**Empirical Analysis on Fault Localization technique as a hybrid approach**

Nadhratunnaim Nasarudin

Department of Computer Science University of York

This dissertation is submitted for the degree of

*Doctor of Philosophy*

January 2022

# Abstract

Debugging is a one of activity in the development process and has been used extensively by developers to localize faults in enhancing the quality and performance in general (Elsaka, 2017) . This critical area has contributed a significant amount of study in improving various Fault Localization technique in assisting developer to allocate fault. Nevertheless, identifying fault using stand-alone fault localization technique are not always effective, thus by combining different technique of Fault Localization that represent distinct analysis might overcome this issue. A few research has shown that combining more than one approaches can assist developer better, since aspects from different sources are included in the fault localization process. To suggest the best combination technique that can perform fault localization effective and efficiently, I have combined three fault localization technique which are Information Retrieval (IR) Based Technique, Spectrum Based Fault Localization (SBFL) Technique, and Text Based Search Engine. The combination of dynamic and static fault localization analysis can assist developer in better fault localization as it includes different sources of analysis, and the Text Based Search Engine approach might cover the fault localization area that IR and SBFL technique might not. To execute the experiment, I have used a real-world java program, Defects4j by (Just, et al., 2014) where about 395 bug reports are selected with their appropriate assignees. The experimental results demonstrate the efficiency of proposed combined technique in comparison with SBFL and IR technique alone in assisting developers to fault localization, while Text Search Engine as a new approach in fault localization also contribute to the overall accuracy performance.

Table of Contents

[Abstract 1](#_Toc90317248)

[1.0 Introduction 2](#_Toc90317249)

[1.1 Objective and Motivation 2](#_Toc90317250)

[1.2 Hypothesis, Problem Statement and Research Question 4](#_Toc90317251)

[1.3 Summary and structure of the thesis 9](#_Toc90317252)

[2.0 Background and related work 10](#_Toc90317253)

[2.1 Fundamental definitions and terminology: error, fault, bug, test 10](#_Toc90317254)

[2.2 Spectrum Based Fault Localization (SBFL) 11](#_Toc90317255)

[2.3 Information retrieval- based (IR) 13](#_Toc90317256)

[2.4.1 Text retrieval technique 15](#_Toc90317257)

[2.4.2 Stack trace technique 16](#_Toc90317258)

[2.4.3 History-based technique 17](#_Toc90317259)

[2.4 Others Fault Localization Technique 18](#_Toc90317260)

[2.4.1 Program slicing 18](#_Toc90317261)

[2.4.2 Mutation Based Fault Localization (MBFL) 19](#_Toc90317262)

[2.4.3 Predicate switching 20](#_Toc90317263)

[2.4.4 Delta Debugging 21](#_Toc90317264)

[2.5 Hybrid Techniques and related works 22](#_Toc90317265)

[2.6 Text Search Engine 24](#_Toc90317266)

[3.0 Experiment methodology 26](#_Toc90317267)

[3.1 Experimental set-up 26](#_Toc90317268)

[3.2 Experimental subjects 27](#_Toc90317269)

[3.3 Studied techniques and their implementations 29](#_Toc90317270)

[3.3.1 SBFL Experiment 29](#_Toc90317271)

[3.3 IR technique Experiment 31](#_Toc90317272)

[3.3.3 Text Search Engine Experiment 37](#_Toc90317273)

[3.3.4 Experiment procedure 38](#_Toc90317274)

[4.0 Results and Analysis 41](#_Toc90317275)

[4.1 Results and Analysis 41](#_Toc90317276)

[*4.1.1 RQ1: How effective are SBFL, IR and Text Search Engine technique individually / standalone?* 41](#_Toc90317277)

[*4.1.2 RQ2: How effective can we combine the techniques for better fault localization?* 44](#_Toc90317278)

[*4.1.3 RQ3: What is the run-time cost of standalone techniques and combined techniques for each program? And how long is the average of execution time for each technique on one bug?* 47](#_Toc90317279)

*[4.1.4 RQ4: How effective the combined approach is when compared with the state-of-the-art techniques? Which combination are the best that this research suggest?](#_Toc90317280)* [51](#_Toc90317280)

[4.3 Limitation 53](#_Toc90317281)

[5.0 Conclusion and Future works 54](#_Toc90317282)

[5.1 Summary 54](#_Toc90317283)

[5.2 Future work and outlook 55](#_Toc90317284)

[Bibliography 56](#_Toc90317285)

[Appendix 63](#_Toc90317286)

# 1.0 Introduction

This chapter is a brief introduction on the motivation of this research, as well as research’s motivation, objectives, hypothesis and problem statement.

## Objective and Motivation

In this subsection, I will briefly explain on my thesis research objective and motivation but before that I will explain first the definition of debugging and its general ideas. Debugging process is an action taken following a program failure and are defined as “determining what runtime faults led to a runtime failure or what software errors were responsible for those runtime faults and modifying the code to prevent the runtime faults from occurring” (fitzgerald, et al., 2008).

In other words, debugging is a process where developer identify what cause the fault and preventing the occurrences by modifying, adding or deleting the code (Masri, 2015). The first activity is referred to as fault localization, while the other one is called a fault repair or bug fixing. Though both activities are vital, my research will only be focusing on the first activity which is fault localization since it considered to be more critical as it is a prerequisite for fault repair.

Improvements are being made to produce effective and efficient debugging tools since decade ago by (Weiser, 1981), (Korel, et al., 1988), (Cleve, et al., 2000), (Jones, et al., 2002), (Renieres, et al., 2003), (Liblit, et al., 2005) , (Liu, et al., 2005), (Wong, et al., 2009), (Srivastav et al, 2010) , (Parnin, et al., 2011), (Shu, et al., 2016), (Sohn, et al., 2017), however debugging is still considered to be an expensive, time consuming activity. My research is focusing on the attempts to identify improvements to fault localization tools.

Latest research by (Zou, et al., 2019) on comparison between all fault localization techniques shows that SBFL technique is the most effective technique in fault localization while IR technique are one of the fastest techniques in fault localization. Fault localization is the specific task of identifying the faulty core of a program in which components are responsible for an error. Alternatively, fault localization can be thought of as reporting which portions of a system should be modified in order to correct an error (Groce, 2005).

In this thesis, I have combined two state-of-the-art technique in fault localization which presented in (Poshyvanyk, et al., 2007)and (Abreu, et al., 2007) that applies Information-Retrieval Based Fault Localization (IR) and Spectrum Based Fault Localization (SBFL) respectively, with Text Search Engine approach. Since the techniques are using different information sources for analysis fault localization, it is interesting to know how much these techniques are correlated to each other. A recent study also confirms that integrating more information sources significantly outperforms any standalone techniques in Fault Localization (Zou, et al., 2019).

The main objective of my research is to observe the relationship between these when used for fault localization. In other word, to investigate how the techniques can be combined, and what results can be produced through the combination. The motivation for combining these techniques is:

1. When fault localization results using Information Retrieval techniques unable to localize fault location, SBFL may provide more precise information.
2. When fault localization results using Spectrum Based technique unable to localize fault location, this situation may be supplemented with analysis from IR.
3. When both IR and SBFL technique unable to localize fault, Text Search Engine might help to shed the light in fault localization.

Second objectives of my research are to evaluate the integration of these technique via experiments to demonstrate the effectiveness and efficiency of using these techniques together for fault localization.

## Hypothesis, Problem Statement and Research Question

In this subsection, I will briefly explain on my thesis research hypothesis, problem statement and research methods. Fault localization considered to be one of the most expensive (Jones, et al., 2005) , (Wong, et al., 2010) , (Srivastav et al, 2010), (Sun, et al., 2016), tedious and time consuming activities in the debugging activity (Wong, et al., 2010) , (Agarwal, et al., 2014). Example of Fault localization technique are such as Spectrum Based Fault Localization (SBFL), Mutation based Fault Localization, Dynamic Slicing, Stack Traces, Predicate Switching, Information Retrieval (IR) and History-Based Fault Localization (Sun, et al., 2016) (Zou, et al., 2019) .

Now, I will briefly explain about SBFL and IR technique that related to my research, as well as the issues that has raised concern. In SBFL technique, location of the fault will be determined after suspiciousness result are generated. The suspiciousness value for each line of code is calculated based on the pass and failed formula calculation. The idea was, the suspiciousness value with the highest number, holds the higher chances of probability that the line of code contains fault or in other word, the fault location. This contribution of idea has been a state-of-the-art technique for quite some time and massive research has evolved since.

However, the suspiciousness value that generated from SBFL technique for fault localization is not always accurate. Top N has been used to measure the accuracy of the result on many faults localization study (Rao, et al., 2011), (Sisman, et al., 2012), (Zhou, et al., 2012) , (Saha, et al., 2013), and most research and study agree that the accuracy of the technique is consider accurate when the location of fault is in between top 1, top 5 and 10 result generated.

Research made by (Zou, et al., 2019) where they perform an experiment to compare between all standalone technique in fault localization family such as Spectrum Based Fault Localization (SBFL), Mutation based Fault Localization, Dynamic Slicing, Stack Traces, Predicate Switching, Information Retrieval (IR) and History-Based Fault Localization, found that SBFL is the most effective fault localization technique. SBFL manage to localize about 44% and 43% faults of in the top 10 result where Ochiai and DStar have the best performance on all metrics respectively.

In the study made by (Le, et al., 2015) , while running their experiment, they consider an output of a fault localization tool to be effective if the root cause appears in the top 10 most suspicious program elements and this approach has been used also in other and previous research. To evaluate the SBFL result reliability, this indicator has also been used in my study to the suspiciousness list generated. Now, the problem arises when the location of the fault is not in the top 10 of the suspiciousness results.

SBFL techniques aim to pinpoint faulty program elements by sorting them only by their suspiciousness scores and developers tend to resort to a different debugging strategy if they do not find the fault in the first positions of a suspiciousness list (De Souza, et al., 2017). The question would be “How can developer determine the location of the fault if it does not appear in the suspiciousness result, or it does appear but not in the top 10 list?”. “What additional information can they use to assist them in fault localization?”.

Traditional approach by manually finding the fault line by line is tedious and time consuming, probably can be done if it involves small programs or simple one, but “how if it involves a huge program or a real-world program and to make it worse the complex one?”. (Xuan et al, 2014) found that to further improve the fault localization effectiveness, extra information sources should be introduced rather than only considering the SBFL technique.

Here is when my research come into sense where the combination of more than one fault localization technique in enhancing fault localization performance. The hypothesis of my research is the combination of more than one fault localization technique increase the result accuracy and effectiveness in locating fault. Another question arise will be “which technique should I choose to make sure optimum result?”.

Research by Rao et al, on comparative study between IR model for fault localization, they found that sophisticated models like Latent Dirichlet Allocation Model (LDA), Latent Semantic Analysis Model (LSA), and Cluster Based Document Model (CBDM) do not outperform simpler models like Unigram Model (UM) or Vector Space Model (VSM) for IR based bug localization in large software systems, where in general almost 50% of the total fault of relevant file at the rank of 10 are managed to be retrieved (Rao, et al., 2011).

In fault localization research, VSM has been shown to outperform many other IR approaches (Wang, et al., 2014) (Rao, et al., 2011) while Ochiai is the best in performance compared to other technique (Zou et al, 2019) in SBFL. This are the reason why this research choose Ochiai technique for SBFL and VSM technique for IR, because of their outstanding performance other than its availability.

IR-based technique assist people in sorting through vast amounts of data or information as quickly and efficiently as possible (Lillis, et al., 2016), where it measures the textual similarity between the bug report and the source files. This approach takes a bug report and source file as input, rather than a set of test cases, and generate a list of relevant source code files as output based on the bug report query (Wong, et al., 2016).

IR- based fault localization techniques do not require program execution information, such as passed and failed test cases (Zhou et al, 2012) where this type of technique may add value and give advantage when IR and SBFL technique are being combined since additional information are best gathered between different type of source to avoid redundancy and confusion.

In IR, due to the nature of language use, the terms that constitute a topic are often semantically related (Blei, et al., 2003), (Chen, et al., 2015) , for example if the topic of the bug report’s results is about “period”, so the source code that contains topic “period” as the highest result might be the location of fault. This might help in contributing accuracy of fault localization in SBFL.

The situation where a developer unable to allocate fault is not new and happens in most fault localization technique since there are no “one-size-fits-all solutions” in addressing fault localization. Same as SBFL, IR technique also face same situation where at some point the topic generated from the result unable to assist developer in allocating fault where the technique unable to map the similarity between documents.

Since different kind source of information might help developer in fault localization, hence SBFL are being applied. Optimization of information from the test cases execution for example, the top 10 of suspiciousness value might contain the location of the faults. The combination of dynamic and static analysis is a well-known and powerful combination (Dit, et al., 2011).

Though the combination of the most effective technique which is SBFL technique and the fastest technique which is IR technique in fault localization might produce a promising result, adding another one more tool which is Text Search Engine can present more accurate result, as it might cover different analysis that gain from static and dynamic debugging. All factors that mention above has driven my research into below research questions:

RQ1: How effective are SBFL, IR and Text Search Engine technique individually/standalone?

This question helps my research to understand the performance of widely-used techniques.

RQ2: How effective can we combine the techniques for better fault localization?

This question considers ways of combining different techniques and evaluates the performance of the combined technique.

RQ3: What is the run-time cost of standalone techniques and combined techniques for each program? And how long is the average of execution time for each technique on one bug?

The previous question concerns on the fault localization standalone and combined technique’s accuracy’s performance, while this question considers the time taken for each of the techniques. The best combination technique is the one that have accuracy and time cost balance.

RQ4: How effective the combined approach is when compared with the state-of-the-art techniques? Which combination are the best that this research suggest?

This research question compares the performance of all approach and to find the answer to research questions above, series of experiment will be done to observe the individual results for SBFL, IR and Text Search Engine, the combination technique complementing each other, and to identify which combination’s performance is the best in. term of accuracy and time cost.

## 1.3 Summary and structure of the thesis

Chapter 2 briefly explains the fundamental definitions of basic term that will be used in this thesis, and the background reviews including related work of research in fault localization. Each fault localization technique will be discussed and previous research on hybrid technique will be highlighted.

In Chapter 3, is where the experiment methodology will be explained in detail. It will include the Experiment set-up, Experimental subjects and the tools that has been used to run the experiment.

In Chapter 4, the result and analysis of each technique either individually or as a combined technique will be discussed. Limitation of this research will also be highlighted.

Chapter 5 is about research summary and conclusion including future works that possibly be continued after this research.

# 2.0 Background and related work

This chapter is a background and related work of this research, and this includes the intersection of Fault Localization techniques in fault localization.

## 2.1 Fundamental definitions and terminology: error, fault, bug, test

To avoid confusion in further discussion, I will explain first the fundamental definitions and terminology of error, fault, bug and test. According to (De Souza, et al., 2017), error is a tricky term, which is also sometimes used to refer to as a fault, failure, or mistake. Usually, the term is often used to indicate an incorrect state during the program’s execution. (McCall, et al., 2014) and (Munson, et al., 2006) define error as the nature of the problem in the source code that causes the compilation or execution to fail. In other word, error is caused by a mistake made that led to unsuccessful execution or code compilation (Krawiec, 2018).

(Hristova, et al., 2003) did significant research in understanding common error that occur among novice programmer and divide it to three category which are Syntax errors, Semantic errors and Logic errors. *Syntax errors* refers to mistakes in the spelling, punctuation and order of words in the program while *Semantic errors* occur from a mistaken idea of how the language interprets certain instructions. *Logic errors* are general errors that often occur which may cause unintended result, and sometimes even without failed execution.

On the other hand, bug can be defined as a program behavior that deviates from its specification (Allen, 2002). (Ko, et al., 2005) define bug as a combination of one or more errors in the code (e.g., software errors), which may produce errors in execution (e.g., runtime faults), which in turn may produce failures in program behavior (e.g., runtime failures). While fault is a manifestation of an error in software (Munson, et al., 2006) and are also called bug or defect (De Souza, et al., 2017). As this is a very clear definition and to avoid confusion later, the definition that will be used for this thesis writing is the term “fault” instead of “bug”.

Test or testing is the main source of information for debugging and a very important process in fault localization. Testing is defined as “an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component” (IEEE, 2008). Testing is performed to make sure that intended function in the program that are tested are working as expected and at the same time the testing requirements are used as a guarantee that the code is widely tested, and most of the program elements are executed (De Souza, et al., 2017).

## 2.2 Spectrum Based Fault Localization (SBFL)

Initially, fault localization was performed manually where developers observe failed test cases and then search the source code for faults. This includes practice such as inserting print statements and breakpoints, checking the stack trace, and verifying failing test cases (De Souza, et al., 2017). Since then, many research has been done to improves fault localization task by created techniques with different approaches.

SBFL or also known as coverage based Fault Localization is a technique that rely on analysing structural execution profiles. SBFL is a dynamic debugging analysis and has received a lot of attention due to its simplicity and effectiveness (Xie, et al., 2013). SBFL is a technique that use a ranking metric to calculate statements’ suspiciousness (De Souza, et al., 2017). Suspiciousness is when a statements more likely to contain faults or have faulty elements (Alipour et al, 2012). The suspiciousness values are calculated according to the frequency of the statements in passing and failing test cases.

Various Spectrum based bug localization (SBFL) approaches have been proposed in the literature (Jones, et al., 2005), (Abreu, et al., 2006), (Lucia, et al., 2010), (Liblit, et al., 2005), (Liu, et al., 2005), (Artzi et al, 2010), (Zeller et al, 2002), (Cleve, et al., 2005), (Lucia, et al., 2014). These approaches analyze a program spectrum which is a record of program elements that are executed in failed and successful executions and generate a ranked list of program elements. Many of these approaches propose formulas that can be used to compute the suspiciousness of a program element given the number of times it appears in failing and successful executions (Le, et al., 2015).

For example, Jones and Harrold proposed Tarantula that uses a suspiciousness score formula to rank program elements (Jones, et al., 2005). Later, Abreu et al. proposed another suspiciousness formula called Ochiai (Abreu, et al., 2006), which outperforms Tarantula. Then, Lucia et al. investigated 40 different association measures and highlighted that some of them including Klosgen and Information Gain are promising for spectrum-based bug localization (Lucia, et al., 2010), (Lucia, et al., 2014). A study conducted by, Xie et al. on theoretical analysis found that several families of suspiciousness score formulas outperform other families and then theoretically they compared the performance of the formulas produced by genetic programming and identified the best performing ones (Xie, et al., 2013).

Next, Yoo et al, proposed to use genetic programming to generate new suspiciousness score formulas that can perform better than many human designed formulas (Yoo et al, 2012). Most recently, Xuan and Monperrus combined 25 different suspiciousness score formulas into a composite formula using their proposed algorithm named MULTRIC, which performs its task by making use of an off- the-shelf learning-to-rank algorithm named RankBoost (Xuan, et al., 2014). MULTRIC has been shown to outperform the best performing formulas studied by (Xie, et al., 2013) and the best performing formula constructed by genetic programming (Yoo et al, 2012), (Xie, et al., 2013).

Several research have also been conducted to investigate the shortcomings of SBFL procedures in order to better understand why SBFL has yet to be widely adopted by practitioners. Studies in early SBFL technique shows that following requirements must hold in order for Tarantula to be effective: (1) a defect is due to a single faulty statement; (2) statements are independent of each other; and (3) executing a faulty statement leads most of the time to a failure (Denmat, et al., 2005). Unfortunately it is obvious when dealing with complex programmes including nontrivial faults, the aforementioned requirements are unlikely to be met. Though several improvement has been made on the formulas and claimed to be better than the previous or exist version, (Baah, et al., 2010) demonstrated that the Tarantula (Jones, et al., 2002) metric and the popular Ochiai metric (Abreu, et al., 2006) are both grounded in statistical approaches, specifically correlation, showing that focusing on improvement of the formulas only might not help SBFL for optimum results hence combining it with other alternatives FL technique might shed some light to this issues.

## 2.3 Information retrieval- based (IR)

Information retrieval-based technique or also known as IR was initially used to index text and search for documents, until then later on it was discovered that it may assist developer in debugging where (Marcus, et al., 2004) paved the way by demonstrating that Latent Semantic Indexing (LSI) could be used for concept location in source code. IR is a static debugging approach that statically locating bugs using different types of analyses such as bug model, reports and source code file rather than a set of test cases, without any actual run of the program (Saha, et al., 2013), (Hovemeyer, et al., 2004), (Lukins, et al., 2008). IR fault localization techniques do not require program execution information, such as passed and failed test cases (Zhou, et al., 2012) where it just generate a list of relevant source code files as output (Wong, et al., 2016).

Akbar highlighted in his research that there are three generations of IR bug localization that can be identified over a fifteen year of research span (Akbar, 2020), where the first generation are based on simple bag-of-words (BoW) assumptions (Marcus, et al., 2004), (Kuhn, et al., 2007), (Lukins, et al., 2008), (Rao, et al., 2011), the second generation utilizing information such as bug reports, source code files, including version histories changes to improve BoW based systems performance (Zhou, et al., 2012), (Sisman, et al., 2013), (Wong, et al., 2014) (Wong, et al., 2014), (Moreno, et al., 2015), and the third one, focusing on the terms order and semantics relationships modelled has been taken into consideration to improve the IR tools (Wen, et al., 2016), (Rahman, et al., 2018), (Akbar, et al., 2019), (Xiao, et al., 2018), (Lam, et al., 2017), (Nguyen, et al., 2017).

IR techniques identify the elements of the software system that need to be modified to correct a bug by assisting people in sorting through vast amounts of data or information as quickly and efficiently as possible (Lillis, et al., 2016). These techniques do not attempt to identify every element of the software system that must be fixed. Instead, they aim to identify a starting point from which correction of the bug can be undertaken.

### 2.4.1 Text retrieval technique

Advanced knowledge and thorough understanding of how a system is built and its various components interact are required in manually attempting fault localization from the information provided in the bug report. Various work has been looked into techniques to aid developer in decreased human effort spent in fault localization activity. Text retrieval technique is a technique in which the system’s source code is indexed into a search space and then queried for code related to a given bug report (Mills et al, 2020). This is a typical IR approaches that locate fault files by comparing the bug reports with the source files (Zhong, et al., 2020). Numerous IR fault localization approaches that employ techniques to calculate the similarity between a bug report and a program element (e.g., a source code file) that has been proposed (Rao, et al., 2011), (Lukins, et al., 2010), (Le, et al., 2013), (Sisman, et al., 2012), (Zhou, et al., 2012), (Saha, et al., 2013), (Wang, et al., 2014), (Ye , et al., 2014).

Lukins et al. used a topic modeling algorithm named Latent Dirichlet Allocation (LDA) for bug localization (Lukins, et al., 2010). In the IR community, historically, Vector Space Model (VSM) is proposed very early about four decades ago (Salton, et al., 1975) , followed by many other IR techniques, including Smoothed Unigram Model (SUM), LSI (Marcus, et al., 2004) and LDA (Blei, et al., 2003). Rao et al, evaluated the performance of several standard IR techniques for bug localization including VSM and SUM and LDA (Rao, et al., 2011) by conducting a comparative study between the tools on bug localization task. In the study, they found that simple text models such as VSM and SUM perform better than more sophisticated models such as LDA. Since then, number of works have been done to improve the effectiveness of standard IR models by considering more information, applying advanced techniques, and refining queried bug reports.

More recently, several approaches which considers information aside from text in bug reports to better locate bugs were proposed. Zhou et al. proposed an approach named BugLocator that includes a specialized VSM (named rVSM) and considers the similarities among bug reports to localize bugs (Zhou, et al., 2012). Next, Saha et al. proposed an approach that considers the structure of source code files and bug reports and employs structured retrieval for bug localization, and it performs better than BugLocator (Saha, et al., 2013).

### 2.4.2 Stack trace technique

Stack trace fault localization is a technique that needs to execute the test cases at least once before it be able to track the fault. A stack trace is the list of active stack frames during execution of a program. Each stack frame corresponds to a function call that has not yet returned. Stack traces are useful information sources for developers during debugging tasks. When the system crashes, the stack trace indicates the currently active function calls and the point where the crash occurred (Zou, et al., 2019).

Bug reports often contains stack trace information; however, existing approaches often treat this information as a plain text. (Wong, et al., 2014) propose to use segmentation and stack-trace analysis to improve the performance of bug localization. (Moreno, et al., 2014) proposed a technique named Lobster where it uses the stack trace that recorded in the bug report to compute the similarity with the code elements and the source code of the programs.

(Davies et al, 2013)

Latest research by (Khvorov, et al., 2021) present the first approach to computing stack trace similarity based on deep learning based techniques and their study demonstrated that their approach outperform state-of-the-art approach on both private JetBrains dataset and open-source NetBeans data. However, stack trace analysis usually works on crash faults and less efficient for other type of faults.

### 2.4.3 History-based technique

History-based fault localization is a technique that only needs to examine the development history of the codes (Zou, et al., 2019) where are usually used for fault prediction. This technique ranks the elements in a program by their likelihood to be defective. Generally, fault prediction and fault localization are considered as different issues, moreover fault prediction runs before any failure has been discovered (Kim, et al., 2007).

(Dallmeier, et al., 2007) introduce a tools named iBUGS where it collects all past successes and failures of a project to leverages it’s history by automatically extract benchmarks for bug localization tools. These benchmarks are useful for both static and dynamic bug localization tools. Sisman and Kak proposed a version history aware bug localization technique which considers past buggy files to predict the likelihood of a file to be buggy and uses this likelihood along with VSM to localize bugs (Sisman, et al., 2012).

Wen et al, propose an approach named “Locus” to locate bugs from software changes, and evaluate it on six large open-source projects. (Wen, et al., 2016). While (Wang, et al., 2016) proposes an approach to prioritizing test cases based on historical data where the priorities of test cases are initialized based on requirement priorities and then are calculated dynamically according to historical data in regression testing. (Le, et al., 2016) propose a technique that utilizes the information of bug fixes across projects in the development history to effectively guide and drive a program repair process.

Wen et al, proposed a Historical SBFL where it is a combination of two technique from different family. The approach record the version histories on how bugs are introduced to software projects and this information reflects the root cause of bugs directly where at the same time, the evolution histories of code can also assist to differentiate those suspicious code entities ranked in tie by SBFL (Wen, et al., 2018).

## 2.4 Others Fault Localization Technique

### 2.4.1 Program slicing

Slicing and SBFL have a similar mechanism where they need to trace the execution of test cases, once. The main difference between these two techniques is their efficiency where SBFL needs to trace all the test cases while slicing only needs to trace failed test cases. Dynamic slicing is an enhance technique that developed from static slicing. A slicing criterion is a set of variables at a program location where they might be variables that have unexpected or undesired values. A program slice is a subset of program elements that potentially affect the slicing criterion (Xu et al, 2005). Program slicing was introduced as a debugging tool to reduce a program to a minimal form while still maintaining a given behavior (Weiser, 1981) and static slicing only uses the source code and accounts for all possible executions of the program.

While a dynamic slicing focuses on one execution for a specific input (Korel, et al., 1988). The key difference between dynamic slicing and static slicing is that dynamic slicing only includes executed statements for the specific input, but static slicing includes possibly executed statements for all potential inputs. Since dynamic slices are significantly smaller, they are more suitable and effective for program debugging compared to static slices (Zhang et al, 2007).

### 2.4.2 Mutation Based Fault Localization (MBFL)

MBFL is the slowest technique compared to other fault localization technique. This technique not only needs a lot of time to execute it, but it also needs a huge capacity of hardware and software to run it. Contrasting from conventional program execution, MBFL uses information from mutation analysis (Jia et al, 2011), as inputs to its ranking metric or risk evaluation calculation. MBFL strategies assess whether the execution of a statement affects the result of a test by injecting mutants, whereas SBFL techniques are evaluate based on whether a statement is executed or not. A mutant in most cases changes one expression or statement by replacing one operand or expression with another (Pearson, et al., 2017). It's more suspicious if a program statement affects failed tests more frequently and passed tests less frequently.

(Nica et al, 2010) proposed a technique to reduce bug candidates by using constraint-based debugging. First, statements that do not violate the constraints and that explain the failing test cases are deemed bug candidates. Second, the technique generates mutants for each bug candidate. Mutants that make the failing test cases pass are used to suggest possible faulty sites. (Moon, et al., 2014) proposed a technique that uses mutation to modify faulty and correct statements. The rationale is that, if a mutant inserted in a faulty statement reduces the amount of failing test cases, then the faulty statement is more likely to be faulty.

Conversely, a mutant inserted in a correct statement which generates more failing test cases is less likely to be faulty. (Hong et al, 2015) proposed a similar approach for multilingual programs. Mutation testing is also used to seed faults for experiments, and to suggest fixes for program repair (Weimer, 2006) (Debroy et al, 2014). (Ali, et al., 2009) used mutation testing to generate faults and shown that these faults are similar to real faults.

### 2.4.3 Predicate switching

Though predicate switching (Zhang, et al., 2006) is similar to MBFL, this approach are designed for faults in control flow or faults that related to it. A predicate, or may also called a conditional statement, directs the execution of various branches. If a failed test case can be made to pass by changing the evaluated result of a predicate, the predicate is said to as a critical predicate, and it could be the source of the defect.

Predicate switching begins by tracing the failed test's execution and identifying all instances of branch predicates. The tests will be re-run several times, where each time forcing a different predicate's outcome. If switching a predicate yields the correct output, the predicate is called a critical predicate since it potentially could be the cause of the error.

Predicate switching and MBFL approaches are similar in that they both use mutations and look at how the execution results change. However, because predicate switching alters the control flow rather than the program itself, we treat it as a separate family. Predicate switching, for example, inverse one evaluation at a time rather than all evaluations if a conditional expression has been examined numerous times during program execution. Furthermore, as far as we are aware, previous work by (Pearson, et al., 2017), (Li, et al., 2017) also does not include predicate switching as an MBFL approach.

### 2.4.4 Delta Debugging

Delta debugging is a technique for simplifying a failing programme input in order to make the debugging process easier (Zeller, et al., 2002) . It iteratively reduces the size of a failing input until it identifies the smallest possible part of it that likewise causes failure through experimentation. Delta debugging is based on the idea that a smaller inputs cover less code thus need less debugging activity, and delta debugging inputs can be simplified while still remaining valid by removing parts from within (Masri, 2015). Despite the fact that none of these assumptions holds true in most cases, delta debugging has proven to be effective not only in simplifying input but also in a variety of other debugging settings aimed at narrowing down failure causes. One drawback of this technique in general is that it can change the programme state (or input) in ways that are difficult to achieve in the original context, resulting in infeasible states (or inputs) that will obstruct the debugging effort.

The work in (Zeller, 2002) reduces the cause of failure to a limited set of variables. It leverages memory graphs (Zimmermann, et al., 2001) to find the common variables between a passing run and a failing run. It then contrasts the program states, induced by the common variables, of the two runs. The suspiciousness of a given variable is evaluate by replacing its value in the passing run with its corresponding value in the failing run and re-executing the program. The variable is deemed suspicious if the identical failure is observed; otherwise, it is excluded from the set of failure-inducing variables.

(Burger, et al., 2011) described a method for reproducing an observed failure in the simplest possible way. The interaction between programme objects is captured and reduced from a single failing run to a simple unit test consisting of a set of method calls that reliably reproduces the failure. JINSI, a tool developed by the authors and based on the record/replay mechanism, delta debugging, and programme slicing, enables the suggested technique.

## 2.5 Hybrid Techniques and related works

Over the decade , there are many researches has been done that applying a solution to combine or hybrid techniques in fault localization. Both SBFL and IR techniques ultimately generate a ranked list of program elements that likely contain a bug; however, they only consider one source of information either bug reports or program spectra, which is not optimal (Hoang, et al., 2019). Different techniques in SBFL family may contain strongly correlated information on real-world projects. To further improve the fault localization effectiveness, extra information sources should be introduced rather than only considering the SBFL family (Zou, et al., 2019). Since this research are focusing on SBFL and IR technique in fault localization area, further explanation and elaboration will be made on this topic. This subsection will explain the intersection research on SBFL and IR technique, and also the combination of varies fault localization technique with each other.

There is so much research that has been done in recent decade regarding Fault localization and it seems to be more promising to find new information sources than optimizing existing information sources (Zou, et al., 2019). Recent studies by (Sohn, et al., 2017) (Li, et al., 2017) (Le, et al., 2016) also confirm that integrating more information sources significantly outperforms any techniques in the SBFL family and the combined techniques significantly outperform any standalone technique. Le et al, build an oracle that could predict whether the output of a fault localization tool either reliable or not before developer proceed to localize fault using the combination of SBFL and machine learning (Le, et al., 2013).

Another hybrid technique to allocate bugs has been done such as research by (Xuan, et al., 2014) and research by (Le, et al., 2016) where they combine SBFL and learning-based technique. (Sohn, et al., 2017) where they combine SBFL, Learning based, Genetic Programming and Linear rank. Research by (Li, et al., 2017) on combining Learning-based and MBFL, (Wen, et al., 2018) that are using SBFL and History-based to allocate bugs. While (Jiang, et al., 2019) did a systematical empirical study on the combination of SBFL and Statistical Debugging techniques, (Cui, et al., 2020) propose an approach by combining SBFL and MBFL to improves localization accuracy.

Research made by (Le, et al., 2015) on Multi-modal feature location takes as input a feature description and a program spectra, and finds program elements that implement the corresponding feature. There are also several multi-modal feature location techniques proposed in the literature by (Poshyvanyk, et al., 2007), (Liu et al, 2007), (Dit, et al., 2013) with varies combination of fault localization technique.

Poshyvanyk et al. proposed an approach named PROMESIR that computes weighted sums of scores returned by an IR-based feature location solution, LSI technique by (Marcus et al, 2003) and a SBFL technique by (Jones et al, 2005). They later rank the program elements based on their corresponding weighted sums (Poshyvanyk et al, 2007). Then, Liu et al. proposed an approach named SITIR which filters program elements returned by an IR-based feature location solution , LSI technique by (Marcus et al, 2003) if they are not executed in a failing execution trace (Liu et al, 2007). Later, Dit et al. proposed a LDA technique and Genetic Algorithm (GA) (Dit, et al., 2013).

Since techniques in different families use different information sources, it is interesting to know how much these techniques are correlated to each other and in (Zou, et al., 2019) paper, they found that different techniques in SBFL family may contain strongly correlated information on real-world projects. The research suggest that to further improve the fault localization effectiveness, extra information sources should be introduced rather than only considering the SBFL family. This research is focusing on one of the critical processes of debugging task which is fault localization, and this research attempts to improve fault localization tools by combining two technique of fault localization from different family.

## 2.6 Text Search Engine

Text search engine is a tools has been used for decades, and indeed it is a very powerful tools that able to cover various fields in text searching process including text, image, signal, and speech processing (Hakak, et al., 2019). String matching algorithms are commonly and widely being used with text search engine to search a certain word or sequence of word in a huge text document (Kabir, et al., 2014) moreover, it work excellently regardless of databases, scripts, and applications (Hakak, et al., 2019).

Indexes are used by search engines to locate relevant documents based on a search query. The search engine also employs an IR techniques such as stemming, lemmatization, and word embeddings, among others (Ledel, et al., 2021). Not just combining fault localization technique from different family such as SBFL and IR technique, combining a search engine as part of the bug localization process that are never done before, are the main contribution to my research.

The text search engine are being used in my research the to find similarity of bug report with the source codes. Though this approach sounds almost the same with another technique that are used in this research which is IR using cosine similarity to find the similarity between documents in bug reports and source codes, however, a search engine covers other dimension of analysis where it is able to present a more accurate result, since it goes beyond cosine similarity for text matching.

There are no research that has been made specifically on Fault Localization with the use of text search engine before , however, one latest research by (Ledel, et al., 2021) that investigate either normal text search engines can improve existing bug localization approaches found that it does improved bug localization performance and it is a useful extension to existing approaches. (Ledel, et al., 2021) introduce Broccoli tools that combined several technique in IR such as version history, report similarity, structure and stack traces with Text Search Engine to improve fault localization performance. The performance of proposed approach has been evaluated against seven state-of-the-art bug localization algorithms of IR technique on open source projects in two data sets. My research expand this idea by combining Text Search Engine technique with two different Fault Localization Technique which is SBFL and IR technique on Defects4j datasets. This highlight the differences between my research and (Ledel, et al., 2021) where in my research the improvement for fault localization are made by combining SBFL, IR and Text Search Engine.

# 3.0 Experiment methodology

This chapter will explain about methodology of experiment for my research. This chapter consist of four subsection which are 3.1. Experimental set-up, 3.2. Experimental subjects, 3.3 Studied techniques and their implementations where the illustration of how the approach works.

## 3.1 Experimental set-up

Experiments are usually referred as research-in-the-small, since they are concerned with a limited scope and most often are run in a laboratory setting (Wohlin et al, 2003). The purpose of my experiments is to observe the relationship between these techniques in fault location. This experiment is to compare and understand “when does the combine technique are accurate or helpful” and “when does it is not?”. “Does combining technique helpful when individual technique is not?”.

Since each technique are using different type of sources to execute, the result produced will also be different in term of the granularity level for example, SBFL technique usually are analyzed at method level while IR technique are analyzed at File level. However, aligned with IR purpose in fault localization, as long as it manages to identify a starting point from which correction of the bug can be undertaken are suffice (Lillis, et al., 2016).

In the study made by (Le, et al., 2015), while running their experiment, they consider an output of a fault localization tool to be effective if the root cause appears in the top 10 most suspicious program elements and this are the rules that are also been applied in my experiment as an indicator for the list result generated. Research frameworks of my experiment are shown in Figure 3.1 below and will be explained further in section 3.3 from each technique perspective.

Diagram

Description automatically generated

Figure 3.1: Research Framework of Experiment

Initially all three techniques will be executed individually to assess its accuracy and time cost. Then all techniques results are combined to evaluate how much different the result generated compared to individually executed technique. The purpose of this experiment is to analyze which combination are the best in term of accuracy and time spent to localize fault. Result of the best combination technique will be suggested at the end of the experiment.

## 3.2 Experimental subjects

My thesis’s experiments evaluate fault localization techniques using the [[1]](#footnote-1)Defects4j framework (Just, et al., 2014), version v1.0.1, see Table 3.1 for details. Defects4J contains 395 faults minimized from real-world faults in six open-source Java projects. Many previous studies on fault localization used Defects4J as their benchmarks (Pearson et al, 2017), (Le, et al., 2016) (Ma, et al., 2015). Defects4J provides a faulty version and a fixed version of the project. It also contains a suite of test cases for each fault that contains at least one failed test case that triggers the fault.

|  |  |  |
| --- | --- | --- |
| Program | Description | Faults |
| Joda-**Time** | A standard date and time library for Java | 27 |
| **Mockito** | A mocking framework to write tests in Java | 38 |
| Apache Commons **Math** | A lightweight mathematics and statistics library for Java | 106 |
| Apache Commons **Lang** | A complement library for  java.lang | 65 |
| JFree**Chart** | An open source framework for Java to create chart | 26 |
| Google **Closure** compiler | A tool to optimize JavaScript source code | 133 |
| Total |  | 395 |

Table 3.1: Defects4J Dataset (version 1.0.1) with each program description. ‘Faults’ is the number of faults of the program.

Defects4j program are used for all experiment in this research to ensure comparability and uniformity between experiments. Defects4j is one of many benchmarks for real-world programs, where provides an extensible set of reproducible bugs derived from Java software systems in the real world, along with a supporting infrastructure to use these bugs aims at advancing software engineering research (Just, et al., 2014), (Gay, et al., 2020). The defects4j database contains of 357 bugs from 5 programs initially, and since then has grows into 835 bugs from 17 programs (Version 2.0.0). However, for my experiment, I’m only using 6 programs with 395 bugs from defects4j due to time constrains.

Defects4j has been used as supporting resources for professionals in both software testing and debugging study (Gay, et al., 2020) where it can be used as a benchmarks to evaluate the effectiveness of automated test generation and corresponding fitness function (Rueda et al, 2016) (Shamsiri et al, 2015), automated program repair (Martinez et al, 2016) (Motwani et al, 2020), and fault localization (Pearson, et al., 2017) research.

## 3.3 Studied techniques and their implementations

### 3.3.1 SBFL Experiment

SBFL is the most effective technique in fault localization family where Ochiai have the best performance on all metrics (You et al, 2019) (Zou, et al., 2019). According to Vancsics et al, Ochiai obtained the best results (Vancsics et al, 2020) where it is more effective in locating individual bugs (Oo et al, 2020) and manage to handle multiple faults (Xiaobo et al, 2018).

Unlike other technique in SBFL, Ochiai is more effective for object-oriented programs thus, most SBFL-based repair tools also use Ochiai (Motwani et al, 2020). Since all programs in Defects4j are Java based that are object-oriented, and based on all these finding, other than its availability, Ochiai has been chosen for this individual SBFL experiment. Figure 3.2 below is a framework of SBFL technique in this experiment.

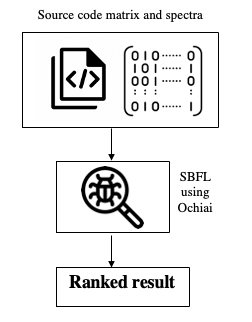


Figure 3.2: Spectrum Based Fault Localization technique (SBFL) framework

*Program spectrum and suspiciousness*

In SBFL, a program spectrum also called as spectra is a measurement of run-time behavior, such as code coverage (Harrold, et al., 2000). Collofello et al, proposed that program spectra to be used for fault localization (Collofello, et al., 1986), where comparing program spectra on passed and failed test cases enable ranking of program elements. The more frequently an element is executed in failed tests, and the less frequently it is executed in passed tests, the more suspicious the element is (Zou, et al., 2019).

Ochiai is an enhanced version of SBFL technique that assigns a suspiciousness to each statement in the program based on the number of passed and failed test cases in a test suite that executed that statement (Abreu, et al., 2007). The intuition for this approach to fault localization is that statements in a program that are primarily executed by failed test cases are more likely to be faulty than those that are primarily executed by passed test cases (Jones et al, 2007). Figure 3.3 below is the formula that has been proposed by (Abreu, et al., 2007):

Text

Description automatically generated

Figure 3.3: Ochiai SBFL formula

A program *E* is a set of elements. Given a program element *e* ∈ *E*, we define the following notations:

• *failed(e)* denotes the number of failed test cases that cover program element *e*.

• *passed(e)* denotes the number of passed test cases that cover program element *e*.

• *totalfailed* denotes the number of all failed test cases.

• *totalpassed* denotes the number of all passed test cases.

SBFL technique are using probability algorithm between pass and fail execution of the test cases, so the result of this technique is in suspiciousness value are represent as; the highest value is 1 and the lowest value is 0. The line number and file name or method name of the fault also being showed together with the suspiciousness values. The suspiciousness values will be rank according to this value which at the same time it also contains the line of code number that are likely containing faults. Before running SBFL, program spectra or code coverage that includes class and method are taken from the program code. All requirement resources such as such as Time, Mockito, Math, lang, Closure and Chart program that are used for this experiment can be downloaded directly from Defects4J site. A fault localization technique outputs for SBFL technique are a ranked list of statement line with suspiciousness value.

### 3.3 IR technique Experiment

According to literature, information that are usually used for fault localization in IR can be divided into below components (Ledel, et al., 2021):

1. Document similarity: using the similarity between the text of a bug report and code files to identify possible locations.
2. Version history: exploitation of hot spots within the codes since the observation on files that contained bugs in the past often also contain bugs in the future.
3. Structure: the direct identification of file names that may be affected through exploiting the frequent patterns of file names.
4. Stack traces: exploiting information from stack traces as source for the bug localization.
5. Reporter information: use the assumption that the same reporter will use the same functionality of the software to find the package of the current bug.
6. Bug report comments: finding earlier bug reports by comparing the current bug with prior bugs, including their discussion. The location of the prior bug fixes can then be used to locate the current bug.

Nevertheless, this experiment chooses document similarity using VSM approach not only because VSM is a popular IR model that has the advantages of being simple and fast, but also have a ranking system that is as good as a vast variety of alternatives (Ngo, et al., 2010). Other factors that also count as pivotal role in determining which information are appropriate for this experiment includes information availability factor such as bug reports and source code of the programs that are easy to access and obtain as well as time constraint factor. Figure 3.4 below is a framework of IR technique using VSM in this experiment.

Diagram

Description automatically generated

Figure 3.4: Information Retrieval (IR) technique using VSM framework

Bug report information will be used to localize fault in source code files, and unlike SBFL techniques, IR techniques do not require program coverage information, but their generated ranking is based solely on source code files (Wong, et al., 2016). Before conducting the experiments, source code of the Defects4j program such as Time, Mockito, Math, lang, Closure and Chart are required. Since all the bugs that selected in this research are categorized as “fixed” bug so, the failing program version or the program version before bug repair are needed. This source code is necessary to be used in Information Retrieval technique and same goes with the bug reports.

*Bug Report and Source Code Requirement*

Research made by (Biggers, et al., 2014) found that the exclusion of comments and literals from the source of source code lowers the accuracy of the end results since bug report might contains natural language context. Later in their research they grouped the sources of source code text into three categories which are Identifier, comments and string literals and combination from the three sources of group are highly recommended to generate more accurate results. Below is the definition of the three source of source code.

**Identifier** are defined to be a class name, attribute name, method name, parameter name, local variable name, enumeration constant name, label name, or a generic/template parameter name (Abebe, et al., 2009) (Biggers, et al., 2014).

**Comments** generally are used either to map requirements to code or to describe the code (Vinz and Etzkorn, 2006).

**String Literals** generally are used either to convey information to the end-user such as an error message which usually contains domain information or to the developer such as a debugging message which usually contains implementation information. Copyright information are also included into string literals (Biggers, et al., 2014).

However, for this research, I only use Identifier and Comments sources since string literals information are not available except the copyright information where typically such data are not indexed, as they add no information about program purpose or behavior hence it has been removed.

The basic source of knowledge for developers to understand a fault is a bug report where, the summary gives a concise overview of the issue (Kim, et al., 2013). That is why bug report is important so that developers manage to map the issues raised in bug report to the fault location in the source code.

Insufficient information generated from bug report may obstructing fault localization process. A complete and good bug report is a combination of bug report’s title and its description (Dit, et al., 2011). However, (Zimmermann, et al., 2010) added that a quality bug report are the one that includes an codes attachment or code snippets.

To investigate this further, (Saha, et al., 2013) found that the important of program constructs such as class names and method names to present in bug reports where this might be effectively used to improve fault localization. While (Tantithamthavorn, et al., 2018) found that the best results from their experiment has to do with the similar textual characteristics between bug reports and source code. They also conclude that increasing the number of topics has little impact on the performance. However their experiment on the bug report representation in Eclipse program, that contains title only without description are sufficient in achieving best performance while for Mozilla system need both title and description showing that length of the documents does not matter as long as the information in it is good enough to use to localize fault.

One of criteria included in (Tantithamthavorn, et al., 2014) research for bug report information is they select only already-fixed issue reports which labelled as “fixed” and exclude issue reports where they could not establish a link to the source code entities. Based on literature evidence, for this research I decided that a bug report that will included in the experiments should have both title and descriptions, or at least contain class name or method name if no description included and best to contains codes attachment or code snippets. However, for bug report that does not have complete information, the documents will also being executed using IR technique to compare and confirm that the incomplete or insufficient information of bug report might affect the similarity of documents results.

*Bug Report and Source code extraction*

IR process started with extract information from each source code and bug report, and this includes information as comments and identifiers. Before writing the semantic information to the document collection, source code needs to be preprocessed first and the steps includes stemming, normalizing, removing stop words and splitting.

Stemming is a process is where we strip suffixes to reduce words to their stems for example “changing” becomes “chang”, and typically using the Porter stemmer algorithm (Porter, 1980). Normalizing is replacing each upper case letter with the corresponding lower case letter while filtering is removing common English language stop words such as “the”, “it”, “on” “an”. Same goes to the programming language keywords such as “if”, “while” are also removed.

Splitting is done by removing all punctuation, numbers including characters related to the syntax of the programming language such as “&&”, “->”. However, unlike common coding style convention, splitting process for this research does not include splitting the word where we retain the original (unsplit) tokens for example such as “AgeCalculator”, “PeriodType”, “LocalDate”. The main idea behind these steps is to capture the semantics of the developer’s intentions, which are thought to be encoded within the identifier names and comments in the source code (Poshyvanyk, et al., 2007) plus. This preprocessing step should be the same with bug report extraction process where all information of the bug report such as title, descriptions, codes attachment or code snippets.

The result generated will indicate a word that represent the file location or feature location of the fault. The file that contains source code will be modeled and word vector for each file or document will be calculated using cosine similarity, an algorithm of VSM to measure the file or document similarity with the bug report document. The value from vector of words calculated will be rank from highest to less similarity according to this value. Figure 3.5 below shows the cosine similarity algorithm that are being used in this experiment.

Text

Description automatically generated

Figure 3.5: Cosine similarity from VSM text model formula

Usually, IR fault localization are used when there’s a need to understand and modify especially an unfamiliar codebase. When a developer unfamiliar with the large code base of the software system and does not know where to begin. Lacking sufficient documentation on the system and the ability to ask the code’s original authors for help, the only option they sees is to manually search for the code relevant to her task which might consume a lot of time (Dit, et al., 2011).

### 3.3.3 Text Search Engine Experiment

For this experiment, source code and bug report are undergoing the same data preprocessing as IR technique mentioned in 3.3.2 above. Figure 3.6 below shows a framework of Text Search Engine for my research. Initially, the pre-processed bug report title will be used to search matching words in the document collection of source code as the title alone are sufficient, however, if the title is too short or does not contains word that represent the actual issues or problem mentioned, the pre-process descriptions of the bug report will be used. The text search engine are using words string matching algorithm since it work excellently regardless of databases, scripts, and applications (Hakak, et al., 2019). Same as IR technique in this experiment, the fault are considered localized when the file name of the fault appear in top 10 search result.

Diagram

Description automatically generated

Figure 3.6: Text Search Engine framework

### 3.3.4 Experiment procedure

Rao et al believe that combining IR fault localization tools with dynamic fault localization could significantly improve the state-of-the-art in fault localization (Rao, et al., 2011). Experiment started with the individually execution of IR technique, SBFL technique and Text Search Engine.

Each source element's that go through as a preprocess data should be saved as a separate document in the document collection. Each document is saved in .txt files and represents one class that contains several methods of the source code, as the experiments in this article are at the method and class level of granularity.

Each document is modeled into VSM text model, using Cosine Similarity or also known as cosine distance technique. Cosine Similarity measure is computed for unique terms in the documents (Ramya, et al., 2018). Cosine Similarity are computed between all possible pairs of document vectors to measures the similarity between two vectors based on the cosine angle between them (Usino, et al., 2019). For each document computed, it will produce a score ordering of the documents based on the degree to which they contained the same distribution of topics.

Bug report extraction that mimic the document extractor’s pre-processing steps also use cosine similarity technique to find word similarity in bug report documents. Both results from source code and bug report extraction now are in the same format and this will ease the query process. Query will be manually done to generate ranking result of word similarity to know the location of fault.

However there are some situation that might lead to inability to allocate fault accurately. This is because, IR techniques aim to identify a starting point from which correction of the bug can be undertaken. For example, when there is limited relevancy of information in bug reports though all precaution and attributes to a good quality of bug report has been considered into account and still provides us with inconclusive results, this situation may be supplemented with SBFL technique where the suspiciousness score will be look at.

On the other hand, SBFL technique that use source code coverage are executed using Ochiai technique in order to generate suspiciousness result. The suspiciousness results are calculated according to the frequency of the statements in passing and failing test cases. The intuition for this approach to fault localization is that statements in a program that are primarily executed by failed test cases are more likely to be faulty than those that are primarily executed by passed test cases (Jones et al, 2007). However, in some cases SBFL suspiciousness score are inconclusive and fault location cannot be determine, here is the example of situation when the results from IR technique come into consideration to allocate fault. SBFL Output includes program’s Method name, Statement line number, and Suspiciousness score. The result of suspiciousness score will be sorted, and the highest suspiciousness value are likely is the location of the fault. The highest score for suspiciousness value is 1 and the lowest score is 0.

Since both IR and SBFL techniques use different information sources, instead of using one information sources or technique, using both in allocating fault localization can be count as using optimal information that has been provided. Adding one more tool in this fault localization experiment, which is Text Search Engine, as it covers other dimension of analysis since it goes beyond cosine similarity for text matching. Combination of IR and SBFL Technique with Text Search Engine can present more accurate result, as it might cover different analysis that gain from static and dynamic debugging.

The idea of combining more than one technique in this research is so that it can be used as a tie breaker when one of the techniques results unable to allocate fault accurately. Usually, when the fault cannot be identified or in other words, the faults were not in the top 10 of the list results, the developer basically will manually examine the report and source code of the faults as a step to identify the possible location of the faults.

# 4.0 Results and Analysis

This chapter will explain on the results and analysis that has been produced from running the experiment. The purpose of the experiment is to answer the questions that has been raised in the beginning of this study. I have embarked a series of experiments for SBFL technique, IR technique and Text search Engine technique where initially each of it are run independently. After that, all techniques are combined and compared with the previous individual experiment in order to find the relationship and compatibility between techniques, and at the same time to observe fault localization performance. As the experiment begins, the time taken to execute the task also being recorded.

## 4.1 Results and Analysis

### *4.1.1 RQ1: How effective are SBFL, IR and Text Search Engine technique individually / standalone?*

The first research question helps my research to understand the performance of widely used techniques, individually in fault localization. The first experiment results and analysis can be found in Table 4.1 below where it contains result for overall accuracy performance of individually/ stand-alone technique on all 395 faults in Defects4j programs. The boldface indicates the best-performing technique for each program. The result shows that overall performance for individual/stand-alone performance, Text Search Engine technique accuracy surprisingly are the best with 68.4% of accuracy in fault localization for all six real-world Defects4j program. While SBFL technique, the state-of-the-art fault localization technique manage to score 68.1% in accuracy, only a slight difference with Text Search Engine technique. Both Text Search Engine and SBFL techniques accuracy in finding fault localization performed better than IR technique that only score 48.5% for fault localization accuracy.

However, SBFL manage to allocate fault of 269 bugs from the total of 395 bugs compared to Text Search Engine technique where it manages to allocate fault of 255 bugs from the 373 bugs as another 22 bugs does not have a bug report to enable the Search process. This also means that SBFL and Text Search Engine technique unable to localized fault of 126 bugs and 118 bugs respectively.

|  |  |  |  |
| --- | --- | --- | --- |
| Program | SBFL | IR | Search Engine |
| Time | 67% (18/27) | 54% (14/26) | **73% (19/26)** |
| Mockito | **66% (25/38)** | 16% (6/37) | 29% (11/38) |
| Lang | 92% (60/65) | 84.1% (53/63) | **94% (61/64)** |
| Math | 69.8% (74/106) | 55.6% (59/106) | **79% (84/106)** |
| Chart | **85% (22/26)** | 87.5% (7/8) | **100% (8/8)** |
| Closure | 53% (70/133) | 31.3% (41/131) | **55% (72/131)** |
| Overall  Performance | 68.1% (269/395) | 48.5% (180/371) | **68.4% (255/373)** |

Table 4.1: Overall individual/stand-alone performance of SBFL, IR and Text Search Engine technique accuracy.

On the other hand, IR technique manages to localize only 180 bugs from the total 371 bugs where 24 bugs do not have bug report or unable to execute (e.g., Bug report title too short and unable to identified document similarity using IR/VSM) meaning that, IR technique unable to localized fault of 191 bugs from the total of 371 bugs that has been executed. Figure 4.1 below shows the comparison on accuracy of overall individual/stand-alone performance for each program in Defects4j using chart. Overall result shows Text Search Engine technique are accurate than SBFL technique in each real-world java Defects4j programs except for Mockito programs where SBFL perform the best for Mockito with 66% fault localization accuracy compared to only 29% for Text Search Engine technique.

Figure 4.1: Chart for comparison on accuracy of overall individual/stand-alone performance for each program

Text Search Engine techniques manage to score the Time program with 73% accuracy compared to 67% for SBFL, Lang program scores 94% compared to 92% for SBFL, Math program scores 79% compared to 69.8% only for SBFL, Closure programs score 55% compared to 53% accuracy for SBFL. Chart program score 100% accuracy with only 8 bugs involved from 26 bugs as another 18 bugs did not have bug report reference and could not proceed without bug report.

SBFL technique are more accurate than document similarity technique for each real-world java Defects4j programs excepts for Chart programs with 87.5% compared to 85% for SBFL technique. However, as mentioned before Chart programs only managed to run 8 bugs as another 18 from the total of 26 bugs does not have any information regarding bug report/link to proceed with. Time program scores 67% for SBFL technique compared to 54% for document similarity, while Mockito score 66% compared to 16%, lang scores 92% compared to 86%, Math program scores 70.8% compared to 55.6% and closure program scores 53% compared to 48.8%, respectively.

### *4.1.2 RQ2: How effective can we combine the techniques for better fault localization?*

The seconds research question considers way of combining different techniques and evaluate the performance of the combined technique. Another finding that has been analyzed from the combine/hybrid experiment are presented in Table 4.2 below, where it shows the result performance of combination/ hybrid techniques of SBFL, IR and Text Search Engine techniques on all 395 faults in all six real-world program Defects4j programs. The boldface indicates the best-performing hybrid technique for each program from accuracy perspective.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Program | SBFL + IR | SBFL + Search Engine | SBF: + IR + Search Engine | Search Engine + IR |
| Time | (21/27) 78% | (24/27) 89% | **(25/27) 93%** | (23/26) 88.46% |
| Mockito | (25/38) 66% | **(28/38) 74%** | (28/38) 74% | (12/38) 31.6% |
| Lang | (62/65) 95% | (64/65) 98.5% | **(65/65) 100%** | (63/64) 98.4% |
| Math | (82/106) 77.4% | (98/106) 92.5% | **(99/106) 93.4%** | (91/106) 85.8% |
| Chart | (23/26) 88.5% | (**23/26) 88.5%** | (23/26) 88.5% | **(8/8) 100%** |
| Closure | (87/133) 65.4% | (97/133) 73% | **(103/133) 77.4%** | (84/131) 64% |
| Overall performance | (300/395) 75.95% | (334/395) 84.56% | **(343/395) 86.83%** | (281/373) 75.34% |

Table 4.2: Overall performance of combination/ hybrid techniques to all programs

For combination/hybrid technique, the best performance on accuracy is the SBFL technique that combines with IR and Search Engine techniques with 86.83% accuracy on fault localization where 343 bugs are managed to be allocate from 395 bugs. Combination of SBFL and Text Search Engine technique is the second best of fault localization accuracy where it scores 84.56% with 334 bugs are managed to be allocate from the total 395 bugs. The combination of IR with SBFL technique results does not have a substantial difference with the combination of IR with Text Search Engine technique where both scores 75.95% and 75.34% respectively. Figure 4.2 below shows the comparison on accuracy of overall combination technique performance for each program in Defects4j.

Figure 4.2: Comparison on accuracy of overall combination technique performance for each program in Defects4j

As shown in the figure 4.2 above, Time programs managed to score 93% of fault localization if executed using the combination of all three techniques which is SBFL, IR and Text Search Engine. However, using the combination of SBFL and Text Search Engine, combination of IR and Text Search Engine, and combination of SBFL and IR, the Time programs score 89%, 88.46% and 78% for fault localization accuracy, respectively.

While Lang programs manage to allocate faults for all 65 bugs in the program where it successfully scores 100% of fault localization accuracy executed using the combination of SBFL, IR and Text Search Engine. Lang is an example of possibility to allocate all faults in the program when a good text case coverage and a good bug report are combined to achieve 100% fault localization accuracy. For the combination of SBFL with Text Search Engine and IR with Text Search Engine, both technique scores 98.5% and 98.4% of fault localization accuracy respectively. For the combination of SBFL with IR technique, Lang program scores 95% of fault localization accuracy which it is the lowest results scored for Lang program in hybrid/combination technique.

Math programs manage to score 93.4% of fault localization accuracy using the combination of SBFL, IR and Text Search Engine. For combination of SBFL with Search Engine technique, Math program scores 92.5%, while for combination of Search Engine with IR Math program scores 85.8%. The combination of SBFL with IR technique is the lowest scored technique for Math program with 77.4% of fault localization accuracy.

Next is the Closure program where it scores 77.4% of accuracy to localize fault when executed with the combination of SBFL, IR and Text Search Engine techniques. The score for combination of SBFL and Text Search Engine for Closure program is the next best with 73% for fault localization. While the combination of SBFL with IR and Text Search Engine with IR manage to score 65.4% and 64% respectively for accuracy in fault localization.

While for Mockito program, the best accuracy performance is the same for both combination of SBFL with Text Search Engine and combination of SBFL with IR and Text Search Engine where both scores 74% in accuracy to localize fault. The combination of SBFL and IR manage to score 66% of fault localization accuracy while the combination of Text Search Engine and IR technique score 31.6% for fault localization accuracy.

For Chart program, the accuracy result for the combination of IR and SBFL are the same with both the combination of SBFL and Text Search Engine technique, and the combination of SBFL, IR and Text Search Engine where all three technique scores 88.5%. Though the combination of Text Search Engine and IR technique managed to get 100% performance in fault localization accuracy, however, as mentioned before this, there are missing bug report for 18 bugs in that programs so only 8 bugs are included in the experiment.

All these four programs which is Time, Lang, Math and Closure programs perform the best in fault localization when the combination of three technique which are SBFL, IR and Text Search Engine are being used as shown in Table 4.2. However, for Mockito and Chart programs, where it contains more than one combination technique that score highest result in fault localization accuracy, another element will be count in which is the time involved to execute those technique to determine the performance of technique.

### *4.1.3 RQ3: What is the run-time cost of standalone techniques and combined techniques for each program? And how long is the average of execution time for each technique on one bug?*

As the previous research question, where RQ1 and RQ2 concern on the fault localization technique accuracy’s performance, this research question considers the time taken for each stand alone and combined technique as well as the time spent on each bug in fault localization. The best combination technique is the one that have the accuracy and time cost balance. Table 4.3 below shows the comparison on overall total time for SBFL, IR and Text Search Engine technique as well as the time spent for each bug in individual/ stand-alone technique.

The Table 4.3 shows that The Text Search Engine technique is the fastest technique where the overall time taken to execute all six real-world Defects4j programs that contains of 395 bugs in only 37.38 seconds with each bug only need around 0.10 seconds or less to be localized. Time, Mockito, Chart and Closure are the fastest in fault localization when using Text Search Engine technique, where the total time taken to localize fault for each program are 1.5 seconds, 2.49 seconds, 0.49 seconds and 13.93 seconds respectively.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Program | SBFL | | IR | | Search Engine | |
| Total time (s) | Time per bug (s) | Total time (s) | Time per bug (s) | Total time (s) | Time per bug (s) |
| Time | 31. 35 | 1.16 | 74.73 | 2.87 | 1.50 | **0.06** |
| Mockito | 5.26 | 0.14 | 89.42 | 2.42 | 2.49 | **0.07** |
| Lang | 0.73 | **0.01** | 191.66 | 3.04 | 4.11 | 0.06 |
| Math | 4.04 | **0.04** | 258.89 | 2.44 | 14.86 | 0.14 |
| Chart | 2.85 | 0.11 | 21.75 | 2.72 | 0.49 | **0.06** |
| Closure | 643.72 | 4.84 | 405.37 | 3.10 | 13.93 | **0.11** |
| Overall time | 687.95 | 1.74 | 1041.82 | 2.81 | **37.38** | **0.10** |

Table 4.3: Overall individual/stand-alone performance of SBFL, IR and Text Search Engine technique total time spent, and time spent for each bug.

Time program only needs an average of time around 0.06 seconds to localize fault, while 0.07 seconds for Mockito program, 0.06 seconds for Chart program and 0.11 seconds for Closure program. On the other hand, Lang and Math are the fastest with SBFL technique where each program score 0.73 seconds and 4.04 seconds of total execution time with each bug only needs an average of 0.01 seconds and 0.04 seconds respectively, to localize fault.

As addition, SBFL Technique overall time spent to localized fault is 687.95 seconds, while IR technique is the slowest time spent to localize fault as it requires around 1041.82 seconds to complete all six real-world defects4j programs. Figure 4.3 below shows the chart comparison on overall time spent for each bug in individual/stand-alone technique.

Figure 4.3: Comparison on overall time spent for each bug in individual/stand-alone technique

As for combination technique, Table 4.4 below shows the total time taken for each combined technique to be executed and as well as the time spent for each bug for the combined technique.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Program | SBFL + IR | | SBFL + Search Engine | | SBFL + IR + Search Engine | | Search Engine + IR | |
| Total time (s) | Time per bug (s) | Total time (s) | Time per bug (s) | Total time (s) | Time per bug (s) | Total time (s) | Time per bug (s) |
| Time | 106.08 | 3.93 | **32.85** | **1.22** | 107.58 | 3.98 | 76.23 | 2.93 |
| Mockito | 94.68 | 2.49 | **7.75** | **0.20** | 97.17 | 2.56 | 91.91 | 2.42 |
| Lang | 192.39 | 2.96 | **4.84** | **0.07** | 196.5 | 3.02 | 195.77 | 3.06 |
| Math | 262.93 | 2.48 | **18.9** | **0.18** | 277.79 | 2.62 | 273.75 | 2.58 |
| Chart | 24.6 | 0.95 | **3.34** | **0.13** | 25.09 | 0.97 | 22.24 | 2.72 |
| Closure | 1049.09 | 7.89 | 657.65 | 4.94 | 1063.02 | 7.99 | **419.3** | **3.20** |
| Overall time | 1729.77 | 20.7 | **725.33** | **6.74** | 1767.15 | 20.27 | 1079.2 | 16.91 |

Table 4.4: Overall performance of combination/ hybrid techniques to all programs and time spent for each bug

The table 4.4 shows that the combination of SBFL and Text Search Engine performance are the fastest where the time taken to execute all six programs Defects4j is 725.33 seconds, the fastest among all combination technique where it only took 6.74 seconds per bugs to be localized. The time taken to execute all six programs for combination of Text Search Engine and IR technique is 1079.2 seconds with average 16.91 seconds per bugs to be allocate while the combination of IR and SBFL spend 1729.77 seconds with average 20.7 seconds time taken for each bug to be localize. The slowest technique for combination is the combination of SBFL, IR and Search Engine technique that took 1767.15 seconds with average calculation for each bug executed are around 20.27 seconds per bug.

The combination of SBFL and Text Search Engine technique for Time program only took 32.85 seconds with average of 1.22 seconds for each bug to localize fault, Mockito program about 7.75 seconds with average time 0.2 seconds per bugs to localize faults, Lang program took 4.84 seconds with 0.07 seconds for each bug to localize fault, Math program took 18.9 seconds with 0.18 seconds for each bug to localize fault and Chart program took 3.34 seconds with 0.13 seconds for each bug to localize.

For Closure program, the fastest combination technique for this program is the combination of Search Engine and IR technique with total time taken is 419.3 seconds and time taken for each bug to be localized is 3.20 seconds. Figure 4.4 shows the comparison on overall time spent for each bug in combination technique.

Figure 4.4: Comparison on overall time spent for each bug in combination technique

### *4.1.4 RQ4: How effective the combined approach is when compared with the state-of-the-art techniques? Which combination are the best that this research suggest?*

This research question compares the performance of all approach and at the same time to identify which combination’s performance is the best in term of accuracy and time cost. To understand deeper, Venn Diagram of Defects4j program result in fault localization has been created as shown in Figure 4.5 below. ‘E’ represents the total of element and this Venn Diagram, the total number of bugs from six program in Defects4j is 395 bugs. In the Venn Diagram below, three subsets represent each technique that has been used in the experiment, which is SBFL, IR and Text Search Engine technique.

As mentioned in previous research question (RQ1), SBFL technique alone manage to allocate 269 bugs (68.1%), IR technique manage to allocate 180 bugs (48.5%) while Text Search Engine manage to allocate 255 bugs (68.4%) from the total of 395 bugs. However, in RQ2 the combination of SBFL, IR and Text Search Engine manage to allocate fault up to 86.83% or 343 bugs from 395 bugs manage to be localized compared to by only using individual technique. This positive outcome from the combination technique reduced the unlocalized fault to only 52 bugs compared to 126 bugs for SBFL, 118 bugs for Text Search Engine and 191 bugs for IR technique if using individual/stand-alone technique.

E =395

SBFL

Symbol

E = Total element

∩ = Intersection of sets

A’ = Complement/ Not in the set

52

59

17

131

23

IR

Search

Figure 4.5: Venn Diagram of fault localization using SBFL, IR and Search Engine technique in Defetcs4j program.

Table 4.5 shows the summary of individual and hybrid/ combination technique results. The results include the accuracy score for each technique on all Defects4j program and as well as the time spent on each bug in fault localization. Since the main reason of fault localization is to identified fault that caused the error. It is important to measure the accuracy first. However, there are cases that involved similarity of accuracy results, thus at this point, the time taken in fault localization process can be used to identify the best combined technique. This will be discussed further in conclusion chapter.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Program | SINGLE  TOOLS | SBFL | | IR | | Search Engine | | HYBRID  TOOLS | SBFL + IR | | SBFL + Text Search Engine | | SBFL + Search Engine + IR | | Search Engine + IR | |
| Acc (%) | T (s) | Acc (%) | T (s) | Acc (%) | T (s) | Acc (%) | T (s) | Acc (%) | T (s) | Acc (%) | T (s) | Acc (%) | T(s) |
| Time | 67 | 1.16 | 54 | 2.87 | 73 | 0.06 | 78 | 3.93 | 89 | 1.22 | 93 | 3.98 | 88.46 | 2.93 |
| Mockito | 66 | 0.14 | 16 | 2.42 | 29 | 0.07 | 66 | 2.49 | 74 | 0.20 | 74 | 2.56 | 31.6 | 2.42 |
| Lang | 92 | 0.01 | 86 | 3.04 | 94 | 0.06 | 95 | 2.96 | 98.5 | 0.07 | 100 | 3.02 | 98.4 | 3.06 |
| Math | 70.8 | 0.04 | 55.6 | 2.44 | 79 | 0.14 | 77.4 | 2.48 | 92.5 | 0.18 | 94.3 | 2.62 | 85.8 | 2.58 |
| Chart | 85 | 0.11 | 87.5 | 2.72 | 100 | 0.06 | 88.5 | 0.95 | 88.5 | 0.13 | 88.5 | 0.97 | 100 | 2.72 |
| Closure | 53 | 4.84 | 31.3 | 3.10 | 55 | 0.11 | 65.4 | 7.89 | 73 | 4.94 | 78.2 | 7.99 | 64 | 3.20 |
| Overall performance | 68.4 | 1.74 | 48.8 | 2.81 | 68.4 | 0.10 | 75.95 | 20.7 | 84.56 | 6.74 | 86.83 | 20.27 | 75.34 | 16.91 |

Table 4.5 above shows the summary of individual and hybrid techniques results.

## 4.3 Limitation

Unfortunately, due to time constraints only three techniques from three different family are considered and due to limited tools access, the experiment has been designed and built from scratch by using the understanding of architecture of the tools from other research paper hence the experiment is being done manually. Compared to other technique in IR such as bug histories, stack traces and structure, the availability and accessibility of bug report for Defects4j driven this research choose SVM or document similarity of IR technique.

I also want to highlight the important of quality bug report where there are cases that fault location unable to be identified as the information provided are not enough. Though the effectiveness of the technique especially for IR and Text Search Engine technique is likely to depend heavily on the quality of the bug reports, (Wang et al, 2015) unfortunately, high-quality bug reports that contain important information are not always available.

# 5.0 Conclusion and Future works

## 5.1 Summary

In this research, I analyzed 395 real world defects from Defects4J programs for fault localization using individually/standalone technique and combination/hybrid technique. Based on the experiment taken, the result shows that the different and independent data from both techniques that has been combined does complement to each other as it produced a new expression of data from both perspectives in fault localization which is dynamic and static analysis of fault localization technique.

As a conclusion from the experiment results, though usually time spent for standalone/ individual technique are slightly shorter than the combined one, the combination or hybrid technique effectiveness in fault localization significantly outperforms standalone techniques. According to (Jiang, et al., 2019), the amount of execution time is small if it is less than three minutes on average and this denotes that the time taken does not heavily damage the accuracy of fault localization approaches. However, even though the combination techniques takes longer time compared to individual technique, when it comes to critical setting in certain area, time does not important as long as the fault are managed to be localized.

I hope that this research might be complementary to existing techniques as it could further improve state-of-the-art technique by combining with existing techniques. Fault localization techniques have always been utilized and evaluated individually. To compare between individual and combined technique, of course time taken for combination technique are higher than individual fault localization technique. This research demonstrates how simple and easy it is to combine even the most disparate fault localization approaches. This research recommend that developer should mix numerous strategies within a time restriction rather than using one technique alone. This implies that, it is more important to understand how the technique contributes to combine with existing techniques, rather than understanding the performance of the technique in isolation.

This also conclude that the proposed hybrid technique is reliable where it does assist the location of fault. However, if the bug report does not meet the quality/requirement that has highlighted in section 3 the result does not assist developer in allocating fault. Our results shows that there is no single technique that can be effective in allocating faults however, it is recommended to use multiple technique as a strategy in fault localization as it improved performance. The main contributions of this thesis can be identified as the followings:

* A novel empirical study that compares a specific range of fault localization techniques on real faults which is between SBFL, IR and Text Search Engine technique.
* Observation on relationship between SBFL, IR and Text Search Engine technique behavior by using a real-world java Defects4j dataset.
* Proposed a combined technique that are configurable based on time spent, and the accuracy to localize faults performance.
* An infrastructure/architecture/model for evaluating and combining fault localization techniques for future research.

## 5.2 Future work and outlook

I intend to replicate this study with additional collections of subject programs, and since Defects4J contains only single-fault versions of programs, so a study involving programs with multiple real faults is needed. These research findings also call for future work on the automation of the combination technique, leading to better fault localization techniques.

# Bibliography

**Myers G.J.** The Art of Software Testing, Second Edition [Book]. - New York : Wiley, 2004. - p. 234.

**Elsaka E.** Fault Localization Using Hybrid Static/Dynamic Analysis [Book]. - University of Maryland, US : Elsevier, 2017.

**Sohn J and Yoo S** Fluccs: using code and change metrics to im- prove fault localization [Book]. - [s.l.] : in Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 2017. - pp. 273–283.

**Li X and Zhang L** Transforming programs and tests in tandem for fault localization [Book]. - [s.l.] : Proceedings of the ACM on Programming Languages, vol. 1, 2017. - p. 92.

**Le T B [et al.]** A learning-to- rank based fault localization approach using likely invariants [Book]. - [s.l.] : in Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM,, 2016. - pp. 177-188.

**Just R, Darioush J and Ernst M D** Defects4J: a database of existing faults to enable controlled testing studies for Java programs [Book]. - [s.l.] : In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA), 2014. - pp. 437-440.

**fitzgerald Sue [et al.]** Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers [Journal] // Computer Science Education. - 2008. - Issue 2: Debugging by Novice Programmers : Vol. 18. - pp. 93-116.

**Ko J and Myers A** Designing the whyline: A debugging interface for asking questions about program behavior. [Journal] // SIGCHI Conference on human factors in computing systems, CHI. - New York, NY, USA : ACM, 2004. - pp. 151-158.

**Zayour I and Hamdar A** A qualitative study on debugging under an enterprise IDE. [Journal] // Information and software technology. - 2016. - pp. 130-139.

**Layman M** Information Needs of Developers for Program Comprehension During Software Maintenance Tasks [Journal] // PhD Thesis. - North Carolina State University : [s.n.], 2009.

**Ko J A [et al.]** An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. [Journal] // IEEE Transactions on Software Engineering. - 2006. - 12 : Vol. 32. - pp. 971-987.

**Korel B and Laski J** Dynamic program slicing [Journal] // Information Processing Letters. - 1988. - 3 : Vol. 29. - pp. 155-163.

**Jones A J, Harrold J M and Stasko T J** Visualization of test information to assist fault localization [Journal] // ICSE. - 24th International Conference on Software Engineering : ACM, 2002. - pp. 467-477.

**Liblit B [et al.]** Scalable statistical bug isolation [Journal] // PLDI. - New York, NY, USA : ACM, 2005. - pp. 15-26.

**Liu C [et al.]** Sober: Statistical model- based bug localization. [Journal] // SIGSOFT, Softw. Eng. Notes. - 2005. - 5 : Vol. 30. - pp. 286-295.

**Parnin C and Orso A** Are automated debugging techniques actually helping programmers? [Journal] // In Proceedings of the 2011 International Symposium on Software Testing and Analysis, ISSTA ’11. - New York, NY, USA : ACM, 2011. - pp. 199-209.

**Renieres M and Reiss P S** Fault localization with nearest neighbor queries [Journal] // In 18th IEEE International Conference on Automated Software Engineering. - 2003. - pp. 30-39.

**Shu T [et al.]** Fault localization based on statement frequency [Journal] // Information Sciences. - 2016. - pp. 43-56.

**Sohn J and Yoo S** Fluccs: Using code and change metrics to improve fault localization [Journal] // In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA. - New York, NY, USA : ACM, 2017. - pp. 273-283.

**Weiser M** Program slicing [Journal] // In Proceedings of the 5th International Conference on Software Engineering. - San Diego, California, USA : [s.n.], 9-12 March, 1981. - pp. 439–449.

**Wong E W and Qi Y** Bp neural network-based effective fault localization [Journal] // International Journal of Software Engineering and Knowledge Engineering. - 2009. - 4 : Vol. 19. - pp. 573– 597.

**Cleve H and Zeller A** Finding failure causes through automated testing [Journal] // Proceedings of the Fourth International Workshop on Automated Debugging. - Munich, Germany : [s.n.], 28-30th August, 2000.

**Lukins Stacy K, Kraft Nicholas A and Etzkorn H Letha** Source Code Retrieval for Bug Localization using Latent Dirichlet Allocation [Journal] // 15th Working Conference on Reverse Engineering. - [s.l.] : IEEE, 2008. - pp. 155-164.

**Abreu R, Zoeteweij p and Gemund Van** On the accuracy of spectrum- based fault localization [Journal] // In Testing: Academic and Industrial Conference Practice and Research Techniques. - 2007. - pp. 89–98.

**Zou Daming [et al.]** An Empirical Study of Fault Localization Families and Their Combinations [Journal]. - 2019.

**Jones J A and Harrold M J** Empirical evaluation of the tarantula automatic fault-localization technique. [Journal] // International Conference on Automated Software Engineering (ASE 2005). - Long Beach, CA, USA : IEEE/ACM , 2005. - pp. 273–282.

**Sun C and Huang T** Using mutation in fault localization [Journal] // Computer Science Information Technology. - Institute of Software, Chinese Academy of Sciences, Beijing, China : [s.n.], 2016. - pp. 59–66.

**Agarwal Pragya and Agrawal Arun Prakash** Fault-localization techniques for software systems: a literature review [Journal] // ACM SIGSOFT Software Engineering Notes. - 2014. - 5 : Vol. 39. - pp. 1–5.

**Rao S and Kak A** Retrieval from software libraries for bug localization: a comparative study of generic and composite text models [Journal] // In Proceeding of the 8th working conference on Mining software repositories. - Honolulu, Hawaii : ACM, 2011. - pp. 43-52.

**Zhou Jian, Zhang Hongyu and Lo David** Where Should the Bugs Be Fixed? [Journal] // ICSE. - Zurich, Switzerland : IEEE, 2012. - pp. 14-24.

**Saha Ripon K [et al.]** Improving Bug Localization using Structured Information Retrieval [Journal] // ASE. - Palo Alto, USA : IEEE, 2013. - pp. 345-355.

**Le Tien-Duy B, Lo David and Thung Ferdian** Should I follow this fault localization tool's output? Automated prediction of fault localization effectiveness [Journal] // Empirical Software Engineering. - 2015. - 5 : Vol. 20. - pp. 1237-1274.

**De Souza Higor A [et al.]** Contextualizing spectrum-based fault localization [Journal] // ScienceDirect. - 2017. - pp. 245–261.

**Kim Sunghun [et al.]** Predicting Faults from Cached History [Journal] // IEEE. - [s.l.] : ICSE, 2007.

**Lillis David and Scanlon Mark** On the Benefits of Information Retrieval and Information Extraction Techniques Applied to Digital Forensics [Journal]. - 2016.

**Wong Eric W [et al.]** A Survey on Software Fault Localization [Journal] // IEEE. - 2016.

**Blei David M, Ng Andrew Y and Jordan Michael I** Latent Dirichlet Allocation [Journal] // Journal of Machine Learning Research. - 2003. - pp. 993-1022.

**Chen Tse-Hsun, Thomas Stephan W and Hassan Ahmed E** A Survey on the Use of Topic Models when Mining Software Repositories [Journal] // Empirical Software Engineering. - 2015.

**McCall Davin and Kölling Michael** Meaningful Categorisation of Novice Programmer Errors [Journal]. - Madrid, Spain : IEEE, 22-25 Oct, 2014.

**Munson John C, Nikora Allen P and Sherif Joseph S** Software faults: A quantifiable definition [Journal] // Advances in Engineering Software. - [s.l.] : Elsevier, 2006.

**Krawiec Magdalena** Terminological discrepancies in the field of software testing: A case of mistake, error, bug, defect, fault, and failure in the specialist language of IT [Journal]. - 2018. - pp. 71-81.

**Hristova Maria [et al.]** Identifying and correcting Java programming errors for introductory computer science students [Journal] // SIGCSE. - Nevada, USA : ACM, 19-23 Febuary, 2003.

**Allen Eric** Bug Patterns in Java [Journal]. - 2002. - pp. 7-24.

**Ko Andrew J and Myers Brad A** Aframeworkandmethodologyforstudyingthe causes of software errors in programming systems [Journal] // Journal of Visual Languages and Computing. - Forbes Ave., Pittsburgh, PA 15213, USA : Elsevier, 2005. - pp. 41–84.

**Wong E W and Debroy V** Software fault localization [Journal] // Encyclopedia of Software Engineering. - [s.l.] : Taylor & Francis, 2010. - pp. 1147–1156.

**Lucia [et al.]** Comprehensive Evaluation of Association Measures for Fault Localization [Journal] // International Conference on Software Maintenance. - Timisoara, Romania : IEEE, 2010.

**Lucia [et al.]** Extended comprehensive study of association measures for fault localization [Journal] // Journal of Software: Evolution and Process. - [s.l.] : John Wiley & Sons, Ltd, 2014. - pp. 172–219.

**Cleve Holger and Zeller Andreas** Locating Causes of Program Failures [Journal] // ICSE. - St. Louis, Missouri, USA : ACM, 15-21 May, 2005.

**Xie Xiaoyuan, Chen Tsong Yueh and Kuo Fei-Ching** A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-Based Fault Localization [Journal]. - [s.l.] : Trans. Softw. Eng. Methodol, 2013. - 4 : Vol. 22. - p. 40.

**Xuan Jifeng and Monperrus Martin** Learning to Combine Multiple Ranking Metrics for Fault Localization [Journal] // International Conference on Software Maintenance and Evolution. - [s.l.] : IEEE, 2014. - pp. 191-200.

**Pearson Spencer [et al.]** Evaluating and improving fault localization [Journal]. - 2017.

**Moon Seokhyeon [et al.]** Ask the Mutants: Mutating Faulty Programs for Fault Localization [Journal] // IEEE. - [s.l.] : International Conference on Software Testing, Verification and Validation, 2014. - pp. 153-162.

**Ali Shaimaa [et al.]** Evaluating the Accuracy of Fault Localization Techniques [Journal] // IEEE. - [s.l.] : International Conference on Automated Software engineering, 2009. - pp. 76-87.

**Wong Chu-Pan [et al.]** Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis [Journal]. - 2014.

**Moreno Laura [et al.]** On The Use of Stack Traces to Improve Text Retrieval-based Bug localization [Journal] // International Conference on Software Maintenance and Evolution. - [s.l.] : IEEE, 2014. - pp. 151-160.

**Zhang Xiangyu, Gupta Neelam and Gupta Rajiv** Locating Faults Through Automated Predicate Switching [Journal] // ICSE. - Shanghai, China : ACM, 20-28 May, 2006.

**Wen Ming, Wu Rongxin and Cheung Shing-Chi** Locus: Locating Bugs from Software Changes [Journal] // ASE'16. - Singapore : ACM, 3-7 September, 2016. - pp. 262-273.

**Zhang Xiao-Yi [et al.]** Using Partition Information to prioritize test cases for Fault Localization [Journal] // Annual International Computers, Software, & Applications Conference. - [s.l.] : IEEE, 2015. - pp. 121-126.

**Zhong Hao and Mei Hong** Learning a graph-based classifier for fault localization [Journal] // Science China - Information Sciences. - [s.l.] : Springer, 2020.

**Lukins Stacy K, Kraft Nicholas A and Etzkorn Letha H** Bug localization using latent Dirichlet allocation [Journal] // Information and Software Technology. - [s.l.] : Elsevier, 2010. - pp. 972-990.

**Le Tien-Duy B and Lo David** Will Fault Localization Work For These Failures ? An Automated Approach to Predict Effectiveness of Fault Localization Tools [Journal] // International Conference on Software Maintenance. - [s.l.] : IEEE, 2013. - pp. 310-319.

**Wang Shaowei and Lo David** Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization [Journal] // ICPC. - Hyderabad, India : ACM, 2-3 June, 2014. - pp. 53-63.

**Ye Xin, Bunescu Razvan and Liu Chang** Learning to Rank Relevant Files for Bug Reports using Domain Knowledge [Journal] // FSE. - Hong Kong, China : ACM, 16-21 November, 2014. - pp. 689-699.

**Salton G, Wong A and Yang C S** A vector space model for automatic indexing [Journal]. - [s.l.] : ACM, November, 1975. - 11 : Vol. 18. - pp. 613-620.

**Sisman Bunyamin and Kak Avinash C** Incorporating Version Histories in Information Retrieval Based Bug Localization [Journal]. - Zurich, Switzerland : IEEE, 2012.

**Dallmeier Valentin and Zimmermann Thomas** Extraction of Bug Localization Benchmarks from History [Journal] // ASE. - Atlanta, Georgia,USA : ACM, 4-9 November, 2007.

**Wang Shaowei and Lo David** AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization [Journal] // Journal of Software Evolution and Process. - [s.l.] : Wiley Online Library, 10 October, 2016. - pp. 921-942.

**Wen Ming [et al.]** Historical Spectrum based Fault Localization [Journal]. - [s.l.] : IEEE, 2018.

**Hoang Thong [et al.]** Network-Clustered Multi-Modal Bug Localization [Journal] // IEEE Transactions on Software Engineering. - [s.l.] : IEEE, October, 2019. - 10 : Vol. 45.

**Jiang Jiajun [et al.]** Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study [Journal] // International Conference on Automated Software Engineering (ASE). - [s.l.] : IEEE/ACM, 2019. - pp. 502-514.

**Cui Zhanqi [et al.]** Improving Software Fault Localization by Combining Spectrum and Mutation [Journal]. - [s.l.] : IEEE, 2020. - Vol. 8.

**Poshyvanyk Denys and Marcus Andrian** Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code [Journal]. - 2007.

**IEEE** IEEE Standard for Software and System Test Documentation [Journal]. - [s.l.] : IEEE Standard 829-2008, 2008.

**IEEE** IEEE Standard Glossary of Software Engineering Terminology [Journal]. - [s.l.] : IEEE Standard 610.12, 1990.

**Hoffman Matthew D, Blei David M and Bach Francis** Online Learning for Latent Dirichlet Allocation [Journal]. - 2010.

**Koyuncu Anil [et al.]** D&C: A Divide-and-Conquer Approach to IR-based Bug Localization [Journal]. - [s.l.] : IEEE, 2019.

**Gay Gregory and Just Rene** Defects4J as a Challenge Case for the Search-Based Software Engineering Community [Journal]. - 2020.

**Poshyvanyk Denys [et al.]** Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval [Journal] // IEEE Transactions on software engineering. - [s.l.] : IEEE, 2007. - 6 : Vol. 33. - pp. 420-432.

**Wang Shaowei, Lo David and Lawall Julia** Compositional Vector Space Models for Improved Bug Localization [Journal]. - [s.l.] : IEEE, 2014. - pp. 171-180.

**Harrold M J [et al.]** An empir- ical investigation of the relationship between spectra differences and regression faults [Journal]. - [s.l.] : Software Testing Verification and Reliability, 2000. - 3 : Vol. 10. - pp. 171-194.

**Collofello J S and Cousins L** Towards automatic software fault location through decision-to-decision path analysis [Journal] // National Computer Conference. - [s.l.] : IEEE, 1986. - p. 539.

**Pan K, Kim S and Whitehead E J** Toward an understanding of bug fix patterns [Journal]. - [s.l.] : Empirical Softw. Engg, Jun, 2009. - 3 : Vol. 14. - pp. 286-315.

**Vancsics Béla, Szatmári Attila and Beszédes Árpád** Relationship between the Effectiveness of Spectrum-Based Fault Localization and Bug-Fix Types in JavaScript Programs [Journal] // SANER. - London, ON, Canada : IEEE, 2020. - pp. 308-319.

**Denmat Tristan, Ducasse ́ Mireille and Ridoux Olivier** Data Mining and Cross-checking of Execution Traces [Journal]. - Long beach, California : ASE, 2005.

**Abreu Rui, Zoeteweij Peter and Gemund Arjan J.C. van** An Evaluation of Similarity Coefficients for Software Fault Localization [Journal] // Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing. - [s.l.] : IEEE, 2006. - Vol. 12. - pp. 39-46.

**Baah George K, Podgurski Andy and Harrold Mary Jean** Causal Inference for Statistical Fault Localization [Journal] // Proceedings of the 19th International Symposium on Software Testing and Analysis. - Trento, Italy : ISSTA’10, 12-16 July, 2010. - pp. 73-83.

**Akbar Shayan A** SOURCE CODE SEARCH FOR AUTOMATIC BUG LOCALIZATION [Journal]. - [s.l.] : Thesis dissertation, 2020.

**Marcus Andrian [et al.]** An Information Retrieval Approach to Concept Location in Source Code [Journal] // 11th Working Conference on Reverse Engineering. - [s.l.] : IEEE, Nov, 2004. - pp. 214–223.

**Hovemeyer David and Pugh William** Finding Bugs is Easy [Journal] // OOPSLA’04. - Vancouver, British Columbia, Canada. : ACM, 24-28 OCT, 2004.

**Kuhn A, Ducasse S and Grba T** Semantic clustering: Identifying topics in source code [Journal] // 12th Working Conference on Reverse Engineering. - [s.l.] : Information and Software Technology , 2007. - 3 : Vol. 49. - pp. 230 – 243.

**Sisman B and Kak C** Assisting code search with automatic query refor- mulation for bug localization [Journal] // in Proceedings of the 10th Working Conference on Mining Software Repositories. - [s.l.] : IEEE Press, 2013. - pp. 309–318.

**Moreno L [et al.]** Query-based configuration of text retrieval solutions for software engineering tasks [Journal] // in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. - [s.l.] : ACM, 2015. - pp. 567–578.

**Rahman M M and Roy C K** Improving ir-based bug localization with context-aware query reformulation [Journal] // in Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. - [s.l.] : ACM, 2018. - pp. 621–632.

**Akbar Shayan A and Kak Avinash C** Scor: Source code retrieval with semantics and order [Journal] // Proceedings of the 16th International Conference on Mining Software Repositories, ser. MSR ’19. - Piscataway, NJ, USA : IEEE Press, 2019. - pp. 1-12.

**Xiao Yan [et al.]** Improving bug localization with word embedding and enhanced T convolutional neural networks [Journal]. - [s.l.] : Elsevier, 2018.

**Lam An Ngoc [et al.]** Bug Localization with Combination of Deep Learning and Information Retrieval [Journal] // 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC). - [s.l.] : IEEE/ACM, 2017. - pp. 218–229.

**Nguyen Thanh Van [et al.]** Combining Word2Vec with Revised Vector Space Model for Better Code Retrieval [Journal] // 2017 IEEE/ACM 39th IEEE International Conference on Software Engineering Companion (ICSE-C). - [s.l.] : IEEE/ACM, May, 2017. - pp. 183–185.

**Khvorov Aleksandr, Vasiliev Roman and Chernishev George** S3M: Siamese Stack (Trace) Similarity Measure [Journal]. - 18 March, 2021.

**Zeller Andreas and Hildebrandt Hildebrandt** Simplifying and isolating failure-inducing input [Journal] // IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. - [s.l.] : IEEE, February, 2002. - 2 : Vol. 28. - pp. 183–200.

**Masri Wes** Automated Fault Localization: Advances and Challenges [Journal] // Advances in Computers. - [s.l.] : Elsevier Inc, 2015. - Vol. 99. - pp. 103-156.

**Zeller Andreas** Isolating cause-effect chains from computer programs [Journal] // Symposium on Foundations of Software Engineering (FSE). - 2002. - pp. 1-10.

**Zimmermann Thomas and Zeller Andreas** Visualizing Memory Graphs [Journal] // Proceedings of the Inter- national Seminar on Software Visualization. - Dagstuhl Castle, Germany : [s.n.], 2001. - pp. 191–204.

**Burger Martin and Zeller Andreas** Minimizing Reproduction of Software Failures [Journal] // Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA’11). - Torrontto,,ON,,Canad : ACM, 17-21 July, 2011. - pp. 221-231.

**Ma L [et al.]** GRT: An Automated Test Generator Using Orchestrated Program Analysis [Journal] // International Conference on Automated Software Engineering (ASE). - [s.l.] : IEEE/ACM, 2015. - pp. 842-847.

**Ledel Benjamin and Herbold Steffen** Broccoli: Bug localization with the help of text search engines [Journal] // 2021 Association for Computing Machinery. - October, 2021. - 1 : Vol. 1.

**Ngo Vuong M, Cao Tru H and Le Tuan M.V** Combining Named Entities with WordNet and Using Query-Oriented Spreading Activation for Semantic Text Search [Journal] // 2010 IEEE RIVF International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future (RIVF). . - [s.l.] : IEEE , 2010.

**Hakak Saqib Iqbal [et al.]** Exact String Matching Algorithms: Survey, Issues, and Future Research Directions [Journal]. - [s.l.] : IEEE, 2019. - Vol. 7.

**Kabir Muhammad N, Alginahi Yasser M and Tayan Omar** Efficient Search of a sequence of words in a Large Text File [Journal]. - [s.l.] : IEEE, 2014.

**Biggers Lauren R [et al.]** Configuring latent Dirichlet allocation based feature location [Journal] // Empir Software Engineering. - 2014. - Vol. 19. - pp. 465-500.

**Abebe Surafel Lemma [et al.]** Lexicon Bad Smells in Software [Journal] // 2009 16th Working Conference on Reverse Engineering. - [s.l.] : IEEE Computer Science, 2009.

**Kim Dongsun [et al.]** Where Should We Fix This Bug? A Two-Phase Recommendation Model [Journal] // IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. - [s.l.] : IEEE Computer Society, November, 2013. - 11 : Vol. 39.

**Dit Bogdan [et al.]** Configuring Topic Models for Software Engineering Tasks in TraceLab [Journal]. - 2013.

**Dit Boghan [et al.]** Feature Location in Source Code: A Taxonomy and Survey [Journal]. - [s.l.] : CRC to Journal of Software Maintenance and Evolution: Research and Practice, 2011.

**Zimmermann Thomas [et al.]** What Makes a Good Bug Report? [Journal] // IEEE TRANSACTIONS ON SOFTWARE ENGINEERING. - [s.l.] : IEEE, September/October, 2010. - 5 : Vol. 36. - pp. 618-643.

**Tantithamthavorn Chakkrit [et al.]** The Impact of IR-based Classifier Configuration on the Performance and the Effort of Method-Level Bug Localization [Journal]. - April, 2018.

**Tantithamthavorn Chakkrit [et al.]** Impact Analysis of Granularity Levels on Feature Location Technique [Journal]. - 2014. - pp. 135-149.

**Ramya S R [et al.]** DRDLC: Discovering Relevant Documents Using Latent Dirichlet Allocation and Cosine Similarity [Journal] // ICNCC 2018. - Taipei City, Taiwan : ACM, 14-16 December, 2018. - pp. 87-91.

**Usino Wendi [et al.]** Document Similarity Detection using K-Means and Cosine Distance [Journal] // (IJACSA) International Journal of Advanced Computer Science and Applications. - 2019. - 2 : Vol. 10. - pp. 165-170.

# Appendix

Result summary for Defects4j program:

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Program** | **SBFL** | | **IR (CS)** | | **SBFL + IR(CS)** | | **IR (Query)** | | **SBFL +IR (Query)** | | **SBFL+IR (CS+Query)** | |
| **E** | **Time** | **E** | **Time** | **E** | **Time** | **E** | **Time** | **E** | **Time** | **E** | **Time** |
| **Time** | 67% | 31. 35s | 54% | 74.73s | 78% | 106.08s | 73% | 1.50s | 89% | 32.85s | 93% | 107.58s |
| **Mockito** | 66% | 5.26 s | 16% | 89.42s | 66% | 94.68s | 29% | 2.49s | 74% | 7.75s | 74% | 97.17s |
| **Lang** | 92% | 0.73 s | 86% | 191.66s | 95% | 192.39s | 94% | 4.11s | 98.5% | 4.84s | 100% | 196.5s |
| **Math** | 70.8% | 4.04s | 55.6% | 258.89s | 77.4% | 262.93s | 79% | 14.86s | 92.5% | 18.9s | 94.3% | 277.79s |
| **Chart** | 85% | 2.85 s | 87.5% | 21.75s | 88.5% | 24.6s | 100% | 0.49 s | 88.5% | 3.34s | 88.5% | 25.09s |
| **Closure** | 53% | 643.72s | 31.3% | 405.37s | 65.4% | 1049.09s | 55% | 13.93s | 73% | 657.65s | 78.2% | 1063.02s |

Result summary for SBFL + IR (Query):

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Program | SBFL | | IR (Query) | | SBFL top 10 +IR (Query) | | SBFL top 30 + IR (Query) | | SBFL top 100 + IR (Query) | |
| E | Time | E | Time | E | Time | E | Time | E | Time |
| Time | 67% | 31. 35s | 73% | 1.50s | 89% | 32.85s | 100% | 32.85 | - | - |
| Mockito | 66% | 5.26 s | 29% | 2.49s | 74% | 7.75s | 86.8% | 7.75s | 100% | 7.75s |
| Lang | 92% | 0.73 s | 94% | 4.11s | 98.5% | 4.84s | - | 4.84s | 100% | 4.84s |
| Math | 70.8% | 4.04s | 79% | 14.86s | 92.5% | 18.9s | 96.2% | 18.9s | 99% | 18.9s |
| Chart | 85% | 2.85 s | 100% | 0.49 s | 88.5% | 3.34s | 92.3% | 3.34s | - |  |
| Closure | 53% | 643.72s | 55% | 13.93s | 73% | 657.65s | 75.2% | 657.65s | 85.7% | 657.65s |

\*E = Effectiveness

\*CS = Cosine Similarity

\*IR = Information Retrieval

\*SBFL = Spectrum Based Fault Localization

\*T = Title

\*C = Combination (Title and Description)

\*BR = Bug report

Top 10 Result analysis:

## Time

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bug | SBFL | IR (CS) | SBFL + IR(CS) | IR (Query) | SBFL +  IR (Query) | SBFL+IR (CS+Query) | IR (CS+Query) |
| 1 | ✓ | ✓(C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✓ |
| 2 | ✓ | ✓ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✓ |
| 3 | ✓ | ✕ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 4 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 5 | ✕ | ✓ (C) | ✓ | ✕ (T,C) | ✕ | ✓ | ✓ |
| 6 | ✓ | ✓ (C) | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 7 | ✓ | ✕ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 8 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 9 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 10 | ✕ | ✕ (C) | ✕✕ | ✓ (C) | ✓ | ✓ | ✓ |
| 11 | ✕ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 12 | ✓ | ✓ (T) | ✓ | ✕ (T) | ✓ | ✓ | ✓ |
| 13 | ✕ | ✕ (C) | ✕✕ | ✕ (T,C) | ✕✕ | ✕✕ | ✕ |
| 14 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 15 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 16 | ✓ | ✕ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 17 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 18 | ✓ | ✕ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 19 | ✕ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 20 | ✓ | ✕ (C) | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 21 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 22 | ✕ | ✕ (C) | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 23 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 24 | ✕ | ✕ (C) | ✕✕ | ✕ (T,C) | ✕✕ | ✕✕ | ✕ |
| 25 | ✕ | ✕ (C) | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 26 | ✕ | ✕ (C) | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 27 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| **Result** | **67%**  **(18/27)** | **54%**  **(14/26)** | **78%**  **(21/27)** | **73%**  **(19/26)** | **89%**  **(24/27)** | **93%**  **(25/27)** | **88.46%**  **(23/26)** |
| **Time**  **(seconds)** | **31. 35s** | **74.73s** | **106.08s** | **1.50s** | **32.85s** | **107.58s** | **76.23s** |

## Lang

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bug | SBFL | IR (CS) | SBFL + IR(CS) | IR (Query) | SBFL +  IR (Query) | SBFL+IR (CS+Query) | IR (CS+Query) |
| 1 | ✓ | ✓(C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 2 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 3 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 4 | ✓ | ✓ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✓ |
| 5 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 6 | ✕ | ✕ (C) | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 7 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 8 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 9 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 10 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 11 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 12 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 13 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 14 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 15 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 16 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 17 | ✕ | ✓ (C) | ✓ | ✕ (T,C) | ✕ | ✓ | ✓ |
| 18 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 19 | ✓ | ✕ (C) | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 20 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 21 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 22 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 23 | ✕ | ✕ (C) | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 24 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 25 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 26 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 27 | ✕ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 28 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 29 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 30 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 31 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 32 | ✓ | ✕ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 33 | ✓ | ✓ (T) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 34 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 35 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 36 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 37 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 38 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 39 | ✓ | ✕ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 40 | ✓ | ✕ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 41 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 42 | ✓ | ✕ (C) | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 43 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 44 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 45 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 46 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 47 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 48 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 49 | ✓ | Title too short | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 50 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 51 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 52 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 53 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 54 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 55 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 56 | ✕ | ✕ (C) | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 57 | ✓ | ✕ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 58 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 59 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 60 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 61 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 62 | ✓ | ✓ (C) | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 63 | ✓ | ✓ (C) | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 64 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 65 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| **Result** | **92%**  **(60/65)** | **84.1%**  **(53/63)** | **95%**  **(62/65)** | **94%**  **(61/64)** | **98.5%**  **(64/65)** | **100%**  **(65/65)** | **98.4%**  **(63/64)** |
| **Time**  **(seconds)** | **0.73 s** | **191.66s** | **192.39s** | **4.11s** | **4.84s** | **196.50s** | **195.77s** |

## Mockito

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bug | SBFL | IR (CS) | SBFL + IR(CS) | IR (Query) | SBFL +  IR (Query) | SBFL+IR (CS+Query) | IR (CS+Query) |
| 1 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 2 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 3 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 4 | ✓ | ✕ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 5 | ✕ | ✕ (C) | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 6 | ✓ | ✓ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✓ |
| 7 | ✓ | ✕ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 8 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 9 | ✕ | ✕ (C) | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 10 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 11 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 12 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 13 | ✕ | ✕ (C) | ✕✕ | ✕ (T,C) | ✕ | ✕ | ✕ |
| 14 | ✕ | ✕ (C) | ✕✕ | ✕ (T,C) | ✕ | ✕ | ✕ |
| 15 | ✕ | ✕ (C) | ✕✕ | ✕ (T,C) | ✕ | ✕ | ✕ |
| 16 | ✕ | ✕ (T) | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 17 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 18 | ✓ | ✕ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 19 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 20 | ✕ | ✕ (C) | ✕✕ | ✕ (T,C) | ✕ | ✕ | ✕ |
| 21 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 22 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 23 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 24 | ✓ | Title too short | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 25 | ✕ | ✕ (C) | ✕✕ | ✕ (T,C) | ✕ | ✕ | ✕ |
| 26 | ✕ | ✕ (C) | ✕✕ | ✕ (T,C) | ✕ | ✕ | ✕ |
| 27 | ✕ | ✕ (C) | ✕✕ | ✕ (T,C) | ✕ | ✕ | ✕ |
| 28 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 29 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 30 | ✓ | ✓ (C) | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 31 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 32 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 33 | ✕ | ✕ (C) | ✕✕ | ✕ (T,C) | ✕ | ✕ | ✕ |
| 34 | ✕ | ✕ (C) | ✕✕ | ✕ (T,C) | ✕ | ✕ | ✕ |
| 35 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 36 | ✕ | ✕ (C) | ✕✕ | ✕ (T,C) | ✕ | ✕ | ✕ |
| 37 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| 38 | ✓ | ✕ (C) | ✓ | ✕ (T,C) | ✓ | ✓ | ✕ |
| **Result** | **66%**  **(25/38)** | **16%**  **(6/37)** | **66%**  **(25/38)** | **29%**  **(11/38)** | **74%**  **(28/38)** | **74%**  **(28/38)** | **31.6%**  **(12/38)** |
| **Time**  **(Seconds)** | **5.26 s** | **89.42s** | **94.68s** | **2.49s** | **7.75s** | **97.17s** | **91.91s** |

## Chart

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bug | SBFL | IR (CS) | SBFL + IR(CS) | IR (Query) | SBFL +  IR (Query) | SBFL+IR (CS+Query) | IR (CS+Query) |
| 1 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 2 | ✕ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 3 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 4 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 5 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 6 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 7 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 8 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 9 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 10 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 11 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 12 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 13 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 14 | ✕ | No BR | ✕✕ | No BR | ✕ | ✕✕ | No BR |
| 15 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 16 | ✓ | ✕ | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 17 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 18 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 19 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 20 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 21 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 22 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 23 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 24 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 25 | ✕ | No BR | ✕✕ | No BR | ✕ | ✕✕ | No BR |
| 26 | ✕ | No BR | ✕✕ | No BR | ✕ | ✕✕ | No BR |
| **Result** | **85%**  **(22/26)** | **87.5%**  **(7/8)** | **88.5%**  **(23/26)** | **100%**  **(8/8)** | **88.5%**  **(23/26)** | **88.5%**  **(23/26)** | **100%**  **(8/8)** |
| **Time** | **2.85 s** | **21.75s** | **24.6s** | **0.49s** | **3.34** | **25.09s** | **22.24s** |

## Math

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bug | SBFL | IR (CS) | SBFL + IR(CS) | Search | SBFL +  IR (Query) | SBFL+IR (CS+Query) | IR (CS+Query) |
| 1 | ✕ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 2 | ✕ | ✓ | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 3 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 4 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 5 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 6 | ✓ | ✕ | ✓ | ✕ (C,T) | ✓ | ✓ | ✕ |
| 7 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 8 | ✓ | ✓ | ✓ | ✕ (C,T) | ✓ | ✓ | ✓ |
| 9 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 10 | ✓ | ✕ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 11 | ✕ | ✓ | ✓ | ✕ (C,T) | ✕ | ✓ | ✓ |
| 12 | ✕ | ✕ | ✕✕ | ✕ (C,T) | ✕ | ✕✕ | ✕ |
| 13 | ✕ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 14 | ✕ | ✕ | ✕✕ | ✕ (C,T) | ✕ | ✕✕ | ✕ |
| 15 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 16 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 17 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 18 | ✓ | ✕ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 19 | ✕ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 20 | ✓ | ✕ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 21 | ✕ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 22 | ✓ | ✕ | ✓ | ✕ (C,T) | ✓ | ✓ | ✕ |
| 23 | ✕ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 24 | ✕ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 25 | ✓ | ✕ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 26 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 27 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 28 | ✕ | ✕ | ✕✕ | ✓ (C) | ✓ | ✓ | ✓ |
| 29 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 30 | ✕ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 31 | ✕ | ✕ | ✕✕ | ✕ (C,T) | ✕ | ✕✕ | ✕ |
| 32 | ✓ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 33 | ✓ | ✕ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 34 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 35 | ✓ | ✓ | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 36 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 37 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 38 | ✓ | ✕ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 39 | ✕ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 40 | ✕ | ✓ | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 41 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 42 | ✓ | ✓ | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 43 | ✓ | ✕ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 44 | ✕ | ✓ | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 45 | ✓ | ✓ | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 46 | ✓ | ✓ | ✓ | ✕ (C,T) | ✓ | ✓ | ✓ |
| 47 | ✓ | ✓ | ✓ | ✕ (C,T) | ✓ | ✓ | ✓ |
| 48 | ✕ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 49 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 50 | ✓ | ✕ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 51 | ✕ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 52 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 53 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 54 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 55 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 56 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 57 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 58 | ✕ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 59 | ✓ | ✕ | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 60 | ✕ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 61 | ✕ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 62 | ✕ | ✕ | ✕✕ | ✕ (C,T) | ✕ | ✕✕ | ✕ |
| 63 | ✓ | ✕ | ✓ | ✕ (C,T) | ✓ | ✓ | ✕ |
| 64 | ✓ | ✕ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 65 | ✓ | ✓ | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 66 | ✕ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 67 | ✕ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 68 | ✓ | ✕ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 69 | ✓ | ✕ | ✓ | ✕ (C,T) | ✓ | ✓ | ✕ |
| 70 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 71 | ✓ | ✕ | ✓ | ✕ (C,T) | ✓ | ✓ | ✕ |
| 72 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 73 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 74 | ✕ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 75 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 76 | ✓ | ✓ | ✓ | ✕ (C,T) | ✓ | ✓ | ✓ |
| 77 | ✕ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 78 | ✕ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 79 | ✕ | ✕ | ✕✕ | ✕ (C,T) | ✕ | ✕✕ | ✕ |
| 80 | ✓ | ✕ | ✓ | ✕ (C,T) | ✓ | ✓ | ✕ |
| 81 | ✓ | ✕ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 82 | ✓ | ✓ | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 83 | ✓ | ✓ | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 84 | ✕ | ✕ | ✕✕ | ✕ (C,T) | ✕ | ✕✕ | ✕ |
| 85 | ✕ | ✕ | ✕✕ | ✕ (C,T) | ✕ | ✕✕ | ✕ |
| 86 | ✕ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 87 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 88 | ✓ | ✓ | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 89 | ✓ | ✓ | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 90 | ✓ | ✓ | ✓ | ✓ (C) | ✓ | ✓ | ✓ |
| 91 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 92 | ✓ | ✕ | ✓ | ✕ (C,T | ✓ | ✓ | ✕ |
| 93 | ✓ | ✓ | ✓ | ✕ (C,T | ✓ | ✓ | ✓ |
| 94 | ✓ | ✓ | ✓ | ✕ (C,T | ✓ | ✓ | ✓ |
| 95 | ✓ | ✕ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 96 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 97 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 98 | ✓ | ✕ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 99 | ✓ | ✕ | ✓ | ✕ (C,T) | ✓ | ✓ | ✕ |
| 100 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 101 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 102 | ✓ | ✕ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 103 | ✕ | ✕ | ✕✕ | ✓ (T) | ✓ | ✓ | ✓ |
| 104 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 105 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| 106 | ✓ | ✓ | ✓ | ✓ (T) | ✓ | ✓ | ✓ |
| **Result** | **69.8%**  **(74/106)** | **55.6%**  **(59/106)** | **77.4%**  **(82/106)** | **79%**  **(84/106)** | **92.5%**  **(98/106)** | **93.4%**  **(99/106)** | **85.8%**  **(91/106)** |
| **Time** | **4.04s** | **258.89s** | **262.93s** | **14.86s** | **18.90s** | **277.79s** | **273.75s** |

## Closure

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Bug | SBFL | IR (CS) | SBFL + IR(CS) | IR (Query) | SBFL + IR (Query) | SBFL+IR (CS+Query) | IR (CS+Query) |
| 1 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 2 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 3 | ✕ | ✓ | ✓ | ✕ | ✕ | ✓ | ✓ |
| 4 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 5 | ✕ | ✓ | ✓ | ✕ | ✕ | ✓ | ✓ |
| 6 | ✓ | ✓ | ✓ | ✕ | ✓ | ✓ | ✓ |
| 7 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 8 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 9 | ✓ | ✓ | ✓ | ✕ | ✓ | ✓ | ✓ |
| 10 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 11 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 12 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 13 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 14 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 15 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 16 | ✕ | ✓ | ✓ | ✕ | ✕ | ✓ | ✓ |
| 17 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 18 | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 19 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 20 | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 21 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 22 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 23 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 24 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 25 | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 26 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 27 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 28 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 29 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 30 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 31 | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 32 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 33 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 34 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 35 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 36 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 37 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 38 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 39 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 40 | ✕ | ✓ | ✓ | ✕ | ✕ | ✓ | ✓ |
| 41 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 42 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 43 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 44 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 45 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 46 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 47 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 48 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 49 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 50 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 51 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 52 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 53 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 54 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 55 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 56 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 57 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 58 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 59 | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 60 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 61 | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 62 | ✓ | ✓ | ✓ | ✕ | ✓ | ✓ | ✓ |
| 63 | ✓ | No BR | ✓ | No BR | ✓ | ✓ | No BR |
| 64 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 65 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 66 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 67 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 68 | ✓ | ✓ | ✓ | ✕ | ✓ | ✓ | ✓ |
| 69 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 70 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 71 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 72 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 73 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 74 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 75 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 76 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 77 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 78 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 79 | ✕ | ✓ | ✓ | ✕ | ✕ | ✓ | ✓ |
| 80 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 81 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 82 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 83 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 84 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 85 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 86 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 87 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 88 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 89 | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 90 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 91 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 92 | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 93 | ✕ | No BR | ✕ | No BR | ✕ | ✕ | No BR |
| 94 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 95 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 96 | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 97 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 98 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 99 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 100 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 101 | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 102 | ✕ | ✓ | ✓ | ✕ | ✕ | ✓ | ✓ |
| 103 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 104 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 105 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 106 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 107 | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 108 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 109 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 110 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 111 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 112 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 113 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 114 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 115 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 116 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 117 | ✓ | ✓ | ✓ | ✕ | ✓ | ✓ | ✓ |
| 118 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 119 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 120 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 121 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 122 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 123 | ✓ | ✕ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 124 | ✓ | ✕ | ✓ | ✕ | ✓ | ✓ | ✕ |
| 125 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| 126 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 127 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 128 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 129 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 130 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 131 | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ | ✕ |
| 132 | ✕ | ✕ | ✕ | ✓ | ✓ | ✓ | ✓ |
| 133 | ✓ | ✓ | ✓ | ✕ | ✓ | ✓ | ✓ |
| **Result** | **53%**  **(70/133)** | **31.3%**  **(41/131)** | **65.4%**  **(87/133)** | **55%**  **(72/131)** | **73%**  **(97/133)** | **78.2%**  **(104/133)** | **64%**  **(84/131)** |
| **Time** | **643.72s** | **405.37s** | **1049.09s** | **13.93s** | **657.65s** | **1063.02s** | **419.3s** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Program | SBFL | | | IR | | | Search Engine | | |
| Accuracy | Time (s) | Time per bug (s) | Accuracy | Time (s) | Time per bug (s) | Accuracy | Time (s) | Time per bug (s) |
| Time | (18/27) 67% | 31. 35 | 1.16 | (14/26)  54% | 74.73 | 2.87 | **(19/26)**  **73%** | 1.50 | **0.06** |
| Mockito | **(25/38) 66%** | 5.26 | 0.14 | (6/37)  16% | 89.42 | 2.42 | (11/38)  29% | 2.49 | **0.07** |
| Lang | (60/65) 92% | 0.73 | **0.01** | (54/63)  86% | 191.66 | 3.04 | **(61/64)**  **94%** | 4.11 | 0.06 |
| Math | (74/106) 69.8% | 4.04 | **0.04** | (59/106)  55.6% | 258.89 | 2.44 | **(84/106)**  **79%** | 14.86 | 0.14 |
| Chart | **(22/26) 85%** | 2.85 | **0.11** | (7/8)  87.5% | 21.75 | 2.72 | **(8/8)**  **100%** | 0.49 | **0.06** |
| Closure | (70/133) 53% | 643.72 | 4.84 | (41/131)  31.3% | 405.37 | 3.10 | **(72/131)**  **55%** | 13.93 | **0.11** |
| Overall performance | (269/395) 68.1% | 687.95 | 1.74 | (181/371)  48.8% | 1041.82 | **2.81** | **(255/373)**  **68.4%** | **37.38** | **0.10** |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Program | SBFL + IR | | | SBFL + Search Engine | | | SBFL + IR + Search Engine | | | Search Engine + IR | | |
| Accuracy | Time (s) | Time per bug (s) | Accuracy | Time (s) | Time per bug (s) | Accuracy | Time (s) | Time per bug (s) | Accuracy | Time (s) | Time per bug (s) |
| Time | (21/27)  78% | 106.08 | 3.93 | (24/27)  89% | 32.85 | 1.22 | **(25/27)**  **93%** | 107.58 | 3.98 | (23/26)  88.46% | 76.23 | 2.93 |
| Mockito | (25/38)  66% | 94.68 | 2.49 | **(28/38)**  **74%** | 7.75 | 0.20 | (28/38)  74% | 97.17 | 2.56 | (12/38)  31.6% | 91.91 | 2.42 |
| Lang | (62/65)  95% | 192.39 | 2.96 | (64/65)  98.5% | 4.84 | 0.07 | **(65/65)**  **100%** | 196.5 | 3.02 | (63/64)  98.4% | 195.77 | 3.06 |
| Math | (82/106)  77.4% | 262.93 | 2.48 | (98/106)  92.5% | 18.9 | 0.18 | **(100/106)**  **94.3%** | 277.79 | 2.62 | (91/106)  85.8% | 273.75 | 2.58 |
| Chart | (23/26)  88.5% | 24.6 | 0.95 | (**23/26)**  **88.5%** | 3.34 | 0.13 | (23/26)  88.5% | 25.09 | 0.97 | **(8/8)**  **100%** | 22.24 | 2.72 |
| Closure | (87/133)  65.4% | 1049.09 | 7.89 | (97/133)  73% | 657.65 | 4.94 | **(104/133)**  **78.2%** | 1063.02 | 7.99 | (84/131)  64% | 419.3 | 3.20 |
| Overall performance | (300/395)  75.95% | 1729.77 | 20.7 | (334/395)  84.56% | 725.33 | 6.74 | **(345/395)**  **87.34%** | 1767.15 | 20.27 | (281/373)  75.34% | 1079.2 | 16.91 |

## Time bugs

SBFL

E =27

2

3

5

8

2

IR

Search

Lang bugs

SBFL

E =65

7

1

50

1

IR

Search

Mockito bugs

SBFL

E =38

10

1

3

5

0

IR

Search

Chart bugs

SBFL

E =26

3

1

0

6

1

IR

Search

Math bugs

SBFL

E =106

7

16

6

44

8

IR

Search

Closure bugs

SBFL

E =133

30

27

6

18

11

IR

Search

1. https://github.com/rjust/defects4j [↑](#footnote-ref-1)