Int J Syst Assur Eng Manag (September 2024) 15(9):4616–4635
https://doi.org/10.1007/s13198-024-02479-5

ORIGINAL ARTICLE

Effective fault localization using probabilistic and grouping approach

Saksham Sahai Srivastava[1] · **Arpita Dutta**[2] · **Rajib Mall**[3]

Received: 8 August 2023 / Revised: 7 August 2024 / Accepted: 9 August 2024 / Published online: 18 August 2024

Abstract Fault localization (FL) is the key activity while debugging a program. Any improvement to this activity leads to significant improvement in total software development cost. In the paper, we present a conditional probability statistics based fault localization technique that derives the association between statement coverage information and test case execution result. This association with the failed test case result shows the fault containing probability of that specific statement. Subsequently, we use a grouping method to refine the obtained statement ranking sequence for better fault localization. We named our proposed FL technique as CGFL, it is an abbreviation of Conditional probability and Grouping based Fault Localization. We evaluated the effectiveness of the proposed method over eleven open-source data sets from Defects4j and SIR repositories. Our obtained results show that on average, the proposed CGFL method is 24.56% more effective than contemporary FL techniques namely D*, Tarantula, Ochiai, Crosstab, BPNN, RBFNN, DNN, and CNN.

Keywords Fault localization · Program analysis · Debugging · Conditional probability · Grouping

✉ Saksham Sahai Srivastava
saksham.srivastava@colorado.edu

Arpita Dutta
arpita@comp.nus.edu.sg

Rajib Mall
rajib@cse.iitkgp.ac.in

[1] Department of Computer Science, University of Colorado Boulder, Boulder, CO 80302, USA

[2] School of Computing, National University of Singapore, Computing Dr, Singapore 117417, Singapore

[3] Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur, Kharagpur 721302, India

1 Introduction

Software development comprises two immensely important as well as labor-intensive components of software testing and debugging. With the advancement in software development, software are growing in scale as well as in complexity, due to which execution failure can be considered inevitable. During software maintenance (Wong et al. 2016), software debugging turns out to be the most arduous task. Software debugging comprises two major activities: localization of fault and the rectification of the fault. Fault localization (FL) is the process of uncovering faults present in different regions of the program which have been responsible for execution failures. There have been multiple attempts to design automated techniques which can achieve the objective of fault localization. Any improvement to these pre-existing automated techniques would significantly reduce total software maintenance cost and also debugging time (Mall 2018).

In the past two-to-three decades, several automated FL techniques have been reported for effective localization of faults with reduced human intervention. These proposed approaches aimed at localizing the faults by examining a small fraction of the code because the lesser the code would be examined, the better would be the effectiveness of the technique. The fault localization techniques can be broadly classified into these categories: slicing-based (Weiser 1984; Korel and Laski 1988), spectrum-based (SBFL) (Wong et al. 2008; Jones et al. 2002; Wong et al. 2010, 2013), machine learning-based (Ascari et al. 2009; Dutta et al. 2019; Wong and Qi 2009; Wong et al. 2011), and mutation-based techniques (Papadakis and Le Traon 2015; Moon et al. 2014; Dutta and Godboley 2021; Dutta et al. 2021). There are three types of slicing based techniques: static (Weiser 1984), dynamic (Korel and Laski 1988), and execution slice-based methods (Agrawal et al. 1995). Slicing-based techniques

worked predominantly on the principle of deleting irrelevant segments of the program, thereby making the remainder of the program behave in a similar fashion as previously (without deletion) with respect to certain specifications. Spectrum Based Fault Localization (SBFL) techniques (Wong et al. 2013; Jones and Harrold 2005) by and large require program spectra information to give quantitative knowledge for localizing faults in a program. The program spectra include program element (such as statement, block, etc.) coverage information (executed/not executed) and test case execution results (pass/fail) which are supplied as input to the fault localizer. The fault localizer is expected to generate a ranked list of statements based on their suspicious scores of containing a bug. Machine learning which helps formulate mathematical models for complex problems also finds its application in solving the problem of fault localization. Therefore, there are machine learning based techniques such as SVM (Ascari et al. 2009), ensemble classifier (Dutta et al. 2019), decision trees (Briand et al. 2007) and neural network models (Wong and Qi 2009; Wong et al. 2011; Xiao et al. 2021; Li et al. 2021; Lou et al. 2021) which manifested promising results for FL. The mutation based techniques such Metallaxis (Papadakis and Le Traon 2015), MUSE (Moon et al. 2014), MSFL (Dutta and Godboley 2021) and Combi-FL (Dutta et al. 2021) have also presented substantial improvement in localizing faults effectively.

But these antecedent approaches hold certain drawbacks. Slicing-based techniques (Weiser 1984; Korel and Laski 1988) possess the disadvantage of not assigning ranks to the program statements while the SBFL techniques (Jones and Harrold 2005; Naish et al. 2011) generate a ranked list with many statements having tie in ranks. The machine learning-based techniques (Ascari et al. 2009; Dutta et al. 2019; Wong and Qi 2009; Wong et al. 2011; Xiao et al. 2021; Li et al. 2021; Lou et al. 2021) although being effective, require a plenty amount of time to complete the computation and therefore have low efficiency. The mutation-based techniques (Papadakis and Le Traon 2015; Moon et al. 2014; Dutta and Godboley 2021; Dutta et al. 2021) in addition to having low efficiency, entail a considerable amount of space which sometimes makes it practically infeasible for large-sized programs. By looking at the limitations of the existing techniques, the key objective of this work is to come up with a fault localization scheme which would be both effective and efficient. Effective in terms of number of statements required to be inspected and efficient as per the amount of the time required by the localization algorithm to generate the ranked list of statements.

Statistical models have been immensely manipulated in the past for accomplishing the objective of better fault localization. Crosstab proposed by Wong et al. (2008) and FTFL proposed by Dutta et al. (2021) have highlighted the dominance of statistical models in producing effective fault localizers. Unlike existing (Jones et al. 2002; Wong et al. 2010) intuitive guesswork or heuristics-based FL techniques, we use the idea of conditional probability table and compute the suspiciousness for each statement. Conditional probability table (Dekking et al. 2006) represents the local probability distribution for a statement in a given program and test case execution results. Therefore, in this article, we extensively employ conditional probability statistics to evaluate the suspicious score of the program statements. Conditional Probability statistic (Yang et al. 2019) portrays it's significance by highlighting the internal linkage between the program spectrum and test execution results. Further, we make use of the grouping method to refine the rank of the buggy statement. It is also a relatively simpler statistical model as compared to Crosstab (Wong et al. 2008) and FTFL (Dutta et al. 2021). Hence, we name our proposed method as Conditional Probability and Grouping based Fault Localization technique(hereafter referred as CGFL). The main novelty of this work is the usage of a lightweight statistical method accompanied with grouping methodology. Our CGFL method efficiently generates ranked list of suspicious program statements and attains the aim of effective fault localization.

Rest of the paper is organized as follows. In Sect. 2, we present a survey of related literature. We discuss our proposed approach in Sect. 3. In Sect. 4, we elaborate upon the experimental result followed by the comparison of our proposed technique with related work in Sect. 5. Finally, we conclude in Sect. 6.

2 Related work

Weiser introduced the concept of static slicing (Weiser 1984). Static slicing (Weiser 1984) was based on the design that if a test case fails in response to a variable attaining the wrong value at a statement, then the slice associated with the variable-statement pair would be held accountable for the resulting defect. Later, Korel and Laski (1988), proposed dynamic slicing which eliminated the drawback of static slicing where all the executable statements which could potentially be affected by the value of a variable at a statement were included. Hence, in dynamic slicing (Korel and Laski 1988) only those statements were included which actually influenced the value of a variable at a statement. But in some cases it may so happen that the slice returned is very large(sometimes even the size of the entire program) which defeats the overall purpose of using this technique.

Next, Spectrum Based Fault Localization(SBFL) methods were proposed which gradually became popular. The SBFL techniques have low computation costs as these methods generally require a mathematical expression for the computation of suspicious score for each executable program

statement. They also utilize various statistics to produce a mathematical model which could determine the suspicious score of program statements. Tarantula is one of the most prominent SBFL techniques proposed by Jones and Harrold (2005). It outperformed set union (Renieris and Reiss 2003), set intersection (Renieris and Reiss 2003), nearest neighbour (Renieris and Reiss 2003), and cause transition (Cleve and Zeller 2005) which were some of the pre-existing effective FL techniques at that time. But eventually, it was realized that Tarantula was incapable of perfectly exploiting the relevant information carried by the successful and failed test case execution results. To overcome the shortcomings of Tarantula (Jones and Harrold 2005), several other SBFL approaches have been proposed. These included Ample (Wong et al. 2016), Jaccard (Wong et al. 2010), Ochiai (Naish et al. 2011), Barinel and DStar(D*) (Wong et al. 2013), etc. Ample (Wong et al. 2016) and Jaccard (Wong et al. 2010) use failed as well as passed test cases into consideration while calculating the suspicious score of statements. Ochiai (Naish et al. 2011), which is basically obtained from the domain of molecular biology, puts a greater emphasis on the failed test cases which were responsible for not executing the statement. Wong et al. (2013) developed DStar(D*) which showed better performance as compared to all the pre-existing techniques and thus became the state-of-the-art technique. The value of '*' in D* was chosen as 2 as it exhibited the best results. But since the DStar technique took a small number of parameters into consideration while calculating the suspicious score, many program statements were assigned the same rank.

Machine Learning (ML) played a key role in the further advancement of FL techniques. The highly adaptive nature of machine learning algorithms helped produce robust models which were found to be very effective. ML techniques generally, learn the program spectra and execution results and generate suspiciousness scores for program elements based on their learning. Back Propagation Neural Network (BPNN) technique proposed by Wong and Qi (2009) is one of the most prevalent machine learning-based FL techniques. It has easy implementation due to its very fundamental structure. But it was noticed that BPNN (Wong and Qi 2009) suffered from problems of local minima (Tan et al. 2013) and paralysis (Wasserman 1993). Hence to resolve this issue, Wong et al. (2011) proposed the Radial Basis Function Neural Network(RBFNN) technique which had radial basis function instead of sigmoid function as its transfer function. However, BPNN (Wong and Qi 2009) and RBFNN (Wong et al. 2011) were not found effective in handling large complex functions because of their shallow architecture. Hence, Zheng et al. (2016) develop Deep Neural Network(DNN) (Zheng et al. 2016) model which made working with complex functions relatively simpler. Later, Dutta et al. (2019) further refined the technique by proposing a hierarchical approach of FL using DNN where they initially investigated the functions for containing a fault and then the statements. But these neural network-based approaches desired training of large number of parameters which demonstrated a very complex model for solving the fault localization problem.

In last four to five years, several deep learning-based FL techniques such as DeepFL (Li et al. 2019) and DeepRL4FL (Li et al. 2021) have been proposed. These FL techniques incorporates several information obtained from different traditional FL methods, for example the suspiciousness scores calculated from the SBFL and MBFL techniques, text similarity, static code metrics etc. These techniques utilizes the learning capability of neural networks to train the classification models for accurately localizing the faulty program entities. DeepFL (Li et al. 2019) make use of the synthetically designed features resulting in underutilization of program spectra, rather than considering the contextual information between program entities. Also, this technique uses several complex features e.g. spectrum-based suspiciousness and complexity-based fault proneness which results in higher overhead for obtaining the required information.

In the recent past, researchers have laid emphasis on the usage of mutation analysis for the purpose of fault localization. Papadakis and Le Traon (2015) introduced the mutation-based FL technique known as Metallaxis-FL (Papadakis and Le Traon 2015), where they generated mutants of the program in such a manner that if a generated mutant was killed by the failed test case then that particular mutant would provide an excellent indication of the faulty location in the program. Later, Moon et al. (2014) developed a technique called MUSE (Moon et al. 2014), which primarily identified faulty statements by making use of the different characteristics in two broad groups of mutants. The first group consisted of mutants which were generated by mutating the faulty statement and the second group comprised of mutants generated by mutating the non-faulty statements of the program. Also, Dutta et al. proposed MBFL techniques such as MSFL (Dutta and Godboley 2021) and Combi-FL (Dutta et al. 2021) which suggested significant improvement to the pre-existing mutation-based techniques. Although these mutation-based fault localization techniques proved to be effective, they required a huge computation cost due to the large number of mutants generated for large programs. Not only that, these heavy computations also lowered the efficiency of the technique.

3 Proposed work: CGFL

The necessity of fault localization arises as soon as a test case execution failure is reported by the program. It signals that the program is faulty. The very first step towards addressing the problem of fault localization is to execute a

Table 1 Notations used in this paper

τ	Total number of test cases
τ_f	Total number of failed test cases
τ_s	Total number of successful test cases
$\tau_c(\zeta)$	Number of test cases covering ζ
$\tau_{cf}(\zeta)$	Number of failed test cases covering ζ
$\tau_{cs}(\zeta)$	Number of successful test cases covering ζ
$\tau_u(\zeta)$	Number of test cases not covering ζ
$\tau_{uf}(\zeta)$	Number of failed test cases not covering ζ
$\tau_{us}(\zeta)$	Number of successful test cases not covering ζ

large number of test cases on the faulty program. Subsequently, the program spectra information is extracted. The "program spectra" or "program spectrum" is an execution profile that indicates which parts of a program are active during a run (Harrold et al. 1998). In our proposed CGFL technique, we used statement coverage information as the program spectra to gather the execution profile of statements present in the program during a test case run. Program spectra helps to determine the linkage of program element with the test case execution result. Program spectra along with the test case execution results are input to a fault localization technique to calculate the suspiciousness of program elements. The fault localization technique utilizes the program spectra information to generate a ranked list of program elements based on their suspicious scores. For simplicity, we have chosen executable statements as the program entity. In this section, we first describe our proposed technique CGFL. Subsequently, we present an illustration of our proposed approach using an example program. Table 1 characterizes the notations used in this paper. We denote the executable program statement with the symbol ζ.

3.1 Overview

The internal linkage between the test case execution result and program spectrum is the fundamental basis for constructing any statement suspiciousness calculating formula. It is always important as well as essential to capture all possible correlations between statement coverage (covered/uncovered) and test result (pass/fail). Conditional probability in statistics effectively captures the likelihood of the occurrence of an event based on the the occurrence of a previous outcome or event. The conditional probabilities provide extremely useful information, even when limited information is provided. In this paper, we design four conditional probability models to capture the association between the program spectrum and test execution results. Using these four models and the concepts of the occurrence of low-probability events in the information theory, we define a new probability based fault localization technique.

We construct a probabilistic model to depict the association of test case execution result(i.e., pass or fail) with the execution of program statements by that particular test case and vice versa as well. A statistic(denoted by ψ) is designed for each dependency relationship. We define 4-ψ statistics which are capable enough to consider all relevant dependency scenarios. In the definition of 4-ψ statistics, C denotes that the statement was executed, U denotes that the statement was not executed, F denotes that the test case was failed and S denotes that the test case was successful. When the actual test case output is different from the expected output then the test case is considered as fail (unsuccessful) otherwise pass (successful).

1. Statistic-1 (ψ_{fc}): This statistic computes the probability of the test case failure when it is known that the statement has been executed by the test case. Mathematically, it can be represented as:

$$\psi_{fc}(\zeta) = P(F|C) = \frac{\tau_{cf}(\zeta)}{\tau_{cf}(\zeta) + \tau_{cs}(\zeta)}$$

where $\tau_{cf}(\zeta) + \tau_{cs}(\zeta) \neq 0$. If $\tau_{cf}(\zeta) + \tau_{cs}(\zeta) = 0$, it exhibits that the statement ζ was not covered by any of the test cases and this situation is likely to provide no information for fault localization.
2. Statistic-2 (ψ_{cf}): This model determines the probability of a test case to execute the statement ζ when it is noted that the execution of the test case resulted in a failure. Mathematically, it can be represented as:

$$\psi_{cf}(\zeta) = P(C|F) = \frac{\tau_{cf}(\zeta)}{\tau_{cf}(\zeta) + \tau_{uf}(\zeta)}$$

where $\tau_{cf}(\zeta) + \tau_{uf}(\zeta) \neq 0$. If $\tau_{cf}(\zeta) + \tau_{uf}(\zeta) = 0$, it indicates that all of the test cases have led to execution failure and such a state of condition is meaningless from that fault localization point of view.
3. Statistic-3 (ψ_{cs}): This statistic is responsible for evaluating the probability of a test case to execute the statement ζ when it is well known that the execution of the test case resulted in a success. Mathematically, it can be represented as:

$$\psi_{cs}(\zeta) = P(C|S) = \frac{\tau_{cs}(\zeta)}{\tau_{cs}(\zeta) + \tau_{us}(\zeta)}$$

where $\tau_{cs}(\zeta) + \tau_{us}(\zeta) \neq 0$. If $\tau_{cs}(\zeta) + \tau_{us}(\zeta) = 0$, it manifests that all of the test cases have lead to execution success and such a situation does not contribute any relevant information to the fault localization technique.
4. Statistic-4 (ψ_{su}): This model quantifies the probability of a test case having a successful execution result when it has already been recognized that the test case did not

execute the statement ζ. Mathematically, it can be represented as:

$$\psi_{su}(\zeta) = P(S|U) = \frac{\tau_{us}(\zeta)}{\tau_{uf}(\zeta) + \tau_{us}(\zeta)}$$

where $\tau_{uf}(\zeta) + \tau_{us}(\zeta) \neq 0$. If $\tau_{uf}(\zeta) + \tau_{us}(\zeta) = 0$, it reveals that the statement ζ was covered by all the test cases, thereby leaving little scope for the fault localization technique to extract some useful information.

These 4-ψ statistic models would be supportive in constructing the fault localizer (denoted by CPFL) which stands for Conditional Probability based Fault Localization. The fault localizer CPFL would generate the suspicious score for each of the program statements. The fault localizer CPFL is defined as follows:

$$CPFL(\zeta) = \begin{cases} -\infty, & \text{if } \psi_{fc} = 0 \text{ or } \psi_{su} = 0 \\ \psi_{fc} + \psi_{cf} + \psi_{su}, & \text{if } \psi_{fc} \neq 0 \text{ and } \psi_{su} \neq 0 \end{cases}$$

The suspicious score of the program statements with ψ_{fc} or ψ_{su} values as zero is considered to be $-\infty$. This is because $\psi_{fc} = 0$ points to a situation where the probability of a test case to fail is always zero given that it has executed a particular statement. Then, such a statement is least likely to be faulty and hence is assigned with the least priority. Similarly, when $\psi_{su} = 0$, it represents a situation where the probability of a test case to pass is always zero given that it does not execute a particular statement. Similarly, in this situation also, that particular statement holds the least probability of being faulty, and therefore, its suspicious score value is assigned as $-\infty$. The fault localizer CPFL generates a list of suspicious scores of program statements. The statements can be ranked from higher to lower based on their degree of suspiciousness. However, we have further extended the FL technique CPFL by incorporating the grouping strategy and then termed it as CGFL which represents the combination of Conditional probability and Grouping strategy for Fault Localization. The grouping strategy improves the ranking strategy of the CPFL. We demonstrate the improvement obtained in CGFL over CPFL in the experimental Sect. 4.

There is a popular intuition in fault localization that if a program statement has been covered by a large number of failed test cases then there is a high possibility that the statement may be buggy. Whereas, if the statement has been covered by a small number of failed test cases, the chance of the statement being buggy turns out to be low. Therefore, we prioritize the statements which have been executed by a large number of failed test cases for examination using the grouping strategy (Debroy et al. 2010). The program statements are grouped subsequently, such that each group consists of all the statements that are executed by a particular number of failed test cases. For example, if a program has n number of failed test cases. Then, at most $n + 1$ groups g_0, g_1, g_2,…, g_n would be formed.

The groups are created in such a manner that group g_0 contains all the statements that have not been executed by any of the failed test cases, group g_1 is a collection of all the statements that have been executed by only one failed test case and similarly group g_n consists of all the statements that have been executed by all (i.e. n) failed test cases. Also, the grouping of statements is purely based on the cardinality of the failed test cases which have executed the particular statement. Let us say, we have two statements ζ_1 and ζ_2 such that they were executed by failed test cases $\{t_1, t_3, t_7\}$ and $\{t_1, t_2, t_9\}$ respectively. Even though both the statements are executed by different sets of failed test cases yet they both would be assigned to the same group g_3(as they were executed by three failed test cases). The statements residing in the same group are then arranged in descending order of their suspicious score. The suspicious score was computed by the fault localizer CGFL.

Figure 1 represents the block diagram of proposed CGFL technique. The walkthrough of this diagram and the workflow of the complete CGFL approach is explained through the following set of steps.

Step 1: (Computation of Program Spectra and Test Results)
We first supply the input program and run it on the given test suite to generate the statement coverage information and test execution results.

Step 2: (Calculation of suspicious score)
Subsequently, statement coverage data and test results are supplied to the fault localizer CPFL. Using the constructed 4-ψ statistics, CPFL evaluates the suspicious score $CPFL(\zeta)$ for each executable program statement ζ.

Step 3: (Assignment of statements to groups)
further, each statement ζ is assigned to the group, which is responsible for holding all the statements that have been covered by a particular number of failed test cases.

Step 4: (Sorting of Groups)
The groups are sorted in such a manner that the one which is a collection of statements executed by larger number of failed test cases get a higher priority in comparison to one which holds statements executed by smaller number of failed test cases. For example, g_7 will have more preference for fault localization than g_5 or g_6.

Step 5: (Sorting of statements within each group)
The statements within each group are sorted in descending order of their suspicious score.

This completes the description of the proposed CGFL technique.

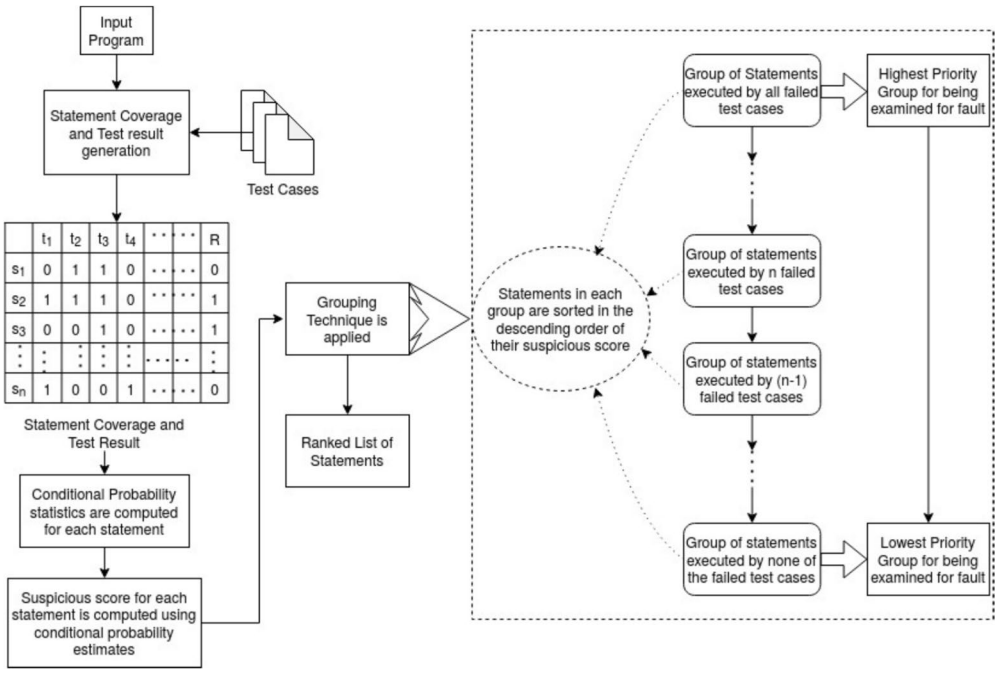

Fig. 1 Block diagram of proposed CGFL technique

Table 2 Program illustrating CGFL technique

Program	Test cases										
	1	2	2	1	2	1	2	2	3	3	2
	2	2	1	3	3	2	1	1	1	2	2
int find_mid(int p, int q, int r){	3	2	3	2	1	1	2	1	2	1	1
int mid	1	1	1	1	1	1	1	1	1	1	1
mid=r	1	1	1	1	1	1	1	1	1	1	1
if (q < r)	1	1	1	1	1	1	1	1	1	1	1
if (p > q) //bug	1	0	1	0	0	0	1	0	1	0	0
mid = q	0	0	1	0	0	0	1	0	1	0	0
else if (p < r)	1	0	0	0	0	0	0	0	0	0	0
mid = p	1	0	0	0	0	0	0	0	0	0	0
else if (p > q)	0	1	0	1	1	1	0	1	0	1	1
mid = q	0	0	0	0	0	0	0	1	0	1	0
else if (p > r)	0	1	0	1	1	1	0	0	0	0	1
mid = p	0	0	0	0	1	0	0	0	0	0	1
return mid	1	1	1	1	1	1	1	1	1	1	1
}	1	1	1	1	1	1	1	1	1	1	1
Fail/Pass	F	P	F	P	P	P	F	P	F	P	P

3.2 Example

In this section, we illustrate the working of our proposed CGFL technique using an example program. Table 2 presents a simple code snippet along with the complete statement coverage and test case execution result for the example program. Each row (starting from 4th row) of Table 2 represents an executable program statement and its corresponding coverage information for all test cases. In the Table, '1' denotes that the test case executed the statement while '0' denotes that the test case did not execute the statement. The last row of Table 2 manifests the execution result for each test case. Here, F and P denote that the corresponding test case has failed and passed respectively.

Table 3 Parameter values for the example program

Parameters	ζ_1	ζ_2	ζ_3	ζ_4	ζ_5	ζ_6	ζ_7	ζ_8	ζ_9	ζ_{10}	ζ_{11}	ζ_{12}	ζ_{13}
$\tau_{cf}(\zeta)$	4	4	4	4	3	1	1	0	0	0	0	4	4
$\tau_{cs}(\zeta)$	7	7	7	0	0	0	0	7	2	5	2	7	7
$\tau_{uf}(\zeta)$	0	0	0	0	1	3	3	4	4	4	4	0	0
$\tau_{us}(\zeta)$	0	0	0	7	7	7	7	0	5	2	5	0	0
$\psi_{fc}(\zeta)$	0.36	0.36	0.36	1	1	1	1	0	0	0	0	0.36	0.36
$\psi_{cf}(\zeta)$	1	1	1	1	0.75	0.25	0.25	0	0	0	0	1	1
$\psi_{cs}(\zeta)$	1	1	1	0	0	0	0	1	0.29	0.71	0.29	1	1
$\psi_{su}(\zeta)$	not def	not def	not def	1	0.88	0.7	0.7	0	0.56	0.33	0.56	not def	not def

Table 4 Suspicious score of program statements

Statements	ζ_1	ζ_2	ζ_3	ζ_4	ζ_5	ζ_6	ζ_7	ζ_8	ζ_9	ζ_{10}	ζ_{11}	ζ_{12}	ζ_{13}
$CPFL(\zeta)$	$-\infty$	$-\infty$	$-\infty$	3.0	2.625	1.95	1.95	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$

Table 5 Statements assigned to each group

Group no	Statements arranged in descending order of CPFL(ζ) value
g_4	$\text{CPFL}(\zeta_4) > \text{CPFL}(\zeta_1) = \text{CPFL}(\zeta_2) = \text{CPFL}(\zeta_3) = \text{CPFL}(\zeta_{12}) = \text{CPFL}(\zeta_{13})$
g_3	$\text{CPFL}(\zeta_5)$
g_2	None
g_1	$\text{CPFL}(\zeta_6) = \text{CPFL}(\zeta_7)$
g_0	$\text{CPFL}(\zeta_8) = \text{CPFL}(\zeta_9) = \text{CPFL}(\zeta_{10}) = \text{CPFL}(\zeta_{11})$

After obtaining the complete statement coverage information, the subsequent step is to determine the values of each statistic for each program statement ζ. Values of the different parameters required to compute the suspicious score of each statement are shown in Table 3.

Subsequently, the suspicious score for each executable program statement is calculated with the aid of fault localizer CPFL. Table 4 shows the computed suspicious scores for the program statements.

Now, the grouping approach is implemented for refining the rank of the buggy statement. There are 4 failed test cases and 7 successful test cases in the example program. So, there would be 5 groups formed(g_0, g_1, g_2, g_3, g_4). The statements assigned to each group are represented in Table 5. In the table, '=' and '>' symbols show the statements ζ_i and ζ_j belonging to same group have 'equal' and 'greater' CPFL score respectively.

According to obtained results for the example, it is found that the statement ζ_4 receives the highest priority for being examined for a bug. This is in resemblance to the veracity that statement ζ_4 was faulty. Therefore, this example explains the complete methodology adopted to implement the CGFL technique.

4 Experimental results

In this section, we first present the experimental setup and the data set used for experimentation. We then listed the evaluation metrics used to determine the performance of the proposed CGFL approach. Subsequently, the experimental results are discussed. Finally, we complete this section by highlighting the threats to the validity of the obtained results.

4.1 Setup

A 64-bit Ubuntu machine having specifications of 15.8 GB RAM and Intel (R) Core(TM)-i7 processor is employed for carrying out all the experiments. One set of input programs i.e., Siemens suite (SIR 2005) are written in ANSI-C format. On the other hand, Defects4J (2014), the other program suite contains Java programs. The statement coverage matrix and test case execution results are generated for each faulty program using GCOV (2002) tool for the C-format programs. GCOV is a utility tool that comes as a product of the GNU Compiler Collection suite. It is primarily used for code coverage analysis and statement-by-statement profiling of C-programs. For Defects4J (2014), we have used open-source available coverage results and other required resources in our experiment from Defects4J (2016). We have used Python language (Python 3.9.7) for scripting the developed modules. Linux powershell version-2[1] scripts were used to connect different components and to realize the user interface. Important libraries used for implementing the existing neural network based fault localization techniques implementation are Pandas, NumPy, SciPy, and Tensorflow.[2]

[1] https://github.com/PowerShell/PowerShell.

[2] https://docs.python.org/3/

Table 6 Program characteristics

S. no	Program name	LOC	No. of exec. lines	No. of flty vers	No. of test cases
1.1	Print_Tokens	565	195	7	4130
1.2	Print_Tokens2	510	200	10	4115
1.3	Schedule	412	152	9	2650
1.4	Schedule2	307	128	10	2710
1.5	Replace	521	244	32	5542
1.6	Tcas	173	65	41	1608
1.7	Tot_info	406	122	23	1052
2.1	Lang	39.8K	30.2K	64	2245
2.2	Math	45K	19K	104	3602
2.3	Mockito	27.8K	19.8K	38	5205
2.4	Time	56.2K	40.1K	26	4130

The activation functions, number of layers and neurons and the optimizers are kept as same as mentioned in the published works (Wong and Qi 2009; Wong et al. 2011; Zheng et al. 2016; Zhang et al. 2019).

4.2 Used data set

In order to analyze the effectiveness of our proposed method CGFL, we considered two different program suites for experimentation. The first program suite is *Siemens suite* which comprises of total seven programs. The Siemens suite programs are extracted from *SIR repository* (SIR 2005). The second program suite is Defects4J (2014). We took four programs from *Defects4j* repository into consideration. The test suites and buggy versions are already present in these benchmark suites.

Table 6 presents a tabular representation of program characteristics. Columns 3–6 indicate the total number of lines (LOC) present in the program, total executable lines of code present, number of faulty versions available, and number of test cases present in the test suite of that program.

Siemens suite has been widely used in the past as a benchmark for evaluating the effectiveness of various fault localization techniques (Wong et al. 2008; Jones et al. 2002; Wong et al. 2013). *Print_Tokens* and *Print_Tokens2* programs are prominent lexical analyzers. Siemens programs *Schedule2* and *Schedule* perform the task of priority scheduling. The *Replace* program is popularly used for pattern matching and substitution. *Tot_info* finds its application in computing several statistics of input data. *Tcas* stands for Traffic Collision Avoidance system and these programs are designed for minimizing the chances of mid-air collision between aircrafts.

The remaining four different Java programs have been taken from Defects4J (2014) repository. *Lang* is a Java utility class that helps to design and model the Java language. *Math* is a lightweight and self-contained library of mathematics and statistics components. *Mockito* is used to test the unit Java classes. *Time* (aka Joda-Time) is the de facto standard time and date library for Java programs.

The Siemens suite consists of a total of 132 faulty versions from seven different programs. However, we have considered only 116 faulty versions and omitted 16 versions, due to one or more of the following scenarios:

- A modification was made in a non-executable patch of the program.
- No semantic difference exists between the original and faulty programs except the header files.
- A test case execution resulted in a segmentation fault.
- None of the test cases failed for the version.

Now, for determining if a test case is successful or it has failed, the following steps were implemented.

1. All the test cases present in the test suite were executed with the faultless program(i.e. original program) and the output generated for each test case was stored in separate files.
2. The same test cases were executed using a faulty version of the program and the generated output was saved in a similar fashion.
3. Finally, a comparison was made to find out if the outputs generated for the fault-free and buggy version of the program are exactly the same or not. If the outputs were exactly identical then it is considered that the test case is successful; else, the test case is deemed to be failed for that particular faulty version.

4.3 Evaluation metric

We make use of four different metrics for analyzing the effectiveness of our proposed CGFL method. The following are the metrics:

EXAM_Score: This metric is popularly used for determining the effectiveness of a fault localization technique (Dutta

et al. 2021; Renieris and Reiss 2003). Equation (1) presents the formula for computing EXAM_Score.

$$EXAM\_Score = \frac{|S_{examined}|}{|S_{total}|} * 100 \quad (1)$$

where $|S_{examined}|$ shows the number of statements examined to localize the fault and $|S_{total}|$ denotes the total statements present in the program. Let us assume, on a program P, we used two fault localization techniques, say, T_a and T_b, and obtained EXAM_Scores are ES_a and ES_b respectively. If ES_a is less than ES_b, then technique T_a is more effective than T_b.

Top-N%: This metric shows the percentage of faulty versions which are correctly localized by examining at most N% of executable program statements (Li et al. 2019). For this study, we consider the N values as 1 and 5. It helps to discover the FL technique that localizes more faulty versions by analyzing up to 1% or 5% of program statements.

Relative Improvement (RImp): It shows the comparative improvement obtained using CGFL technology over existing FL methods. It is calculated using Eq. (2).

$$RImp_{a,b} = \frac{\#\ statements\ examined\ by\ Technique_a}{\#\ statements\ examined\ by\ Technique_b} * 100 \quad (2)$$

In this equation, Technique_a represents our proposed CGFL method and Technique_b can be any of the existing FL technique such as DStar (Wong et al. 2013), Tarantula (Jones and Harrold 2005) etc. with which proposed CGFL is compared. If the number of statements examined by Technique_a is lesser than Technique_b then the value of $\text{RImp}_{a,b}$ is lesser than 100% otherwise it is more than 100%.

Average Improvement: This metric indicates the average improvement realized on using an FL technique over another FL technique (Dutta et al. 2021). It is computed using Eq. (3).

$$IA_{a,b} = \frac{Avg.ES_b - Avg.ES_a}{Avg.ES_a} * 100 \quad (3)$$

Where, $IA_{a,b}$ exhibits the improvement achieved using T_a over T_b. $Avg.ES_a$ and $Avg.ES_b$ indicate the average *ES* obtained by T_a and T_b respectively. Hence, the lesser the average *EXAM_Score* better the technique is.

4.4 Results

In this section, we discuss the obtained comparative results for the proposed method CGFL with eight existing FL techniques. We consider four techniques from spectrum based family viz., Tarantula (Jones et al. 2002), DStar (Wong et al. 2013), Ochiai (Naish et al. 2011), and Crosstab (Wong et al. 2008) and remaining four methods from neural network models which includes BPNN (Wong and Qi 2009), DNN (Zheng et al. 2016), RBFNN (Wong et al. 2011) and CNN (Zhang et al. 2019). Tarantula (Jones et al. 2002) is a classic SBFL technique. Crosstab (Wong et al. 2008) is a statistics-based SBFL method that uses the Chi-square test to decide the dependency of test case result (success/failure) on the invocation of a specific program statement. Ochiai (Naish et al. 2011) is another prominent SBFL technique. DStar (Wong et al. 2013) is known as one of the state-of-the-art SBFL methods. Among the neural network-based FL models, BPNN (Wong and Qi 2009) is the most simple and easiest to implement. It was the first neural network-based model used for FL. RBFNN (Wong et al. 2011) deals with the problems of paralysis (Wasserman 1993) and local minima (Tan et al. 2013). DNN (Zheng et al. 2016) and CNN (Zhang et al. 2019) are the two most robust and effective models for fault localization.

Some of the FL techniques may assign identical suspicious scores to multiple program elements. This leads to two different types of effectiveness for that FL method viz., the *best* effectiveness and the *worst* effectiveness. When the fault localizer points to the buggy line first for examination, among all the statements holding the same suspiciousness values then it is termed as its *best* effectiveness. On the other hand, when the fault localizer points to the buggy line at last for examination, among all the statements holding the same suspiciousness values then it is termed as its *worst* effectiveness.

Figures 2, 3, 4, 5, 6, 7, 8, 9, 10 and 11 show the effectiveness comparison of CGFL and existing FL techniques. In these line graphs, the x-axis is used to denote the percentage of executable program statements examined and the y-axis is used to depict the percentage of buggy versions localized successfully. A point (x,y) in the graph represents that the y% of faulty programs are correctly localized by examining at most x% of statements of the corresponding program. We represent the *best* and the *worst* effectiveness with two different plots in these graphs.

Figure 2 show the comparative results of CGFL with Tarantula and DStar using the Siemens suite data set. It can be noted from the graph that by examining only 2% of the program statements CGFL(Best) localizes bugs in 44.66% of faulty versions. While, Tarantula(Best), Tarantula(Worst), DStar(Best), and DStar(Worst) localize bugs in only 19.41, 11.65, 26.21, and 12.62% of faulty versions by examining the same amount of program code. On average, CGFL(Best) is 56.15 and 50.31% more effective than Tarantula(Best) and DStar(Best) respectively. Similarly, CGFL(Worst) is respectively 38.15 and 34.68% more effective than Tarantula(Worst) and DStar(Worst).

Figure 3 pictorially represents the comparison of the effectiveness of CGFL, Ochiai, and Crosstab using the Siemens suite programs. By examining only 1% of program code, CGFL(Best) localizes bugs in 30.09% of faulty

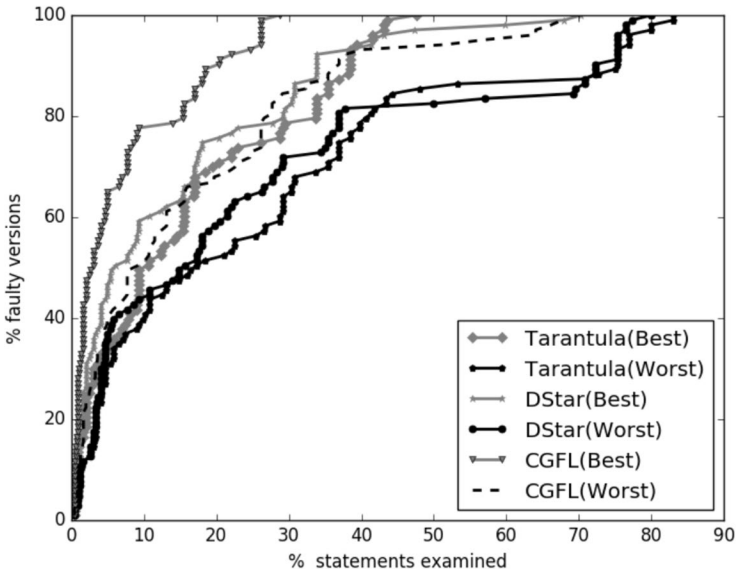

Fig. 2 Effectiveness comparison of CGFL with DStar and Tarantula for Siemens suite

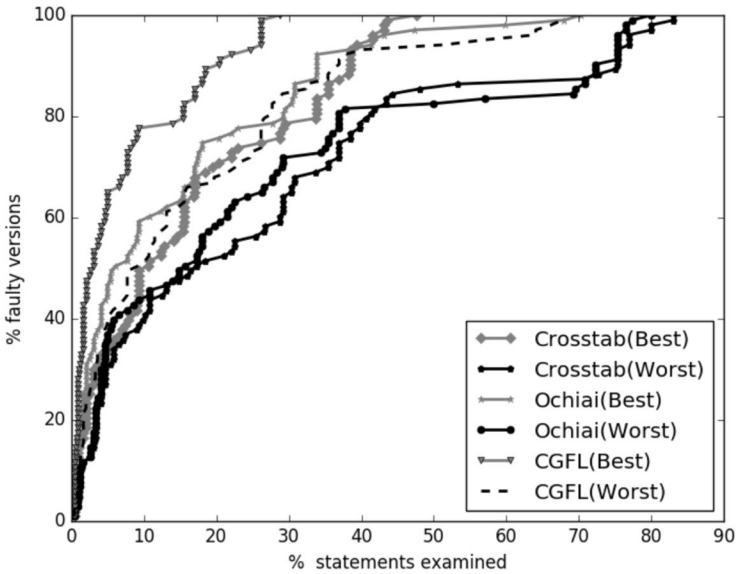

Fig. 3 Effectiveness comparison of CGFL with Crosstab and Ochiai for Siemens suite

versions whereas, Ochiai(Best) and Crosstab(Best) localize faults in only 18.44 and 3% of programs for the same percentage of code examination. In the worst case, CGFL(Best) is respectively 9.78 and 45.53% better than Ochiai(Best) and Crosstab(Best). Similarly, CGFL(Worst) is 7.69 and 18.46% more effective than Ochiai(Worst) and Crosstab(Worst) in the worst case. On average, CGFL is 31.86 and 49.16% more effective than Ochiai and Crosstab respectively.

Figure 4 shows the effectiveness comparison of CGFL with BPNN and RBFNN for the Siemens suite. We can observe from the line graph in Fig. 4 that by analyzing only 0.5% of program code, CGFL localizes bugs in 11.65% of faulty versions. On the other hand, by examining the same amount of code RBFNN(Best) and BPNN localize bugs in only 5.8 and 2.91% of faulty versions respectively. In the worst case, CGFL(Best) is respectively 44.27 and 20.55% better than RBFNN(Best) and BPNN. Likewise, CGFL(Worst) requires 21.54% less code examination than RBFNN(Worst) in the worst case. Whereas, with respect to BPNN, CGFL(Worst) checks 18.45% of more statements than BPNN in the worst-case scenario. On average, to localize bugs in all the faulty versions present in the Siemens suite, CGFL, RBFNN, and BPNN examine 11.24, 21.63, and 16.43% of program code respectively.

Figure 5 shows the effectiveness comparison between CGFL, DNN, and CNN using the Siemens suite data set. We can observe from this line graph that to localize bugs in 20% of faulty versions CGFL(Best) and CGFL(Worst) examines 0.81 and 1.5% of program code. On the other hand, to locate the buggy statements in the same amount of faulty

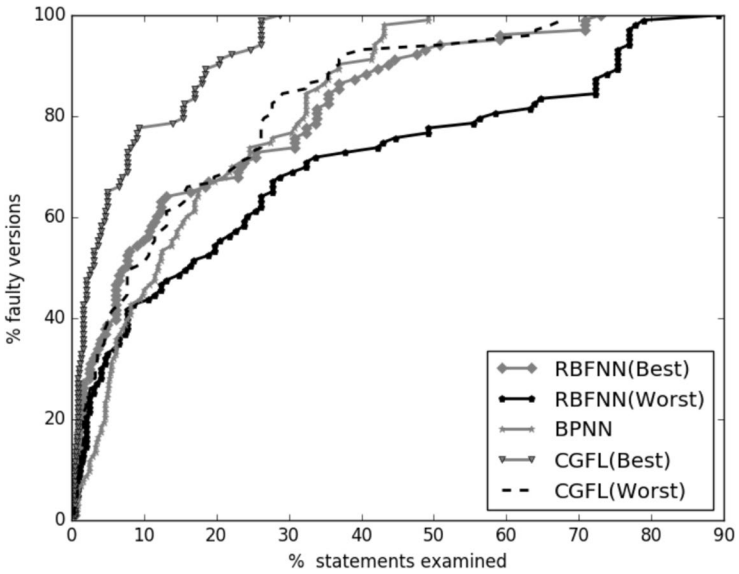

Fig. 4 Effectiveness comparison of CGFL with RBFNN and BPNN for Siemens suite

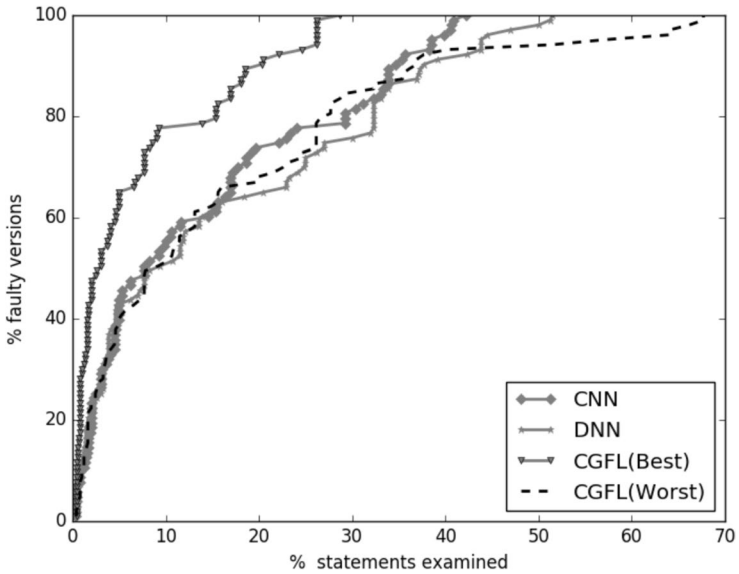

Fig. 5 Effectiveness comparison of CGFL with DNN and CNN for Siemens suite

statements, DNN and CNN respectively require to examine 1.6 and 2.04% of executable program statements. In the worst case, CGFL(Best) is 22.85 and 13.50% better than DNN and CNN. On average, CGFL is 39.28 and 23.36% better than DNN and CNN respectively.

Figures 6, 7, 8 and 9 show the comparison of results obtained using Defects4j data set for CGFL with the existing FL methods. Figure 6 shows the effectiveness comparison among Tarantula, DStar, and CGFL using a line graph. We can observe from this graph that by examining 35% of program code, CGFL(Best) locates bugs in almost all the faulty versions. However, only 96% of faulty versions are correctly localized by Tarantula(Best) on examining the same percentage of program code. On the other hand, while comparing with DStar, we observe that by checking only 2% of program code, CGFL(Best) locates bugs in 68.12% of faulty versions and DStar(Best) localizes only in 58% of buggy programs from the same set. In the worst case, CGFL(Best) requires 35.62 and 44.79% of less code examination than Tarantula(Best) and DStar(Best) respectively. On average, CGFL is 7.38 and 51.1% more effective than Tarantula and DStar respectively for the Defects4j suite.

Figure 7 represents the effectiveness of CGFL, Ochiai, and Crosstab. It can be noticed from Fig. 7 that CGFL has a better performance as compared to both Ochiai and Crosstab for most of the program points. By examining only 1% of program code, CGFL(Best) and CGFL(Worst) localize bugs in 55.56 and 37.20% of faulty versions. On the contrary, when the same patch of program code is inspected, Crosstab(Best), Crosstab(Worst), and Ochiai(Worst) locate

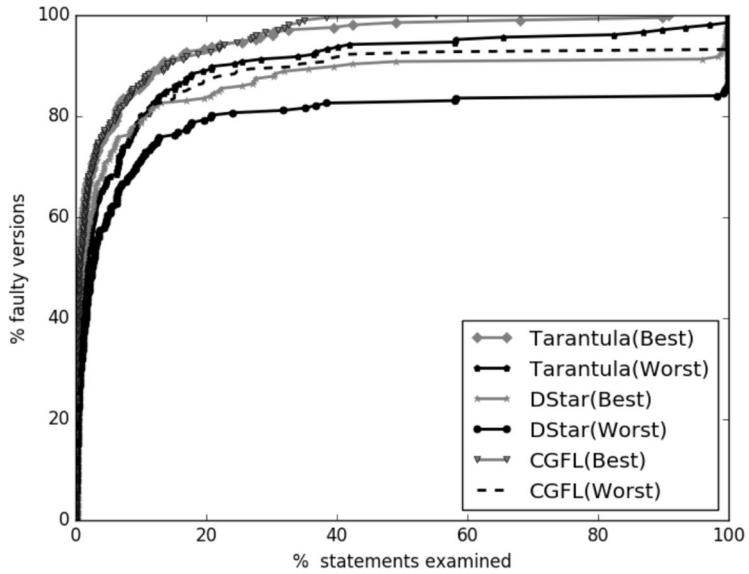

Fig. 6 Effectiveness comparison of CGFL with DStar and Tarantula for Defects4j suite

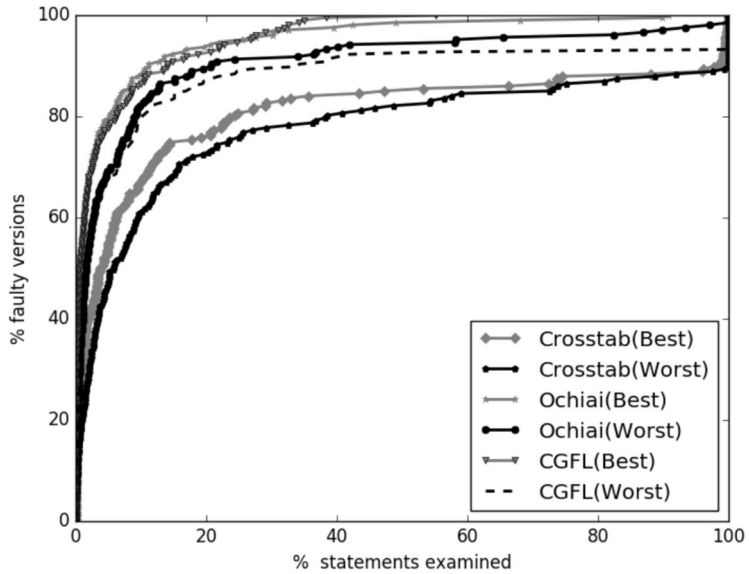

Fig. 7 Effectiveness comparison of CGFL with Crosstab and Ochiai for Defects4j suite

faults in 28.99, 28.29, and 36.71% of faulty versions respectively. In the worst case, CGFL(Best) requires 35.62 and 44.79% of less code examination than Ochiai(Best) and Crosstab(Best) respectively. On average, CGFL is 61% more effective than Crosstab. However, Ochiai requires 17% less code examination than CGFL for the Defects4j program suite.

Figure 8 presents the effectiveness comparison of CGFL, BPNN, and RBFNN with plotted EXAM_Score points in the range of 1 to 100%. On examining 5% of program code, CGFL(Best) and CGFL(worst) localize bugs in 77.78 and 67.63% of faulty versions. Whereas, BPNN and RBFNN(Worst) are able to locate bugs in only 12 and 71% of faulty versions. In the worst case, CGFL(Best) requires 44 and 44.79% of less code examination than BPNN and RBFNN(Best). On an average, EXAM_Score of BPNN, RBFNN(Best), RBFNN(Worst), CGFL(best) and CGFL(Worst) are 30.87, 8.54, 10.90, 4.40, and 12.03% for Defects4j data set respectively.

Figure 9 shows the comparison of CGFL, CNN, and DNN based on their effectiveness in locating program bugs. We can observe from the figure that for most of the EXAM_Score points, CGFL(Best) performs better than both DNN and CNN. However, there are some faulty versions present for which DNN localizes the faults more effectively compared with the remaining two techniques. For the Defects4j program suite, in the worst case, CNN and DNN require 15.83 and 35.62% of more code examination than CGFL(Best) respectively. On average, CNN is 24% more effective than CGFL.

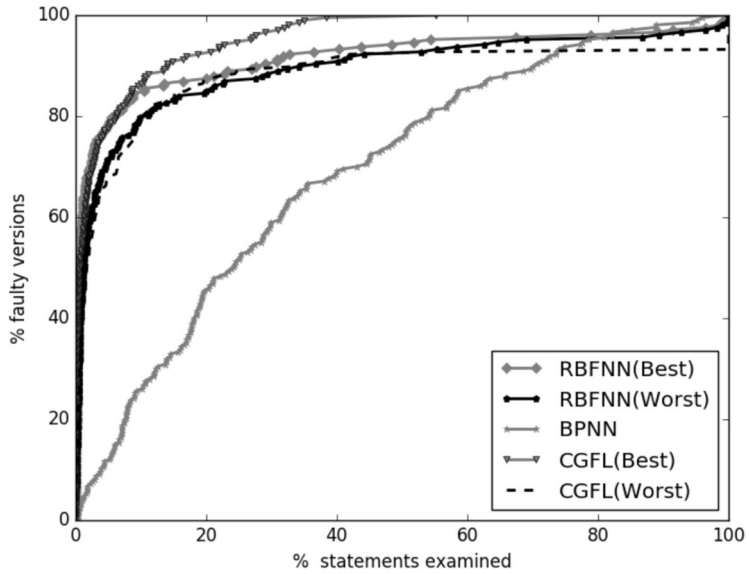

Fig. 8 Effectiveness comparison of CGFL with RBFNN and BPNN for Defects4j suite

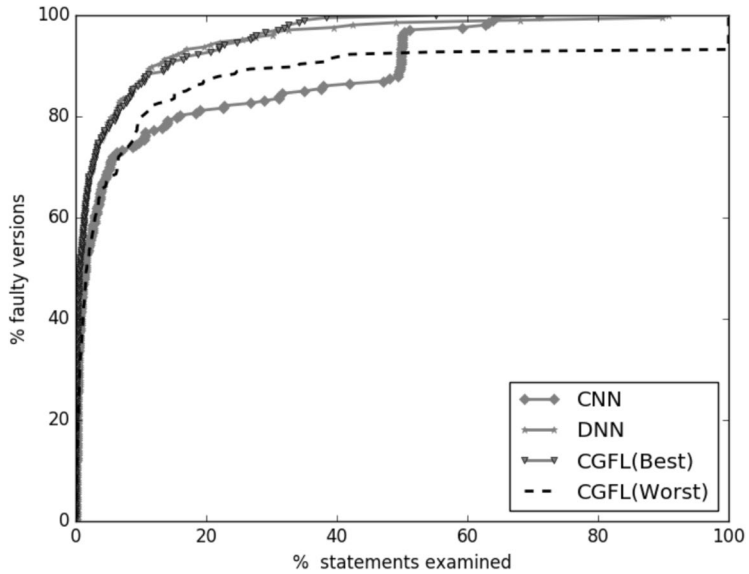

Fig. 9 Effectiveness comparison of CGFL with DNN and CNN for Defects4j suite

Figures 10 and 11 show the effectiveness comparison of CGFL and CPFL over the Siemens suite and Defects4j programs respectively. CPFL is our proposed conditional probability-based fault localization approach without using the grouping method. From both figures, we can observe that for almost every program point, CGFL(Best) performs more effectively than CPFL(Best). However, there are few program points present on which CGFL(Worst) is lesser effective than CPFL(worst). But the count of such program points is comparatively low.

It can be observed from the Fig. 10 that in the worst case CPFL(Best) requires 35.93% Exam_Score whereas CGFL(Best) examines only 28.68% of program code. On average, CPFL(Best), CPFL(Worst), CGFL(Best), and CGFL(Worst) require 8.40, 17.73, 6.74, and 15.74% of code examination for Siemens suite respectively. Using the grouping method, we have obtained an improvement of 14.05%, on average, over CPFL for the Siemens suite.

For the Defects4j program suite, we have obtained, on average, 4.97% of improvement over CPFL(Best) by using CGFL(Best). Also, for the complete data set, we require 3% less statements examination than the non-grouping method while localizing the faults with the CGFL method.

Table 7 presents the pairwise comparison between CGFL and other fault localization techniques viz., Tarantula (Jones et al. 2002), DStar (Wong et al. 2013), Crosstab (Wong et al. 2008), Ochiai (Naish et al. 2011), BPNN (Wong and Qi 2009), RBFNN (Wong et al. 2011), CNN (Zhang et al. 2019), DNN (Zheng et al. 2016) and our proposed CPFL. In the sub-column titles (Best v/s Best), (Worst v/s Worst), and

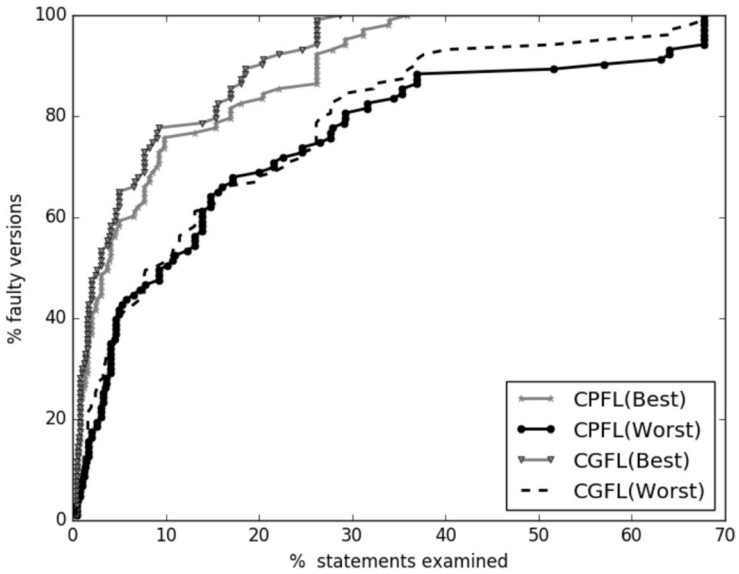

Fig. 10 Effectiveness comparison of CGFL and CPFL for Siemens suite

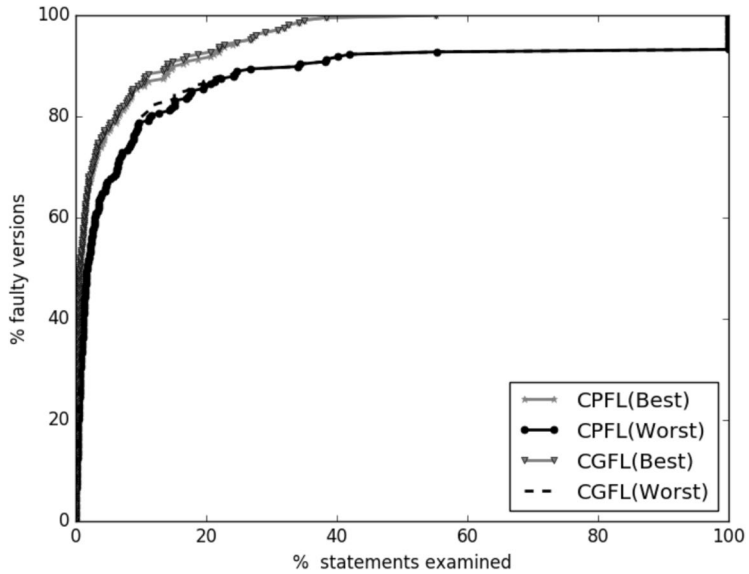

Fig. 11 Effectiveness comparison of CGFL and CPFL for Defects4j suite

(Worst v/s Best), the second tag (either Best or Worst) represents the effectiveness of the existing technique. The first tag stands for the effectiveness of our proposed CGFL method. For example, the sub-column (Worst v/s Best) contains the percentage of faulty versions for which the proposed CGFL technique's worst-case behavior is more effective than the respective existing technique's best-case results. Table 7 shows the percentage of buggy programs on which CGFL performs more effective, equally effective, and less effective than the existing fault localization techniques.

It is discovered from the table that CGFL(Best) is more effective in at least 58% of the faulty versions than the existing fault localization techniques' best-case effectiveness. Also, for less than 50% of buggy versions, the worst-case effectiveness of CGFL is less effective than the existing techniques' effectiveness in the worst case. Similarly, for Defects4j programs, CGFL(Best) is at least as effective or more effective than 50% of the faulty versions. Program on which the number of resultant tied statements (statements with the same suspiciousness score) are large with respect to an FL technique may lead to a bad performance by the CGFL method.

Table 8 presents the relative improvement(*RImp*) obtained using CGFL over the existing FL techniques as well as for the proposed CPFL method. It is calculated using Eq. (2). *RImp* value shows the percentage of statements examined by CGFL with respect to any other FL technique. It can be observed from the table that for almost all the Siemens suite programs, the obtained *RImp* value is lesser than 100%. Only for programs Schedule2 and Tcas, the effectiveness of

Table 7 Pairwise comparison of CGFL with other fault localization techniques

Technique	Effectiveness	Siemens			Defects4j		
		Best v/s Best	Worst v/s Worst	Worst v/s Best	Best v/s Best	Worst v/s Worst	Worst v/s Best
Tarantula	More effective	69.09	64.55	52.73	39.61	46.86	28.02
	Equally effective	15.45	13.64	4.55	15.94	2.42	4.35
	Less effective	15.45	21.82	42.73	44.44	50.72	67.63
Dstar	More effective	60.91	56.36	40.00	44.93	51.21	32.85
	Equally effective	20.00	17.27	5.45	12.56	3.38	4.83
	Less effective	19.09	26.36	54.55	42.51	45.41	62.32
Crosstab	More effective	76.36	75.45	54.55	60.87	63.29	50.72
	Equally effective	2.73	0.91	5.45	8.70	2.42	1.93
	Less effective	20.91	23.64	40.00	30.43	34.30	47.34
Ochiai	More effective	58.18	57.27	41.82	38.16	45.41	27.05
	Equally effective	20.00	17.27	3.64	15.94	2.90	4.35
	Less effective	21.82	25.45	54.55	45.89	51.69	68.60
BPNN	More effective	71.82	50.91	50.91	88.41	80.19	80.19
	Equally effective	3.64	2.73	2.73	1.45	0.97	0.97
	Less effective	24.55	46.36	46.36	10.14	18.84	18.84
RBFNN	More effective	64.55	60.91	42.73	45.41	47.83	33.33
	Equally effective	8.18	8.18	9.09	12.08	1.45	2.90
	Less effective	27.27	30.91	48.18	42.51	50.72	63.77
DNN	More effective	68.18	43.64	43.64	39.13	27.54	27.54
	Equally effective	5.45	4.55	4.55	14.98	3.86	3.86
	Less effective	26.36	51.82	51.82	45.89	68.60	68.60
CNN	More effective	63.64	45.45	45.45	55.07	39.61	39.61
	Equally effective	11.82	3.64	3.64	6.28	2.90	2.90
	Less effective	24.55	50.91	50.91	38.65	57.49	57.49
CPFL	More effective	30.91	30.00	13.64	43.48	49.76	30.92
	Equally effective	49.09	48.18	13.64	14.49	1.93	3.86
	Less effective	20.00	21.82	72.73	42.03	48.31	65.22

Table 8 Relative improvement for CGFL w.r.t. existing FL techniques

	Tarantula (Best)	Tarantula (Worst)	DStar (Best)	DStar (Worst)	Crosstab (Best)	Crosstab (Worst)	Ochiai (Best)	Ochiai (Worst)	BPNN	RBFNN (Best)	RBFNN (Worst)	DNN	CNN
Print_Token	21.15	31.67	9.40	15.20	9.36	15.14	51.16	64.41	29.70	11.46	18.91	28.17	32.49
Print_Token2	13.27	19.74	5.49	8.54	15.82	22.73	21.88	31.03	22.12	51.85	72.58	25.89	17.16
Schedule	17.86	33.82	4.72	10.13	34.48	56.10	52.63	74.19	22.92	6.90	14.74	9.24	14.29
Schedule2	59.59	63.65	69.52	72.80	47.46	51.77	81.70	85.01	119.35	89.30	88.37	85.94	82.58
Replace	40.48	67.38	57.92	87.94	32.95	61.57	57.72	98.38	40.13	40.57	75.00	75.02	90.90
Tcas	64.55	72.85	70.97	76.61	56.21	68.83	77.55	79.55	95.00	44.85	61.41	91.96	124.00
Tot_Info	26.73	60.49	54.37	96.89	26.42	59.76	27.12	60.92	74.84	26.48	63.28	72.34	73.51
Lang	67.33	88.15	19.62	32.76	16.00	34.99	78.04	96.06	14.62	71.26	127.07	124.94	53.18
Math	101.04	135.25	22.51	40.26	15.62	31.12	111.24	141.75	15.85	130.78	156.81	181.89	65.96
Mockito	83.14	142.76	74.66	91.93	37.11	111.02	87.06	146.38	64.68	36.29	110.73	179.62	103.14
Time	63.82	106.36	126.55	110.07	84.28	129.17	66.33	109.27	16.93	71.40	115.38	107.08	150.28

CGFL is lesser than BPNN and CNN respectively. For a few programs, CGFL requires more amount of code examination than existing techniques. The reason behind this is probably the assignment of the same suspiciousness scores to multiple statements. On average, there is a reduction of 37.22, 47.32, 53.73, 22.75, 53.08, 32.48, 10.72, and 26.59% against

Tarantula, DStar, Crosstab, Ochiai, BPNN, RBFNN, DNN, and CNN respectively.

Tables 9 and 10 show the percentage of faulty versions localized by examining only Top-1% and Top-5% of program code respectively. The fault localization technique is considered to be better if more buggy versions are localized by just examining a smaller portion of the code. It can be observed from both Table 9 and 10 that there are different programs for which various FL techniques are unable to localize the bugs in any of the faulty versions by examining only Top-1% of program code. CGFL(Best) is able to locate bugs in a minimum of 37.50% of faulty versions for all the programs except Tcas and Schedule2. On the other hand, by examining 5% of program code, CGFL(Best) is able to locate bugs in at least 12.50% of faulty versions for all the considered sets of programs. Moreover, for the Schedule program, CGFL(Best) is able to locate bugs in all the faulty versions by examining only Top-5% of the code.

Table 11 presents a run time analysis of different fault localization techniques along with our proposed CGFL method. Considered FL techniques are Tarantula, DStar, Crosstab, Ochiai, BPNN, RBFNN, DNN, and CNN. In this table, 'min' 'sec', and 'ms' represent minute, second, and millisecond, respectively. We have not considered the test case result generation time and statement coverage computation time because these two times were similar for all the methods. We can observe from the table that SBFL methods (Tarantula, DStar, Crosstab, Ochiai, and CGFL) are in the order of seconds, whereas machine learning-based FL methods (CNN, DNN, BPNN, and RBFNN) require more time which is in order of minutes. NN-based methods hold the drawback of consuming extra time due to time spent on training of the model for each faulty program. Based on the observation of results as depicted in Table 11 it can be concluded that the CGFL method takes comparable time with SBFL techniques and shows greater efficiency than the NN-based methods.

4.5 Threats to the validity

In this section, we discuss some important threats to the validity of our proposed approach.

- Construct Validity
  - There can be a scenario where all the test cases may fail or all the test cases may pass, in that situation, CGFL would fail to localize the faulty statement. Therefore, the effectiveness of CGFL has a dependence on both failed and passed test cases.
  - While computing the suspiciousness score of statements, we have considered the same contribution of each test case. In reality, individual test cases have

Table 9 Percentage of faulty versions are successfully localized by examining Top-1% of executable program code

	Tarantula (Best)	Tarantula (Worst)	Dstar (Best)	Dstar (Worst)	Crosstab (Best)	Crosstab (worst)	Ochiai (Best)	Ochiai (Worst)	BPNN	RBFNN (Best)	RBFNN (Worst)	DNN	CNN	CGFL (Best)	CGFL (Worst)
Print_Token	40.00	20.00	40.00	0.00	0.00	0.00	60.00	20.00	0.00	20.00	20.00	20.00	40.00	60.00	20.00
Print_Token2	37.50	12.50	12.50	0.00	25.00	0.00	37.50	12.50	0.00	12.50	0.00	37.50	37.50	37.50	12.50
Schedule	0.00	0.00	20.00	0.00	0.00	0.00	20.00	0.00	0.00	0.00	0.00	0.00	0.00	40.00	0.00
Schedule2	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Replace	24.14	13.79	31.03	13.79	0.00	0.00	24.14	13.79	10.34	31.03	17.24	17.24	13.79	44.83	20.69
Tcas	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Tot_Info	10.53	0.00	21.05	0.00	5.26	0.00	15.79	0.00	15.79	15.79	0.00	15.79	5.26	42.11	0.00
Lang	47.17	26.42	41.51	24.53	28.30	20.75	45.28	28.30	5.66	56.60	37.74	47.17	39.62	47.17	28.30
Math	62.77	37.23	47.87	25.53	24.47	11.70	61.70	36.17	4.26	67.02	44.68	62.77	47.87	58.51	32.98
Mockito	51.35	40.54	48.65	40.54	24.32	18.92	51.35	43.24	0.00	48.65	45.95	51.35	37.84	48.65	40.54
Time	69.57	65.22	78.26	73.91	56.52	56.52	73.91	69.57	4.35	60.87	56.52	69.57	69.57	69.57	65.22

Table 10 Percentage of faulty versions are successfully localized by examining Top-5% of executable program code

	Tarantula (Best)	Tarantula (Worst)	Dstar (Best)	Dstar (Worst)	Crosstab (Best)	Crosstab (worst)	Ochiai (Best)	Ochiai (Worst)	BPNN	RBFNN (Best)	RBFNN (Worst)	DNN	CNN	CGFL (Best)	CGFL (Worst)
Print_Token	40.00	40.00	60.00	60.00	60.00	20.00	80.00	80.00	0.00	80.00	80.00	60.00	60.00	80.00	80.00
Print_Token2	62.50	62.50	37.50	37.50	62.50	62.50	62.50	62.50	25.00	75.00	75.00	50.00	50.00	87.50	87.50
Schedule	40.00	40.00	60.00	60.00	60.00	60.00	80.00	60.00	40.00	60.00	20.00	60.00	60.00	100.00	80.00
Schedule2	0.00	0.00	12.50	0.00	0.00	0.00	12.50	12.50	12.50	0.00	0.00	0.00	12.50	12.50	0.00
Replace	48.28	44.83	65.52	62.07	51.72	44.83	65.52	58.62	27.59	62.07	62.07	65.52	68.97	82.76	65.52
Tcas	16.67	13.89	22.22	13.89	5.56	0.00	19.44	16.67	8.33	2.78	0.00	19.44	19.44	33.33	5.56
Tot_Info	26.32	21.05	57.89	31.58	21.05	5.26	31.58	26.32	47.37	42.11	26.32	42.11	36.84	73.68	26.32
Lang	71.70	54.72	69.81	49.06	56.60	43.40	75.47	56.60	15.09	77.36	67.92	75.47	69.81	79.25	58.49
Math	80.85	72.34	69.15	59.57	47.87	41.49	82.98	73.40	7.45	84.04	75.53	81.91	68.09	78.72	71.28
Mockito	67.57	62.16	67.57	62.16	51.35	43.24	70.27	64.86	10.81	62.16	62.16	70.27	62.16	64.86	59.46
Time	86.96	86.96	86.96	86.96	82.61	78.26	86.96	86.96	21.74	78.26	78.26	86.96	82.61	86.96	86.96

different contributions to deciding the suspiciousness score.

- External Validity
  - Used operating systems, coverage measurement tools, compilers, and hardware platforms also have an impact on the generated results. But, to eliminate the discrepancies, we have generated results for all the existing techniques along with our proposed method on the same platform.
- Internal Validity
  - We assess the performance of our proposed CGFL technique on limited empirical data. There is a possibility that the technique may not work in a similar fashion with a different set of programs. But to lower the possibility of such risk, programs with varying features and from different domains are taken into consideration.

5 Comparison with related work

In this section, we compare the proposed CGFL technique with a few related works.

Different slicing based techniques were reported in the literature (Weiser 1984; Korel and Laski 1988) to handle the problem of fault localization. However, these techniques are not effective enough. A number of times, slicing-based approaches return the whole program as a slice and sometimes a null set of statements too. Furthermore, the statements do not receive any rank by using these techniques. While our technique CGFL generates a ranked list of executable statements. The ranks are assigned based on the suspiciousness of program statements to contain a bug.

Jones and Harrold (2005) developed an automated fault localization technique called Tarantula. In this method, the test execution result and statement coverage information are utilized for computing the suspicious scores of statements based on their probability of containing a bug. Their experimental result showed that Tarantula is more effective than cause-transition (Cleve and Zeller 2005), nearest neighbour (Renieris and Reiss 2003), set union (Jones and Harrold 2005), and set intersection (Jones and Harrold 2005) based fault localization methods. Our empirical evaluation shows that CGFL performs 32.81% more effectively in locating the bugs as compared to Tarantula.

Renieris and Reiss (2003) discussed the nearest neighbor technique for effective fault localization. They have defined two distance metrics viz., binary distancing and permutation distancing to calculate the similarity between a failed and a passed test case. They select an arbitrary failed test case and compute distance with every passed test case. Subsequently, they select the passed test case with minimum

Table 11 Time analysis of different fault localization techniques

Program	Tarantula	Dstar	Crosstab	Ochiai	BPNN	RBFNN	DNN	CNN	CGFL
Print_Token	31 ms	30 ms	47 ms	35 ms	1 min 22 sec	6 min 20 sec	3 min 40 sec	5 min 28 sec	31 ms
Pint_ Token2	32 ms	34 ms	50 ms	33 ms	1 min 58 ms	4 min 14 sec	4 min 08 sec	6 min 04 sec	33 ms
Replace	58 ms	56 ms	1 sec 22 ms	52 ms	3 min 21 sec	11 min 02 sec	8 min 44 sec	12 min 08 sec	58 ms
Tcas	6 ms	7 ms	12 ms	8 ms	26 sec 02 ms	52 sec 12 ms	39 sec 41 ms	44 sec 12 ms	7 ms
Tot_info	6 ms	6 ms	10 ms	7 ms	1 min 19 ms	3 min 18 sec	2 min 52 sec	3 min 33 sec	7 ms
Schedule	20 ms	22 ms	38 ms	18 ms	1 min 28 sec	2 min 05 sec	2 min 08 sec	2 min 40 sec	23 ms
Schedule2	13 ms	14 ms	24 ms	14 ms	2 min 12 sec	4 min 27 sec	3 min 06 sec	4 min 10 sec	13 ms
Lang	33 sec 18 ms	31 sec 02 ms	56 sec 16 ms	36 sec 20 ms	58 sec 02 ms	2 min 12 sec	2 min 10 sec	3 min 02 sec	30 sec 12 ms
Math	1 sec 11 ms	1 sec 22 ms	2 sec 26 ms	1 sec 54 ms	2 min 48 sec	4 min 51 sec	5 min 10 sec	8 min 48 sec	1 sec 48 ms
Mockhito	2 sec 28 ms	1 sec 38 ms	3 sec 47 ms	2 sec 18 ms	3 min 12 sec	6 min 52 sec	7 min 17 sec	9 min 10 sec	2 sec 12 ms
Time	11 sec 59 ms	12 sec 18 ms	16 sec 2 ms	14 sec 12 ms	20 min 16 sec	85 min 19 sec	96 min 40 sec	98 min 12 sec	10 sec 58 ms

distance and remove all the statements executed by that test case from the set of statements executed by the failed test case. The major drawback of this technique is the sensitivity towards the selected test cases. If the correct pair of test cases are not selected then it would either return a null set or an irrelevant set of statements. It mainly occurs when the buggy statement is executed by both the pass and the failed test cases. Whereas, CGFL always return a ranked list of executable statements based on their suspiciousness of containing a bug.

The conditional probability model adopted by Yang et al. (2019) did not incorporate the impact of an important probability statistic ψ_{fc} (probability of program element to be faulty if it has been executed by the test case) while computing the suspicious score of an executable program statement. Moreover, Yang et al. (2019) scaled down the influence of probability parameter ψ_{pu} leading to negligence of scenarios where the bug is present in conditional statements of the program. Also, the factor chosen for scaling down is the inverse of the total number of test cases while the cardinality of test cases present in the program's test suite holds no relation to the fault's location in the program. Therefore, different numbers of test cases considered may lead to significant changes in suspicious scores calculated for the same program. CGFL is independent of the cardinality of test cases therefore, it behaves similarly for different test suites of the same program. The proposed method assigns equal weightage to all the probability statistics involved thus considering different statement invocation and execution result scenarios equally. Also, our empirical evaluation shows CGFL is 28.56% more effective than Yang et al. (2019) proposed approach.

Wong et al. (2008) developed a statistical analysis approach for fault localization. They used the chi-square test to determine the association between the invocation of a statement and the execution outcome of a test case. Their proposed Crosstab approach is not applicable to different-sized programs. Whereas, the CGFL technique is easily applicable to any sized program. Also, we have compared the effectiveness of CGFL with Crosstab and found it to be 58.90% more effective than Crosstab. Along with Crosstab, we have also compared the effectiveness of CGFL with three other SBFL techniques viz., DStar (Wong et al. 2013), Tarantula (Jones et al. 2002), and Ochiai (Naish et al. 2011). Our empirical evaluation shows that CGFL requires to examine 26.84, 32.81, and 28.11% of less code than DStar, Tarantula, and Ochiai respectively.

In the past two decades, different mutation-based techniques have been used for fault localization. Some of the prominent mutation-based FL techniques are MUSE (Moon et al. 2014) and Metallaxis (Papadakis and Le Traon 2015). Mutation-based FL techniques are effective but they are not easily applicable for large-sized programs. Since these techniques suffer from the problem of scalability. They require huge computational power to generate a large number of mutants and thereafter to generate the statement coverage data and test execution information for each of the generated mutants. Also, it is challenging to generate all possible mutants of a program. On the other hand, our proposed CGFL technique is lightweight and straightforward. It does not require any additional investment in terms of space and time.

Wong et al. proposed different neural network-based FL techniques such as BPNN (Wong and Qi 2009) and RBFNN (Wong et al. 2011). Zheng et al. (2016) used a deep neural network for the same. Later, Zhang et al. (2017) extended the DNN model (Zheng et al. 2016) for FL by appending

contextual information into it. Dutta et al. (2019) proposed a hierarchical approach for FL where they first localized the bug at the function level and subsequently at the statement level. Though NN-based techniques are effective they suffer from the problems of non-deterministic feedback loops and parameter estimations. Also, the recent deep learning-based FL techniques such as DeepFL (Li et al. 2019) and DeepRL4FL (Li et al. 2021) require several complex features (e.g. spectrum-based suspiciousness and complexity-based fault proneness) leading to a higher overhead for obtaining the required information. These models also require huge training time. On the other hand, CGFL is a light-weight and simple conditional probability-based mathematical approach with minimal time requirement.

6 Conclusion

Software debugging is a tedious and time-consuming activity. Any improvement to it leads reduction in total software development cost. In this work, we proposed a fault localization technique that helps to mitigate the debugging cost up to a large extent. Our proposed CGFL technique is based on conditional probability-based statistics which captures the association between the execution of the statement and the test case outcome. We further appended a test case execution-based grouping approach to mitigate the ties among the statements along with more effective rank list generation. Our empirical evaluation of two popular data sets shows that on average, CGFL requires 24.56% less code examination than existing fault localization techniques.

In the future, we intend to provide different weights to all the test cases as different test cases have different contributions in computing the suspicious score of a statement. The contribution value is computed using statement coverage information and the execution result(pass/fail) of the test case. In this way, the fault localizer becomes more targeted toward program faults. We also plan to incorporate statement frequency information in our CGFL technique. Statement frequency shows the number of times a statement is executed by any test case.

Funding This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

Declarations

Conflict of interest The authors have no Conflict of interest.

Human and animal rights This article does not contain any studies involving human and animal participants performed by any of the authors.

Informed consent Informed consent was obtained from all individual participants included in the study.

References

Agrawal H, Horgan JR, London S, Wong WE (1995) Fault localization using execution slices and dataflow tests. In: Proceedings of sixth international symposium on software reliability engineering, ISSRE'95. IEEE, pp 143–151

Ascari LC, Araki LY, Pozo AR, Vergilio SR (2009) Exploring machine learning techniques for fault localization. In: 2009 10th Latin American test workshop. IEEE, pp 1–6

Briand LC, Labiche Y, Liu X (2007) Using machine learning to support debugging with tarantula. In: The 18th IEEE international symposium on software reliability (ISSRE'07). IEEE, pp 137–146

Cleve H, Zeller A (2005) Locating causes of program failures. In: Proceedings of the 27th international conference on software engineering, pp 342—351

Debroy V, Wong WE, Xu X, Choi B (2010) A grouping-based strategy to improve the effectiveness of fault localization techniques. In: 2010 10th international conference on quality software. IEEE, pp 13–22

Defects4J (2014) https://github.com/rjust/defects4j

Defects4J (2016) http://fault-localization.cs.washington.edu/data/

Dekking FM, Kraaikamp C, Lopuhaä HP, Meester LE (2006) A modern introduction to probability and statistics: understanding why and how. Springer Science & Business Media, Berlin

Dutta A, Manral R, Mitra P, Mall R (2019) Hierarchically localizing software faults using DNN. IEEE Trans Reliab 69(4):1267–1292

Dutta A, Srivastava SS, Godboley S, Mohapatra D (2021) Combi-FL: neural network and SBFL based fault localization using mutation analysis. J Comput Lang 66:101064

Dutta A, Godboley S (2021) MSFL: a model for fault localization using mutation-spectra technique. In: International conference on lean and agile software development. Springer, pp 156–173

Dutta A, Kunal K, Srivastava SS, Shankar S, Mall R (2021) FTFL: a Fisher's test-based approach for fault localization. Innov Syst Softw Eng 17(4):381–405

Dutta A, Pant N, Mitra P, Mall R (2019) Effective fault localization using an ensemble classifier. In: 2019 international conference on quality, reliability, risk, maintenance, and safety engineering (QR2MSE). IEEE, pp 847–855

gcov (2002) http://man7.org/linux/man-pages/man1/gcov-tool.1.html

Harrold MJ, Rothermel G, Wu R, Yi L (1998) An empirical investigation of program spectra. In: Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering, pp 83–90

Jones JA, Harrold MJ (2005) Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM international conference on automated software engineering, pp 273–282

Jones JA, Harrold MJ, Stasko J (2002) Visualization of test information to assist fault localization. In: Proceedings of the 24th international conference on software engineering, ICSE 2002. IEEE, pp 467–477

Korel B, Laski J (1988) Dynamic program slicing. Inf Process Lett 29(3):155–163

Li X, Li W, Zhang Y, Zhang L (2019) DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In: Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis, pp 169–180

Li Y, Wang S, Nguyen T (2021) Fault localization with code coverage representation learning. In: 2021 IEEE/ACM 43rd international conference on software engineering (ICSE). IEEE, pp 661–673
Lou Y, Zhu Q, Dong J, Li X, Sun Z, Hao D, Zhang L, Zhang L (2021) Boosting coverage-based fault localization via graph-based representation learning. In: Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering, pp 664–676
Mall R (2018) Fundamentals of software engineering. PHI Learning Pvt. Ltd., Delhi
Moon S, Kim Y, Kim M, Yoo S (2014) Ask the mutants: mutating faulty programs for fault localization. In: 2014 IEEE seventh international conference on software testing, verification and validation. IEEE, pp 153–162
Naish L, Lee HJ, Ramamohanarao K (2011) A model for spectra-based software diagnosis. ACM Trans Softw Eng Methodol TOSEM 20(3):1–32
Papadakis M, Le Traon Y (2015) Metallaxis-FL: mutation-based fault localization. Softw Test Verif Reliab 25(5–7):605–628
Renieris M, Reiss SP (2003) Fault localization with nearest neighbor queries. In: 18th IEEE international conference on automated software engineering, 2003. Proceedings, pp 30–39
SIR (2005) http://sir.unl.edu/portal/index.php
Tan PN, Steinbach M, Kumar V (2013) Data mining cluster analysis: basic concepts and algorithms. Introduction to data mining, vol 487. Pearson Education India, Bengaluru, p 533
Wasserman PD (1993) Advanced methods in neural computing. John Wiley & Sons Inc, Hoboken
Weiser M (1984) Program slicing. IEEE Trans Softw Eng 4:352–357
Wong WE, Qi Y (2009) BP neural network-based effective fault localization. Int J Softw Eng Knowl Eng 19(04):573–597
Wong WE, Debroy V, Choi B (2010) A family of code coverage-based heuristics for effective fault localization. J Syst Softw 83(2):188–208
Wong WE, Debroy V, Golden R, Xu X, Thuraisingham B (2011) Effective software fault localization using an RBF neural network. IEEE Trans Reliab 61(1):149–169
Wong WE, Debroy V, Gao R, Li Y (2013) The DStar method for effective software fault localization. IEEE Trans Reliab 63(1):290–308
Wong WE, Gao R, Li Y, Abreu R, Wotawa F (2016) A survey on software fault localization. IEEE Trans Softw Eng 42(8):707–740
Wong E, Wei T, Qi Y, Zhao L (2008) A crosstab-based statistical method for effective fault localization. In: 2008 1st international conference on software testing, verification, and validation. IEEE, pp 42–51
Xiao X, Pan Y, Zhang B, Hu G, Li Q, Lu R (2021) ALBFL: a novel neural ranking model for software fault localization via combining static and dynamic features. Inf Softw Technol 139:106653
Yang Y, Deng F, Yan Y, Gao F (2019) A fault localization method based upon conditional probability. In: Proceedings of the 19th international conference on software quality, reliability and security companion (QRS-C). IEEE
Zhang Z, Lei Y, Tan Q, Mao X, Zeng P, Chang X (2017) Deep learning-based fault localization with contextual information. IEICE Trans Inf Syst 100(12):3027–3031
Zhang Z, Lei Y, Mao X, Li P (2019) CNN-FL: an effective approach for localizing faults using convolutional neural networks. In: 2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER). IEEE, pp 445–455
Zheng W, Hu D, Wang J (2016) Fault localization analysis based on deep neural network. Math Probl Eng 2016:1820454