n_p(s)}{n_p}}$	Ochiai	$\frac{n_f(s)}{\sqrt{n_f \times (n_f(s)+n_p(s))}}$
Barinel	1- $\frac{n_p(s)}{n_p(s)+n_f(s)}$	DStar	$\frac{n_f(s)^2}{n_p(s)+(n_f-n_f(s))}$
Jaccard	$\frac{n_f(s)}{n_f(s)+n_p(s)+n_f-n_f(s)}$	O^p	$\frac{n_f(s)}{n_f(s)+2\times n_p(s)+n_f-n_f(s)}$
Russell	$\frac{n_f(s)}{n_f+n_p}$	Sørensen	$\frac{2\times n_f(s)}{2\times n_f(s)+n_p(s)+n_f-n_f(s)}$

A. *Control-flow spectra*

SBFL techniques use control-flow spectra to pinpoint the program entities that are more likely to cause the unexpected output [Ribeiro et al.(2019)]. 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 [Ribeiro et al.(2019)].

As far back as 1969, Balzer [Balzer(1969)] 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. [Zhang et al.(2009)] 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 [de Souza et al.(2016)][Zhang and Zhang(2014)].

B. *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 [Abreu et al.(2009b)], [Abreu et al.(2007a)], [Ribeiro et al.(2019)]. SBFL techniques pinpoint program elements that are more likely to contain faults by producing a ranking based on their suspicious score computed [de Souza et al.(2016)].

A key aspect of SBFL is the variety of suspiciousness formulas proposed in the literature. Table I summarizes state-of-the-art SBFL techniques' formulas. Note that the formula for a code statement s is represented by $n_f(s)$, number of failing tests executing statement s, $n_p(s)$, number of passing tests executing statement s, n_f,total failing tests in test suite, and n_p, total passing tests in test suite. The following techniques are included:

1) *Tarantula:* Contrasts the ratios of failing and passing test coverage to identify suspicious statements [Jones et al.(2002)].

2) *Ochiai:* Uses cosine similarity to measure the correlation between failing tests and statement execution[Abreu et al.(2007b)], [Abreu et al.(2006)].

3) *Barinel:* Estimates fault probability by analyzing the ratio of passing and failing test coverage, which accounts for the fact that faulty components may fail intermittently[Abreu et al.(2009a)].

4) *DStar:* Emphasizes strong correlations between statements and failing tests, penalizing irrelevant executions [Wong et al.(2014)].

5) *Jaccard:* Computes suspiciousness based on the intersection of executed statements and failing tests[Abreu et al.(2007b)].

6) *O^p:* Adjusts Jaccard by giving higher weight to passing tests, reducing false positives[Naish et al.(2011)].

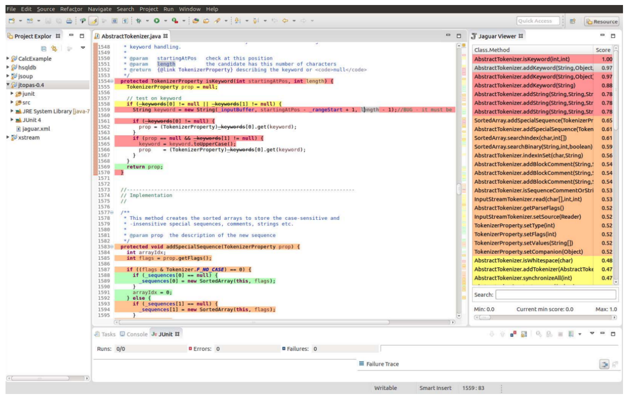
7) *Russell:* Measures fault likelihood as the proportion of failing tests covering a statement[Russell and Rao(1940)].

8) *Sørensen:* : Balances failing test coverage against all executions, similar to Jaccard but less strict[Sørensen(1948)].

C. *Tarantula technique*

In *Visualization of Test Information to Assist Fault Localization*[Jones et al.(2002)], Jones et al. provide a new visualization technique 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 technique to a technique using a much richer mapping, which they called the continuous technique. In the discrete technique of color mapping, there are only three cases of coloring: red if the statement is executed only by failing test cases, green if the



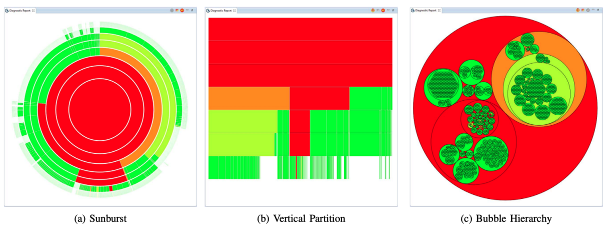
```
   * keyword handling.
   *
   * @param   startingAtPos   check at this position
   * @param   length          the candidate has this number of characters
   * @return  {@link TokenizerProperty} describing the keyword or <code>null</code>
   */
  protected TokenizerProperty isKeyword(int startingAtPos, int length) {
    TokenizerProperty prop = null;

    // test on keyword
    if (_keywords[0] != null || _keywords[1] != null) {
      String keyword = new String(_inputBuffer, startingAtPos - _rangeStart + 1, length - 1);//BUG - it must be

      if (_keywords[0] != null) {
        prop = (TokenizerProperty)_keywords[0].get(keyword);
      }
      if (prop == null && _keywords[1] != null) {
        keyword = keyword.toUpperCase();
        prop    = (TokenizerProperty)_keywords[0].get(keyword);
      }
    }
    return prop;
  }


  //---------------------------------------------------------------
  // Implementation
  //

  /**
   * This method creates the sorted arrays to store the case-sensitive and
   * -insensitive special sequences, comments, strings etc.
   *
   * @param prop  the description of the new sequence
   */
  protected void addSpecialSequence(TokenizerProperty prop) {
    int arrayIdx;
    int flags = prop.getFlags();

    if ((flags & Tokenizer.F_NO_CASE) == 0) {
      if (_sequences[0] == null) {
        _sequences[0] = new SortedArray(this, flags);
      }
      arrayIdx = 0;
    } else {
      if (_sequences[1] == null) {
        _sequences[1] = new SortedArray(this, flags);
      }
```

Class.Method	Score
AbstractTokenizer.isKeyword(int,int)	1.00
AbstractTokenizer.addKeyword(String,Object,	0.97
AbstractTokenizer.addKeyword(String,Object)	0.97
AbstractTokenizer.addKeyword(String)	0.88
AbstractTokenizer.addString(String,String,Str	0.78
AbstractTokenizer.addString(String,String,Str	0.78
AbstractTokenizer.addString(String,String,Str	0.78
SortedArray.addSpecialSequence(TokenizerPr	0.65
AbstractTokenizer.addSpecialSequence(Token	0.61
SortedArray.searchIndex(char,int[])	0.61
SortedArray.searchBinary(String,int,boolean)	0.59
AbstractTokenizer.indexInSet(char,String)	0.56
AbstractTokenizer.addBlockComment(String,S	0.54
AbstractTokenizer.addBlockComment(String,S	0.54
AbstractTokenizer.addBlockComment(String,S	0.54
AbstractTokenizer.isSequenceCommentOrStri	0.53
InputStreamTokenizer.read(char[],int,int)	0.53
AbstractTokenizer.getParseFlags()	0.52
InputStreamTokenizer.setSource(Reader)	0.52
TokenizerProperty.setType(int)	0.52
TokenizerProperty.setFlags(int)	0.52
TokenizerProperty.setValues(String[])	0.52
TokenizerProperty.setCompanion(Object)	0.52
AbstractTokenizer.isWhitespace(char)	0.48
AbstractTokenizer.addTokenizer(AbstractToke	0.47
AbstractTokenizer.synchronizeAll(int)	0.47

Search:
Min: 0.0   Current min score: 0.0   Max: 1.0

Runs: 0/0   Errors: 0   Failures: 0

(a) Example for Coloring in Editors: Jaguar [Ribeiro et al.(2018)]



(b) Example for Sunburst, Vertical Partition, and Bubble Hierarchy [Gouveia et al.(2013)]

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

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 technique, they color the statement s by varying color and brightness, which is computed using the formula shown in Equation (1).

$$\text{suspiciousness\_T(s)} = \frac{\%\text{passed(s)}}{\%\text{passed(s)} + \%\text{failed(s)}} \tag{1a}$$

$$\text{color(s)} = \text{color (red)} + \text{suspiciousness\_T(s)} * \text{color range} \tag{1b}$$

However, there are limitations to this technique: 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 techniques, 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.

D. *Visualization*

One approach for visually assisting fault localization is the discrete coloring scheme [Jones et al.(2002)], [Sarhan and Beszédes(2022)]. 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 [Sarhan and Beszédes(2022)]. For instance, Ribeiro et al. [Ribeiro et al.(2018)] 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. [Gouveia et al.(2013)] proposed Sunburst, Vertical Partition, and Bubble Hierarchy visualizations, shown in Fig 1b.

III. Approach

To improve on existing SBFL techniques including Tarantula, Ochiai, etc., we propose using a Control Flow Graph (CFG) to visualize the likelihood of each code line containing the fault, which is computed by the suspicious score calculated based on different techniques (shown in (1a) and (**??**)). This 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 examples are based on the adjusted equation from the color component computation used in the Tarantula technique, which could be modified to other techniques by simply changing the computations of the suspiciousness score.

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 fail.

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

A. *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.

Before coloring the code, 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 technique uses

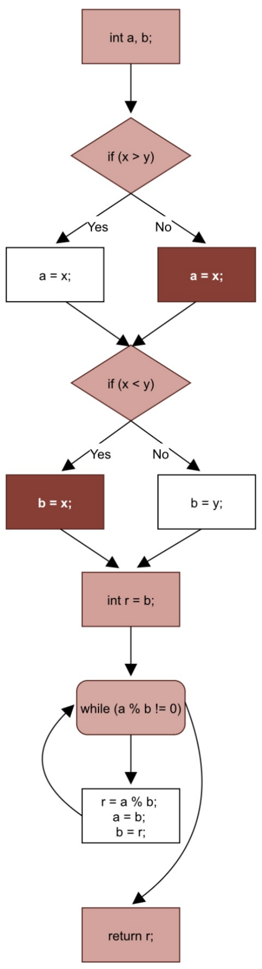


Fig. 2: Control Flow Graph for the method *gcd*
The fault is “a = x;”, it should be “a = y;”

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 2.

$$\begin{aligned} \text{opacity}(s) &= 1 - \text{suspiciousness\_T}(s) \\ &= 1 - \frac{\%\text{passed}(s)}{\%\text{passed}(s) + \%\text{failed}(s)} \end{aligned} \tag{2}$$

$$\begin{aligned} \%\text{passed}(s) &= \frac{\text{pass\_test}(s)}{\text{total\_pass}} \\ \%\text{failed}(s) &= \frac{\text{fail\_test}(s)}{\text{total\_fail}} \end{aligned} \tag{3}$$

Here, $\%\text{passed}(s)$, 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 3; similarly, $\%\text{failed}(s)$, 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 ($total\_fail$).

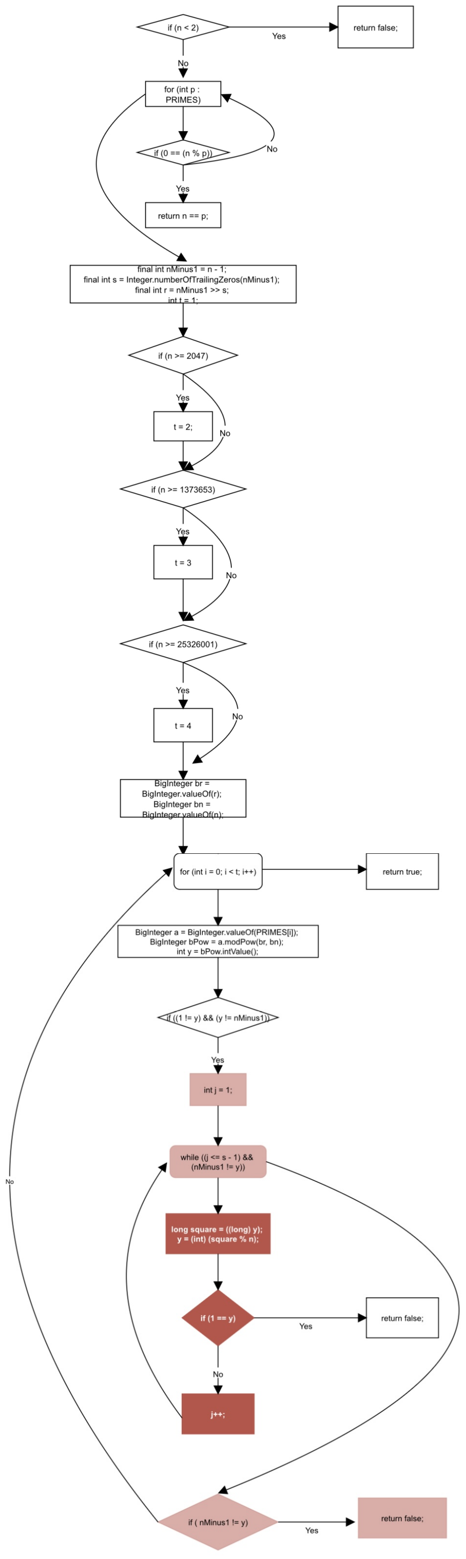

if (n < 2)
Yes
return false;
No
for (int p : PRIMES)
if (0 == (n % p))
No
Yes
return n == p;
final int nMinus1 = n - 1;
final int s = Integer.numberOfTrailingZeros(nMinus1);
final int r = nMinus1 >> s;
int t = 1;
if (n >= 2047)
Yes
No
t = 2;
if (n >= 1373653)
Yes
No
t = 3
if (n >= 25326001)
Yes
No
t = 4
BigInteger br = BigInteger.valueOf(r);
BigInteger bn = BigInteger.valueOf(n);
for (int i = 0; i < t; i++)
return true;
BigInteger a = BigInteger.valueOf(PRIMES[i]);
BigInteger bPow = a.modPow(br, bn);
int y = bPow.intValue();
if ((1 != y) && (y != nMinus1))
Yes
int j = 1;
while ((j <= s - 1) && (nMinus1 != y))
long square = ((long) y);
y = (int) (square % n);
if (1 == y)
Yes
return false;
No
j++;
No
if (nMinus1 != y)
Yes
return false;

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

Therefore, for the statement s, with the opacity calculated from the test cases, we can adjust the opacity of the block with the

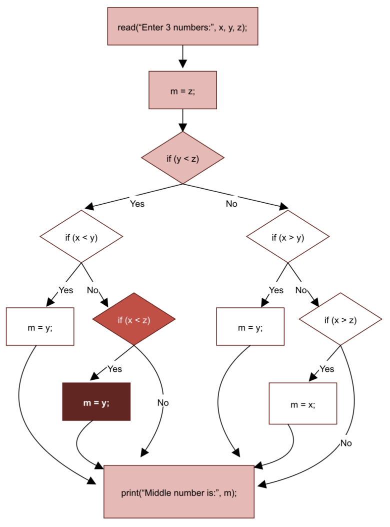
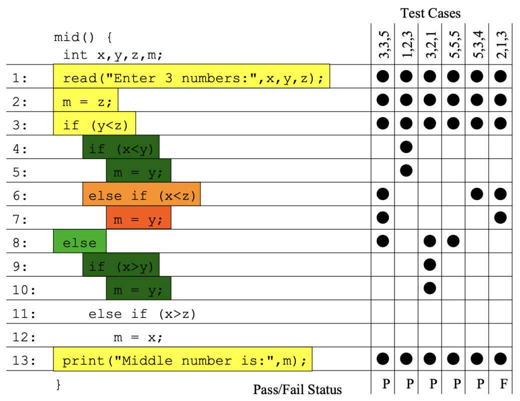


Fig. 4: Control Flow Graph for Faulty Program Example



	mid() { int x,y,z,m;	Test Cases 3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
1:	`read("Enter 3 numbers:",x,y,z);`	●	●	●	●	●	●
2:	`m = z;`	●	●	●	●	●	●
3:	`if (y<z)`	●	●	●	●	●	●
4:	`if (x<y)`		●				
5:	`m = y;`		●				
6:	`else if (x<z)`	●				●	●
7:	`m = y;`	●					●
8:	`else`	●		●	●		
9:	`if (x>y)`			●			
10:	`m = y;`			●			
11:	`else if (x>z)`						
12:	`m = x;`						
13:	`print("Middle number is:",m);`	●	●	●	●	●	●
	} Pass/Fail Status	P	P	P	P	P	F

Fig. 5: Straight Line Format for Faulty Program Example [Jones et al.(2002)]

statement s as text inside.

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 2 with dynamic testing information.

Moreover, for determining the opacity of the code block, we use the precise relative value generated by Equation 2 instead of using the actual value and segmenting the values into 10 partitions imprecisely, as in the Tarantula technique. 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 [Jones et al.(2002)]. Whereas the Tarantula technique (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 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.

B. 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. As shown in Algorithm 1, our proposed algorithm could be applied to all SBFL techniques, returning the modified versions.

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 $\%\text{passed}(s)$ and $\%\text{failed}(s)$ 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 $\%\text{passed}(s)$ and $\%\text{failed}(s)$, and this conditional statement will not be highly ranked. Similarly, this will also influence all techniques' suspiciousness calculation formulas, shown in Table I.

1) Algorithm: In our modified technique, which we call +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. Specifically, this could be done by following the Algorithm 1 and 2. Then, we use the collected information of the number of passing and failing test cases for both edges to compute the suspiciousness of containing the faults separately, and then we can perform each of the following computation:

1) +Edges_max, by choosing the higher one – either the True branch's value or the value of the False branch; or
2) +Edges_mean, by computing the average of the two branches' value

to be the overall value of the suspiciousness for this conditional statement.

Algorithm 1 Modify the code line number

```
Input: code line
Output: modified code line
1: $numb$ = the number of the code line ▷ i.e. #246
   CHECK and MODIFY ▷ For if statement
2: if code line contains if or else if then
3:     Mark the current line as $numb.0$ ▷ i.e. #246.0
4:     if there is an else then
5:         Mark the else line as $numb.1$ ▷ i.e. #246.1
6:     else
7:         Create a new line as $numb.1$ ▷ i.e. #246.1
8:     end if
9: end if
```

Algorithm 2 Record the coverage information

```
Input: code line and coverage information
Output: modified coverage information
1: $numb$ = the number of the code line ▷ i.e. #246.0
2: if $numb$ contains .0 then
3:     if $numb$ coverage == 1 then
4:         $numb.1$ coverage = 0
5:     else
           $numb.1$ coverage = 1
6:     end if
7: end if
```

2) Example: For example, in the code shown in Table II , 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 IV, 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 III, 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.

IV. Evaluation

To evaluate the performance of our proposed +Edges approach on those state-of-the-art SBFL techniques, we compare the differences between the original SBFL techniques and those with the +Edges modifications with 262 real faults from Defects4J from 5 different open source libraries.

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 [Just et al.(2014)]. It includes real 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.

TABLE II: 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 III: 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

We evaluate the +Edges modification for the following SBFL techniques that include tarantula, Ochiai, Barinel, DStar, Jacuard, O^p, Russel, and Sørensen. For +Edges, we include both +Edges_max and +Edges_mean as two different ways for computing the suspiciousness score for the edge code line. We evaluate the performance of +Edges_max and +Edges_mean on different techniques and also on different source code libraries.

A. Metrics

In this study, we aim to evaluate the performance of the +Edges modification for those SBFL techniques. This is directly proportional to how much it helps users to locate the fault. The better the technique can minimize the amount of work for bug localization, the better the technique is. Previous works [Xuan and Monperrus(2014)], [Lou et al.(2021)], [Li et al.(2019)] widely use Top-n, Mean Average Rank (MAR), and Mean First Rank (MFR) as metrics for software fault localization techniques. Therefore, we will use the following three metrics to compare the effectiveness of adding +Edges modification.

Note that we will use the worse ranking when there are multiple code lines with the same ranking if there are both faulty code line and correct code line with this ranking. If there are 3 code lines ranked at n-th position, one of them is a faulty line and two of them are correct, then the ranking of the faulty code line(s) is $n+2$-th.

TABLE IV: Suspiciousness Ranking for Each Code Line in Table II

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%

1) **Mean Average Rank (MAR):** For all faulty versions in Defects4J, MAR computes the average ranking for all faulty code lines [Chen et al.(2024)]. We then compute the mean within each library and compare different techniques. This is computed as:

$$MAR = \frac{1}{nm}\sum_{i=1}^{n}\sum_{j=1}^{m} rank_i(j) \quad (4)$$

where n is the number of faulty versions each library has in Defects4J, m as the number of faulty code line in each faulty version, and $rank_i(j)$ indicates that in the code line ranking we computed for faulty version i of a library, the ranking of the faulty code line j. The smaller MAR is, the better performance the SBFL technique has.

2) **Mean First Rank (MFR):** When we try to solve for a specific fault, locating a specific place to start can always help. The MFR computes the average of the first rank, which is, among all faulty code lines in a faulty version, the rank of the faulty code line the highest ranking. This is computed as:

$$MFR = \frac{1}{n}\sum_{i=1}^{n} min\{rank_i(j)\} \quad (5)$$

Similarly, the better MFR is, the better performance the SBFL technique has.

B. Methodology

In order to evaluate the effectiveness of +Edges modification, we compared the different techniques with and without the +Edges modification in 262 real faults from Defects4J, specifically 27 from Time, 26 from Chart, 65 from Lang, 106 from Math, and 38 from Mockito.

We first generate the code line information with the matrix information provided by Defects4J, corresponding to different test cases for each fault. We then modify the code line information by the Algorithm 1. For each fault, we will generate the suspiciousness scores by different techniques for each code line, using the code lines' coverage information for passing and failing test cases. After generating the sorted list of code lines based on the suspiciousness score for each code line, we then locate its faulty code lines' ranking.

C. Experimental Results

1) Time: As shown in Table V, our experiments show that +Edges modification generally enhance fault localization effectiveness on the Time dataset, though the optimal edge configuration varies across different techniques. The comparisons between +Edges_mean and +Edges_max performances reveal that +Edges_mean has a better performance. The mean configuration produces significant improvements for the majority of techniques, with Sørensen exhibiting the most substantial enhancement (27% reduction in MFR from 1501.15 to 1087.35 and 22% reduction in MAR from 1507.0583 to 1167.8817). We have observed a similar improvement pattern consistently holding for Ochiai, Barinel, DStar, Jaccard, and Russell, with improvement ranging from 7% to 25% for MFR and from 6% to 21% for MAR. However, for the Tarantula technique, +Edges_max has a better performance compared to +Edges_mean. The O^p technique's performances are not improved after either +Edges modification.

2) Chart: The experimental results for the Chart dataset, shown in Table VI demonstrate that +Edges modifications generally enhance fault localization effectiveness, with +Edges_mean proving superior for most techniques. Notably, Sørensen shows the most substantial improvements under +Edges_mean, achieving 37.3% and 37.2% reductions in MFR and MAR respectively, while other techniques like Ochiai, Barinel, Jaccard, O^p, and Russell exhibit consistent gains ranging from 6.6% to 15.9% for MFR and from 9.3% to 15.7% for MAR. In contrast, +Edges_max benefits only Tarantula and Barinel marginally (about 1.0% improvement) while degrading performance for most other methods, particularly Russell and Sørensen (¿40% deterioration). These findings suggest that +Edges_mean should be the preferred configuration for most techniques, with +Edges_max reserved for specific cases like Tarantula where it demonstrates limited but measurable benefits.

3) Lang: The experimental results for the Lang dataset demonstrate a similar trend, that +Edges_mean significantly enhances fault localization performance for most techniques, with Sørensen showing the most dramatic improvements (55.8% reduction in MFR and 43.0% reduction in MAR), shown in Table VII. Similar substantial gains are observed for Russell (52.2% MFR improvement) and O^p (24.4%), while other techniques like Ochiai, Barinel, and Jaccard achieve consistent improvements ranging from 14.5% to 17.4%. In contrast, +Edges_max provides more modest benefits, with the best results seen in Tarantula (8.2% MFR improvement) and Barinel (9.5% for MFR). Notably, both edge modifications degrade performance for O^p, Russell, and Sørensen when using the max configuration, with performance drops exceeding 10%. These findings suggest that +Edges_mean is generally more effective for the Lang dataset, particularly for techniques like Sørensen and Russell, while +Edges_max offers limited advantages for specific methods like Tarantula. The consistent performance degradation under +Edges_max for certain techniques highlights the importance of selecting the appropriate edge modification strategy based on both the technique and dataset characteristics.

4) Math: The experimental results for the Math dataset demonstrate three key findings about edge modifications. First, all techniques show improved fault localization performance

TABLE V: Comparison Across Techniques on Defects4J: Time Dataset Results

Metric Type	Technique	Fault Localization Technique Scores							
		Tarantula	Ochiai	Barinel	DStar	Jaccard	O^p	Russell	Sørensen
Time-MFR Evaluation									
	MFR	74.5	72.15	74.5	68.1	72.15	160.0	1501.15	1501.15
	MFR-mean	113.8↑	63.5 ↓	68.8 ↓	62.55 ↓	68.0 ↓	212.8 ↑	1117.35 ↓	1087.35 ↓
	MFR-max	73.7↓	71.35 ↓	73.7 ↓	67.25 ↓	71.35 ↓	162.9 ↑	1545.4 ↑	1545.4 ↑
Time-MAR Evaluation									
	MAR	78.2625	76.5708	78.2625	73.4142	77.9808	260.3358	1507.0583	1507.0583
	MAR-mean	215.0217↑	70.4183↓	73.4525↓	68.6292↓	73.1658↓	436.9933↑	1185.8017↓	1167.8817↓
	MAR-max	77.4625↓	75.7708↓	77.4625↓	72.5642↓	77.1808↓	261.9758↑	1549.9483↑	1549.9483↑

Note: ↓indicates superior performance with +edge modification; ↑indicate inferior performance. Green highlights show which +edges modification has better performance for Time using the specific technique, either +Edges_mean or +Edges_max. All values represent Time metrics.

TABLE VI: Comparison Across Techniques on Defects4J: Chart Dataset Results

Metric Type	Technique	Fault Localization Technique Scores							
		Tarantula	Ochiai	Barinel	DStar	Jaccard	O^p	Russell	Sørensen
Chart-MFR Evaluation									
	MFR	94.4348	165.5833	90.8333	201.4091	163.0000	274.4583	630.2083	630.2083
	MFR-mean	93.6957↓	145.3750↓	84.8750↓	180.8636↓	143.1667↓	230.9167↓	533.5000↓	395.0833↓
	MFR-max	93.4783↓	164.958 ↓	89.9583↓	236.0909↑	164.3333↑	321.0417↑	907.7500↑	907.7500↑
Chart-MAR Evaluation									
	MAR	159.9652	183.7333	153.6333	208.5098	181.6833	282.8861	631.8444	631.8444
	MAR-mean	161.5420↑	154.8264↓	139.2903↓	212.9568↑	160.1708↓	251.5333↓	535.3500↓	396.9333↓
	MAR-max	159.1942↓	199.5972↑	152.9361↓	242.3886↑	197.9000↑	327.1167↑	909.6000↑	909.6000↑

Note: ↓indicates superior performance with +edge modification; ↑indicate inferior performance. Green highlights show which +Edges modification has better performance for Chart using the specific technique, either +Edges_mean or +Edges_max. All values represent Chart metrics.

after applying either +Edges_mean or +Edges_max modifications, with Sørensen exhibiting the most dramatic gains (48.1% MFR improvement with mean edges). Second, the Mean First Rank (MFR) metric is consistently better optimized by +Edges_mean across all techniques, with improvements ranging from 11.2% (Barinel) to 48.1% (Sørensen). Third, while Mean Average Rank (MAR) shows mixed preferences between edge configurations, the performance differences are relatively minor when +Edges_max performs better - for instance, Tarantula achieves 15.0% MAR improvement with mean edges versus 14.9% with max edges, and similar small margins are observed for other techniques. This suggests that while +Edges_mean should generally be preferred for comprehensive optimization, +Edges_max remains a viable alternative for specific techniques where its performance approaches that of the mean configuration.

5) Mockito: According to Table IX, building on the consistent patterns observed across datasets, the Mockito results further validate the effectiveness of edge modifications for fault localization. Like the Math and Lang datasets, Mockito shows universal performance improvements with edge modifications, with Sørensen again demonstrating the most dramatic gains (28.7% MFR improvement with +Edges_mean) - a pattern consistent with its 48.1% improvement in Math and 55.8% in Lang. The superior performance of +Edges_mean for MFR optimization holds true across all three datasets, with Mockito's improvements (14.4-28.7%) falling between Lang's (14.5-55.8%) and Math's (11.2-48.1%) ranges. Similarly, the mixed but minor advantages of +Edges_max for MAR in Mockito (e.g., Tarantula's 6.2% improvement) mirror the subtle differences seen in previous datasets, where max configurations showed limited but measurable benefits for specific techniques. These cross-dataset consistencies strengthen the recommendation that +Edges_mean should be the default configuration, while acknowledging that +Edges_max may offer situation-specific advantages, particularly for techniques like Tarantula that show comparable performance with both configurations across multiple datasets.

D. Discussion

Our experimental results across five datasets reveal several important patterns about edge modifications in fault localization. The findings consistently show that incorporating edge information improves diagnostic accuracy, though the degree

TABLE VII: Comparison Across Techniques on Defects4J: Lang Dataset Results

Metric Type	Technique	Fault Localization Technique Scores							
		Tarantula	Ochiai	Barinel	DStar	Jaccard	O^p	Russell	Sørensen
Lang-MFR Evaluation									
	MFR	44.3043	41.2826	44.9348	34.2500	40.6957	47.4348	187.4348	187.4348
	MFR-mean	37.0652↓	34.4130↓	38.3913↓	29.7750↓	33.6087↓	35.8478↓	89.5870↓	82.8043↓
	MFR-max	40.6522↓	37.1957↓	40.6522↓	30.5750↓	36.4348↓	50.7826↑	212.9348↑	212.9348↑
Lang-MAR Evaluation									
	MAR	52.0995	54.3685	52.7300	52.7263	49.1518	62.1587	191.6679	191.6679
	MAR-mean	44.3593↓	41.8225↓	44.9082↓	47.3535↓	41.1153↓	65.2736↑	123.0628↓	109.2657↓
	MAR-max	48.6655↓	54.4155↑	48.6655↓	54.5972↑	45.1244↓	68.9475↑	217.5616↑	217.5616↑

Note: ↓indicates superior performance with +edge modification; ↑indicate inferior performance. Green highlights show which +Edges modification has better performance for Lang using the specific technique, either +Edges_mean or +Edges_max. All values represent Lang metrics.

TABLE VIII: omparison Across Techniques on Defects4J: Math Dataset Results

Metric Type	Technique	Fault Localization Technique Scores							
		Tarantula	Ochiai	Barinel	DStar	Jaccard	O^p	Russell	Sørensen
Math-MFR Evaluation									
	MFR	64.6333	64.3833	64.6333	75.4468	63.5333	104.2167	483.0500	483.0500
	MFR-mean	54.4000↓	55.9833↓	57.3667↓	63.8085↓	56.1167↓	70.9833↓	260.9500↓	250.6667↓
	MFR-max	61.3000↓	61.5167↓	61.3000↓	69.8511↓	60.4000↓	86.1167↓	406.8667↓	406.8667↓
Math-MAR Evaluation									
	MAR	120.0215	122.3379	120.0215	154.1166	121.4139	166.9173	518.8162	518.8162
	MAR-mean	101.9866↓	107.9282↓	99.6930↓	135.6632↓	103.6754↓	147.8363↓	334.2301↓	319.2845↓
	MAR-max	102.1211↓	107.7478↓	102.1211↓	134.2482↓	103.5278↓	140.6561↓	440.3250↓	440.3250↓

Note: ↓indicates superior performance with +edge modification; ↑indicate inferior performance. Green highlights show which +Edges modification has better performance for Math using the specific technique, either +Edges_mean or +Edges_max. All values represent Math metrics.

of improvement depends on both the technique and the type of edge modification used.

The most significant finding is the superior performance of +Edges_mean across nearly all techniques and datasets. This configuration reliably enhanced fault localization, with particularly strong results for Sørensen (improving by 27-55.8% across datasets) and Russell techniques. The mean-based approach appears particularly effective because it smooths out irregularities in test coverage data while preserving important diagnostic signals. Only Tarantula showed better results with +Edges_max, suggesting its unique ranking formula benefits more from extreme coverage values.

We observed interesting variations between datasets that merit attention. While the general patterns held, the magnitude of improvement differed - Lang showed the most dramatic gains (up to 55.8%), while Mockito's improvements were more modest (up to 28.7%). These differences likely reflect variations in test suite characteristics and fault types across projects. The consistent exception was O^p, which either showed minimal improvement or degraded with edge modifications, indicating it may require a different optimization approach.

We also noticed that even though the lower MFR and MAR values allow the users to locate the fault and start debugging successfully after applying the +Edges modification, some specific techniques have a slightly worse performance for the specific datasets.

For those cases, 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 the buggy versions 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

TABLE IX: Comparison Across Techniques on Defects4J: Mockito Dataset Results

Metric Type	Technique	Fault Localization Technique Scores							
		Tarantula	Ochiai	Barinel	DStar	Jaccard	O^p	Russell	Sørensen
Mockito-MFR Evaluation									
	MFR	63.4815	56.7778	63.4815	57.5000	62.6667	122.5926	594.3333	594.3333
	MFR-mean	52.1111$\downarrow$	42.5926$\downarrow$	54.3333$\downarrow$	43.5000$\downarrow$	53.1481$\downarrow$	90.8889$\downarrow$	431.3333$\downarrow$	423.7407$\downarrow$
	MFR-max	59.2963$\downarrow$	52.6296$\downarrow$	59.2963$\downarrow$	53.1923$\downarrow$	58.3333$\downarrow$	112.1481$\downarrow$	545.8889$\downarrow$	545.8889$\downarrow$
Mockito-MAR Evaluation									
	MAR	91.7125	91.1323	91.7125	95.8361	90.9912	150.8042	595.1138	595.1138
	MAR-mean	93.5732$\uparrow$	73.2050$\downarrow$	80.1799$\downarrow$	77.7958$\downarrow$	79.1649$\downarrow$	142.9881$\downarrow$	439.4475$\downarrow$	434.6698$\downarrow$
	MAR-max	86.0366$\downarrow$	85.1548$\downarrow$	86.0366$\downarrow$	89.6488$\downarrow$	85.1671$\downarrow$	138.8276$\downarrow$	546.3951$\downarrow$	546.3951$\downarrow$

Note: $\downarrow$indicates superior performance with +edge modification; $\uparrow$indicate inferior performance. Green highlights show which +Edges modification has better performance for Mockito using the specific technique, either +Edges_mean or +Edges_max. All values represent Mockito metrics.

	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 X: Code Example #3. The fault is on line 5.

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%

TABLE XI: Suspiciousness Rankings for Elements in Code Example #3

edge also being taken.

Because most of the test methods we use are math methods, most if-else conditions in the code have correlation relationships. Table X 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 XI. We can notice that by the Tarantula technique, we only need to read lines 2 and 5; however, by the Tarantula+Edges technique, 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 technique is likely to contain extra code lines, which are ranked before the faulty line. This is why for those methods, the Tarantula+Edges technique is not better than the original Tarantula or the Set-union technique.

V. Limitations

The +Edges modification 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 technique 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 [Wong et al.(2016)]. 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.

VI. 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

[Abreu et al.(2009b)] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J.C. van Gemund. 2009b. 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.

[Abreu et al.(2006)] 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

[Abreu et al.(2007a)] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2007a. 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

[Abreu et al.(2007b)] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2007b. 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

[Abreu et al.(2009a)] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2009a. Spectrum-Based Multiple Fault Localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 88–99. https://doi.org/10.1109/ASE.2009.25

[Balzer(1969)] 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

[Chen et al.(2024)] Xin Chen, Tian Sun, Dongling Zhuang, Dongjin Yu, He Jiang, Zhide Zhou, and Sicheng Li. 2024. HetFL: Heterogeneous Graph-Based Software Fault Localization. *IEEE Transactions on Software Engineering* 50, 11 (2024), 2884–2905. https://doi.org/10.1109/TSE.2024.3454605

[Collofello and Woodfield(1989)] 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

[de Souza et al.(2016)] 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).

[Gouveia et al.(2013)] 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

[Jones et al.(2002)] 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

[Jones and Harrold(2005)] 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

[Just et al.(2014)] 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

[Kauffman et al.(2024)] 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

[Li et al.(2019)] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 169–180. https://doi.org/10.1145/3293882.3330574

[Lou et al.(2021)] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 664–676. https://doi.org/10.1145/3468264.3468580

[Naish et al.(2011)] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.* 20, 3, Article 11 (Aug. 2011), 32 pages. https://doi.org/10.1145/2000791.2000795

[Ribeiro et al.(2018)] 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

[Ribeiro et al.(2019)] 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

[Russell and Rao(1940)] P. F. Russell and T. Ramachandra Rao. 1940. On Habitat and Association of Species of Anopheline Larvae in South-eastern Madras. 3 (1940), 153–178 pp. Issue 1.

[Sarhan and Beszédes(2022)] 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

[Sørensen(1948)] T. Sørensen. 1948. *A Method of Establishing Groups of Equal Amplitude in Plant Sociology Based on Similarity of Species Content and Its Application to Analyses of the Vegetation on Danish Commons*. Munksgaard. https://books.google.com/books?id=rpS8GAAACAAJ

[Tiwari et al.(2011)] 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

[Wong et al.(2014)] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2014. The DStar Method for Effective Software Fault Localization. *IEEE Transactions on Reliability* 63, 1 (2014), 290–308. https://doi.org/10.1109/TR.2013.2285319

[Wong et al.(2016)] 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

[Xuan and Monperrus(2014)] Jifeng Xuan and Martin Monperrus. 2014. Learning to Combine Multiple Ranking Metrics for Fault Localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 191–200. https://doi.org/10.1109/ICSME.2014.41

[Zhang and Zhang(2014)] 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

[Zhang et al.(2009)] 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