

ALLEGHENY COLLEGE
DEPARTMENT OF COMPUTER SCIENCE

Senior Thesis

AFLuent: An Implementation and Evaluation of Automated Fault Localization Tool in Python

by

Noor Buchi

ALLEGHENY COLLEGE

COMPUTER SCIENCE

Project Supervisor: **Dr. Gregory Kapfhammer**
Co-Supervisor: **Maria Kim Heinert**

May 14, 2022

Abstract

Debugging code is one of the most time consuming and costly tasks for developers. To reduce the time spent on debugging code, many efforts have been focused to automate this process. Spectrum-Based fault localization is one of the ways developers can rank statement based on their potential of being faulty. This research proposes *AFLuent*[2], an easy to use Spectrum-Based fault localization tool integrated directly into Python’s testing framework, Pytest, making it user friendly for installation and usage. The tool implements four main SBFL equations, Tarantula, Ochiai, Ochiai2, and DStar, which are evaluated in this research. Additionally, several fixes are proposed and evaluated in aims to reduce the frequency of ties when ranking statements. By parsing the syntax tree of the code, several indicators can be analyzed to determine error prone statements and create a metric that breaks the ties produced by the SBFL equations. While the results of this research do not declare a clear winner, a new understanding of the effectiveness of tie breaking approaches is achieved. The study concludes that the logical variants of Ochiai, Ochiai2, and DStar, while very similar to each other, outperformed the rest of the approaches. Lastly, the study evaluates the time overhead introduced by AFLuent and the time cost developers must pay if they choose to use the tool. Integrating AFLuent into small scale projects brings very little increase in the time taken to run the test, however, the performance becomes worse as the codebase increases in size. There are many possibilities of extending this work, improving effectiveness, efficiency, and performing a more thorough evaluation can significantly increase confidence in AFLuent’s ability to locate faults.

Acknowledgment

Many thanks to Dr. Kapfhammer for his invaluable assistance in the development of this idea, the implementation, and evaluation of this tool. His continuous advice and encouragement throughout the research was crucial in pointing me in the right direction. I'm grateful for the time and effort he dedicated to ensure the completion of this project. I'm also thankful for the Allegheny College Computer Science faculty and my classmates for all their support and willingness to give constructive feedback.

Abbreviations

AFL	automated fault Localization
SBFL	spectrum-based fault localization

Glossary

Spectrum Based Fault Localization	The automated approach to rank statements based on a measure of how suspicious they are of being faulty.
Suspiciousness Score	A numerical score that indicates the likelihood of a statement being faulty.
Pytest	An extendable Python tool for automated tests.
Per-test Coverage	The lines of code under test executed by each test case.
Mutant Density	The number of possible mutants that could be added to a statement.
Random Tie Breaking	The randomization of ranks between tied statements with the same suspiciousness scores.
Cyclomatic Tie Breaking	The reliance on the cyclomatic complexity score of a function to resolved ties between ranked statements.
Logical Tie Breaking	A metric used to break ties by measuring the mutant density of a statement.
Enhanced Tie Breaking	A holistic metric used to break ranking ties by including mutant density of a statement and its parent nodes.
<i>EXAM</i> Score	The percentage of results that a developer must parse before finding the correct location of the fault.

Contents

Abstract	i
Acknowledgment	ii
Abbreviations	iii
Glossary	iv
Contents	vi
List of Figures	viii
List of Tables	x
1 Introduction	1
1.1 Motivation	1
1.2 Background	1
1.2.1 Debugging and Fault Localization	1
1.2.2 Spectrum-Based Fault Localization	3
1.2.3 SBFL in Action	5
1.2.4 SBFL Criticism	6
1.3 Main Aims	6
1.4 Research Questions	6
1.5 Thesis Outline	7
2 Related Work	9
2.1 Automated Fault Localization	9
2.1.1 SBFL Approaches	10
2.1.2 Combining Approaches	13
2.1.3 Acknowledging Problems	14
2.2 Existing Tools	14
2.3 Usability and Accessibility	15
3 Method of Approach	17
3.1 Development Environment and Toolset	17
3.1.1 Poetry	17

3.1.2	Pytest	17
3.1.3	Coverage.py	18
3.1.4	Radon	18
3.1.5	Libcst	18
3.2	AFLuent as a Pytest Plugin	18
3.3	Installing AFLuent	19
3.3.1	Adding Command-Line Arguments	19
3.3.2	Activating AFLuent	21
3.3.3	Calculating Per-test Coverage and Test Result	21
3.3.4	Reporting Results	21
3.4	AFLuent's Components	22
3.4.1	Line Object	22
3.4.2	ProjFile Object	25
3.4.3	Spectrum Object	26
3.4.4	Afluent Object	26
3.4.5	Objects Overview	26
3.5	AFLuent's Output	26
3.5.1	Feedback Messages	27
3.5.2	Console Report	30
3.5.3	File Report	32
3.6	Tie Breaking	32
3.6.1	Random	33
3.6.2	Cyclomatic Complexity	33
3.6.3	Mutant Density: Logical Set	33
3.6.4	Mutant Density: Enhanced Set	33
3.6.5	Tiebreaking Overview	35
4	Experimental Results	37
4.1	Experimental Design	37
4.1.1	Approach Overview	37
4.1.2	Research Questions	38
4.1.3	Choosing a Sample	38
4.2	Evaluation	42
4.2.1	Data Collection	42
4.2.2	Results	43
4.3	Threats to Validity	53
5	Discussion and Future Work	54
5.1	Summary of Results	54
5.2	Future Work	54
5.3	Ethical Implications	56
	Bibliography	57

List of Figures

1.1	API Developers Time Allocation [1]	2
1.2	Variables in SBFL	4
1.3	Tarantula Equation[13]	4
1.4	Ochiai Equation[6]	4
1.5	DStar Equation[26]	5
1.6	Applied Example of SBFL Equations	5
2.1	AFL papers found by Wong et al. [27]	9
2.2	Coefficient Based Formulas [27]	11
2.3	Ochiai2 Equation[27]	12
2.4	Example of Zoltar Interface [11]	15
2.5	Example interface of CharmFL [22]	16
3.1	Pytest Tasks Flowchart	19
3.2	Tarantula Division by Zero Example 1	23
3.3	Tarantula Division by Zero Example 2	23
3.4	Ochiai Division by Zero Example 1	23
3.5	Ochiai Division by Zero Example 2	23
3.6	Ochiai2 Division by Zero Example 1	24
3.7	Ochiai2 Division by Zero Example 2	24
3.8	Ochiai2 Division by Zero Example 3	25
3.9	Ochiai2 Division by Zero Example 4	25
3.10	DStar Division by Zero Example	25
3.11	AFLuent Object Structure	27
3.12	Example of Success Message	28
3.13	Example of Error Message	29
3.14	Maxfail Warning Message	29
3.15	AFLuent Disabled Warning Message	30
3.16	Conflicting Plugin Warning Message	31
3.17	Example Report Using Ochiai and DStar	31
3.18	Example Report With Safe Statements	32
3.19	Enhanced Score Equation	35
4.1	SLOCcount Output After Initial Filtering	40
4.2	SLOCcount Output After Initial Filtering	41
4.3	Cyclomatic Complexity of Sample	41
4.4	EXAM Scores Using Random Tiebreaking	44

4.5	<i>EXAM</i> Scores Using Cyclomatic Tiebreaking	44
4.6	<i>EXAM</i> Scores Using Logical Tiebreaking	45
4.7	<i>EXAM</i> Scores Using Enhanced Tiebreaking	45
4.8	Average <i>EXAM</i> Scores for all Approaches	46
4.9	<i>p-values</i> of Two Sided Mann-Whitney Test	47
4.10	<i>p-values</i> of One Sided Mann-Whitney Test	49
4.11	Results of the non-parametric Cohen D Effect Size	51
4.12	Test Execution Time	52
4.13	Fault Localization Time	52

List of Tables

3.1	Mutants in the Logical Set	34
3.2	Mutants in the Enhanced Set	34
3.3	Construct Scoring in Enhanced Set	35
3.4	Tiebreaking Score Examples	35
4.1	Remaining Projects Prior to Test Generation	39

Chapter 1

Introduction

1.1 Motivation

Debugging is a task that every developer and software engineer will have to go through while writing code. It's often a tedious process that involves manually checking the code and running it multiple times to simply find the location of the error in the code. More formally, debugging “involves analyzing and possibly extending the given program that does not meet the specification in order to find a new program that is close to the original and does satisfy the specifications... it is the process of ‘diagnosing the precise nature of a known error and then correcting it’ ” [10]. Finding possible solutions to fix the fault would then be the next step in the debugging process. In Postman’s 2020 State of the API Report [1], developers reported allocating seventeen percent of their time debugging and manually testing their code. Fig 1.1 demonstrates that debugging makes up the second most time-consuming task for API developers. In an effort to facilitate and automate this process, different tools have been created to guide the developer in locating error(s) and suggesting how to go about fixing them. Solving this problem can significantly increase the efficiency of experienced developers and increase the quality of the code they ship. Additionally, a solution can help in decreasing the time spent on debugging and allow developers to focus on other important tasks. Beginner and novice developers can also benefit greatly from tools that facilitate the debugging process, which could assist in learning and help them avoid making mistakes. Overall, contributions to this field of study could benefit developers and software engineers of all levels. With special concentration on novice and beginner developers in an educational environment, this research implements and evaluates a tool focused on helping developers find bugs in their code.

1.2 Background

1.2.1 Debugging and Fault Localization

The debugging process for developers typically starts when a “bug”, unexpected/incorrect behavior has been observed. One of many ways to check whether incorrect behavior is taking place is through unit testing. This type of testing consists of a set of

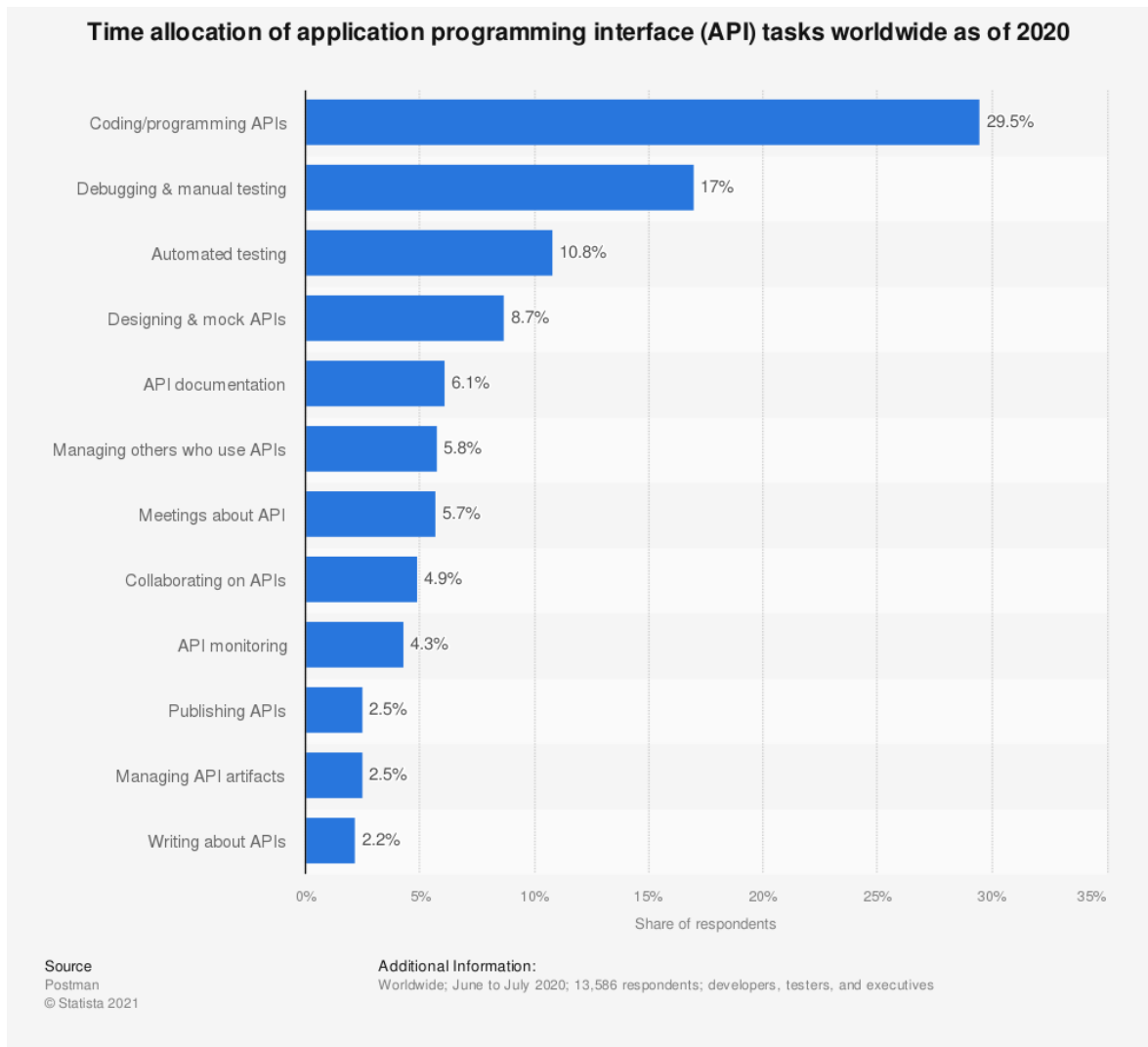


Figure 1.1: API Developers Time Allocation [1]

test cases that check the different functionalities of the code by asserting that expected output and actual output match. When a test case fails due to a mismatch in the assertion, the search for the causing bug begins. This research focuses on this crucial step in debugging and aims to guide the search process by setting priority locations for the developers to search in.

Debugger tools have often been used to find and fix bugs in code, and while they have several benefits, they also have drawbacks. Debuggers provide the developer with a way to essentially pause the runtime of a program using breakpoints to display information about the state of different variables and function call stack. Through debuggers, developers can run the code line by line until an unexpected behavior is detected. There is no denying that debuggers are very effective, however, using them can be time-consuming, especially for complex bugs. Additionally, they require that the code is run during the debugging process which could increase the time needed to debug and might not always be feasible. With these drawbacks in mind, several

automated fault localization (AFL) algorithms and approaches have been studied to increase the efficiency of the debugging process.

Automated Fault Localization approaches perform different types of analysis to the code and attempt to find the location of existing faults. Some approaches use trained artificial intelligence models to assess the code and make predictions on where some faults may exist. Others construct and analyze abstract syntax trees to look for potential errors. On the other hand, approaches such as spectrum-based fault localization rely on data collected during the runtime of the code to “identify the part of the program whose activity correlates most with the detection of errors” [5]. More specifically, the results of the program’s test suite are used to find this correlation. This latter approach is the focus of this research, where a spectrum-based automated fault localization tool is implemented.

1.2.2 Spectrum-Based Fault Localization

The reliance of spectrum-based fault localization (SBFL) on data already collected during running and testing reduces the overall cost of debugging since there is no need to rerun the code. When focusing on collected test suite data, SBFL becomes even more applicable since it can be integrated with many unit test frameworks. In addition to the per-test outcome of a test suite, per-test coverage data is also used in SBFL to identify which lines are executed by each test case. Once this data is collected following a test suite run, SBFL assesses the suspiciousness of code blocks depending on how many failing test cases run those blocks. From a logical point of view, the block/line of code executed the most by a failing test case is the most likely to contain the fault. SBFL also uses various equations to calculate and quantify the suspiciousness of blocks of code. The equations are expressed as functions of elements, where each element is the line or block of code under inspection. The functions typically include the variables shown in Figure 1.2. The figure also provides a table to visualize the relationship of the variables [26].

Using these variables, past research has come up with many equations to calculate elements’ suspiciousness scores. This research focuses on four main popular equations and adds support for them in AFLuent. The reasoning behind the equation choices is further discussed in Related Works, however, this section will give an overview of how each looks like.

Tarantula

The Tarantula equation is one of the most popular Coverage based fault localization formulas to calculate suspiciousness scores of code elements. It uses the variables listed previously to generate a score between 0 and 1, where 0 is assigned for non-suspicious elements and 1 is assigned for most suspicious ones.

Ochiai

Ochiai is another equation used to calculate suspiciousness scores between 0 and 1. It relies on similar code test and coverage variables as the Tarantula equation. AFLu-

N_{CF} : number of failed test cases that execute/cover the element
 N_{UF} : number of failed test cases that do not execute/cover the element
 N_{CS} : number of passed test cases that execute/cover the element
 N_{US} : number of passed test cases that do not execute/cover the element
 Where an element is a unit of code such as a statement or a block.

		Was the Statement Covered?		SUM
		Yes (1)	No (0)	
Execution Result	Failure (1)	a (N_{CF})	b (N_{UF})	$a + b$ (N_F)
	Success (0)	c (N_{CS})	d (N_{US})	$c + d$ (N_S)
SUM		$a + c$ (N_C)	$b + d$ (N_U)	n

Figure 1.2: Variables in SBFL

$$Tarantula(element) = \frac{\frac{N_{CF}}{N_{CF}+N_{UF}}}{\frac{N_{CS}}{N_{CS}+N_{US}} + \frac{N_{CF}}{N_{CF}+N_{UF}}} \quad (1.1)$$

Figure 1.3: Tarantula Equation[13]

$$Ochiai(element) = \frac{N_{CF}}{\sqrt{(N_{CF} + N_{UF}) \cdot (N_{CF} + N_{CS})}} \quad (1.2)$$

Figure 1.4: Ochiai Equation[6]

ent supports using this equation in its automated fault localization process. More discussion around Ochiai can be found in the related works section.

DStar

The scores resulting from the DStar equation can range beyond 0 and 1 because it includes a new variable represented as the * symbol. This variable can give more weight to N_{CF} and can be set to any number. However, the researchers suggested that * is set to 2 or 3.

$$Dstar(element) = \frac{(N_{CF})^*}{N_{CS} + N_{UF}} \quad (1.3)$$

Figure 1.5: DStar Equation[26]

1.2.3 SBFL in Action

Code	Input Tests							Suspiciousness		
0: def maximum():	1,2,3	0,-2,1	1,3,2	1,3,3	5,5,1	2,2,2	6,1,3	Tarantula	Ochiai	DStar
1: a = int(input("Enter first number:"))	●	●	●	●	●	●	●	0.5	0.378	0.1667
2: b = int(input("Enter second number:"))	●	●	●	●	●	●	●	0.5	0.378	0.1667
3: c = int(input("Enter third number:"))	●	●	●	●	●	●	●	0.5	0.378	0.1667
4: print("Finding maximum...")	●	●	●	●	●	●	●	0.5	0.378	0.1667
5: if (a >= b) and (a >= c):	●	●	●	●	●	●	●	0.5	0.378	0.1667
6: largest = b # bug here						●	●	0.857	0.707	1
8: elif (b >= a) and (b >= c):	●	●	●	●	●			0	0	0
9: largest = b			●	●	●			0	0	0
10: else:	●	●						0	0	0
11: largest = c	●	●						0	0	0
12: print(f"Maximum: {largest}")	●	●	●	●	●	●	●	0.5	0.378	0.1667
Results	PASS	PASS	PASS	PASS	PASS	PASS	FAIL			

Figure 1.6: Applied Example of SBFL Equations

With the discussed equations and techniques to calculate suspiciousness in mind, the following example demonstrates and discusses how AFLuent utilizes unit test and coverage data to locate faults in a program. Figure 1.6 shows a sample python program to find the middle value from three integers. The program contains a bug on line 6 where the wrong maximum value is detected. The figure also shows seven different test cases that send various inputs to the function and check whether the actual output matches the expected. The results of each test are found on the last row of the table. Additionally, the large dots under the Input Tests column illustrate the concept of code coverage. For each line of code and test input, a dot in the cell means that the line was executed when this input was passed. On the rightmost column of the table, suspiciousness scores for each line are calculated using the three formulas (Tarantula, Ochiai, and DStar). High suspiciousness scores are also highlighted in warmer colors. Overall, all three equations are able to detect that line 9 is the most suspicious and is likely the cause of the test failure.

The suspiciousness were calculated for each line by plugging values into the equations shown on Figures 1.3, 1.4, and 1.5. For example, the scores for line 5 were calculated as follows:

$$\begin{aligned}
 Tarantula(5) &= \frac{\frac{1}{6}}{\frac{1}{6} + 1} \\
 Ochiai(5) &= \frac{1}{\sqrt{(1+0) \cdot (1+6)}} = \frac{1}{\sqrt{7}} \\
 DStar(5) &= \frac{1^3}{6+0}
 \end{aligned}$$

1.2.4 SBFL Criticism

While SBFL can provide great insight for developers looking to debug their code, it's not always reliable or applicable. In the case where a test suite hasn't been fully implemented, the output of SBFL may not narrow down potentially faulty code to a single line. Additionally, ties in suspiciousness scores between lines and blocks may come up, which may not offer the developer any valuable output. Overall limitations of SBFL will be discussed in Related Works and acknowledged while evaluating the outcome of this research.

1.3 Main Aims

With the demonstrated importance of debugging tools and strategies that increase developer efficiency, this research implements and evaluates an AFL tool to assist in debugging code. AFLuent is built using Python programming language and uses SBFL approach to identify faults in a program. Stack Overflow 2021 Developer Survey [23] identified Python as the third most used programming/markup language used by developers worldwide. With that in mind, an automated fault localization tool in Python can reach a wide audience in various fields such as data science and AI who may have various levels of experience. By implementing AFLuent using Python, powerful libraries and packages such as Pytest and Coverage.py can be used to collect test suite data in order to calculate suspiciousness. AFLuent runs as a Pytest plugin and is integrated with the command line interface of Pytest, this feature increases its accessibility by allowing developers to easily integrate it into their development environment.

Following the implementation of AFLuent, this research evaluates the tool's performance and accuracy. Several open source projects available on GitHub are handpicked to run AFLuent against. A representative sample of projects with various sizes and relevance will be used to collect data. Mutation testing libraries are used to introduce bugs into the code, following that step, AFLuent uses the resulting test suite data to identify the introduced fault. Through this process, AFLuent's performance and ability to correctly identify and rank suspicious statements/and blocks assessed. The evaluation section will describe this crucial process that ensures that AFLuent functions as intended and is able to locate faults in code.

1.4 Research Questions

Throughout the implementation, testing, and evaluation of AFLuent, this research presents the following research questions and aims to answer each in detail. There are two main research questions that focus on different aspects of AFLuent. Each one is further split into sub-questions to facilitate organization.

RQ1. What are the most accurate available techniques to automate the process of fault localization using test coverage data?

This general research question focuses on the core implementation and evaluation of AFLuent. Considering that many SBFL strategies and formulas can be used, finding the most suitable and most accurate one in detecting bugs is one of the goals of this research. In order to answer this question, available literature on SBFL is analyzed and the most popular and cited formulas are included in the implementation of AFLuent. Answering this question also requires that each approach is evaluated through an experiment section. Since this research question includes two separate sections, it's further split into smaller sub-questions discussed below.

RQ1.1. What are the four most popular formulas for code suspiciousness created and evaluated by past literature?

Available literature surrounding SBFL is reviewed to identify four different formulas used to calculate suspiciousness scores using test coverage information. Each of these formulas are integrated into AFLuent where the user is able to select the approach to use while debugging. The popularity, accuracy, and performance as found in past work are the main criteria used to pick the top formulas.

RQ1.2. What is the accuracy and efficiency of the chosen formulas when applied to Python projects?

To ensure correctness and effectiveness, the implemented formulas in AFLuent are evaluated through experiments that measure their accuracy in sorting suspicious statements and blocks. More specifically, the formulas will be assessed in the context of Python projects that use the Pytest unit testing framework. More details on this research question can be found in the evaluation section.

While RQ1 involves technical details around implementation and assessment of AFLuent, RQ2 focuses on user experience. Since the target audience of AFLuent is beginner developers, it's important to ensure that AFLuent produces meaningful and clear results that guide developers. In order to answer this question, AFLuent adopts various standards in interacting with the user through console output, as well as error messages and warnings.

RQ2. How can automated fault localization techniques be implemented in Python as a novice developer friendly tool?

In addition to ensuring a smooth user experience while utilizing AFLuent functionalities, setup process and usage of the tool AFLuent is simplified to facilitate installation. Clear and descriptive documentation is also a crucial step in making AFLuent accessible and available for new users.

1.5 Thesis Outline

Starting with a literature review, this research identifies important sources that guide the implementation of AFLuent and discusses their use. Additionally, the methods section extensively describes how AFLuent is implemented and the steps taken to

ensure it functions correctly while being up to industry standards. The different tools used to build and test AFLuent are also discussed in the methods sections. Following that, the evaluation section describes the steps taken to evaluate AFLuent by testing the tool and collecting data regarding its output. The evaluation section also includes an analysis of the results of the evaluation and various plots and charts that show the findings.

Chapter 2

Related Work

Automated fault localization in general and more specifically spectrum-based fault localization have extensive literature exploring the different approaches to facilitate debugging and increase developer efficiency. Considering that AFLuent relies on many concepts developed by this literature, this section will explore and discuss how past work shapes AFLuent. Several sections are created for specific areas of literature.

2.1 Automated Fault Localization

Survey papers are one of the many ways that assist in better understanding and having a wide collection of the existing literature surrounding automated fault localization (AFL). Wong et al. [27] explores the variety of AFL approaches and surveys research completed in that area between 1977 and 2014. The survey paper finds and discusses more than 334 different papers in many approaches in AFL.

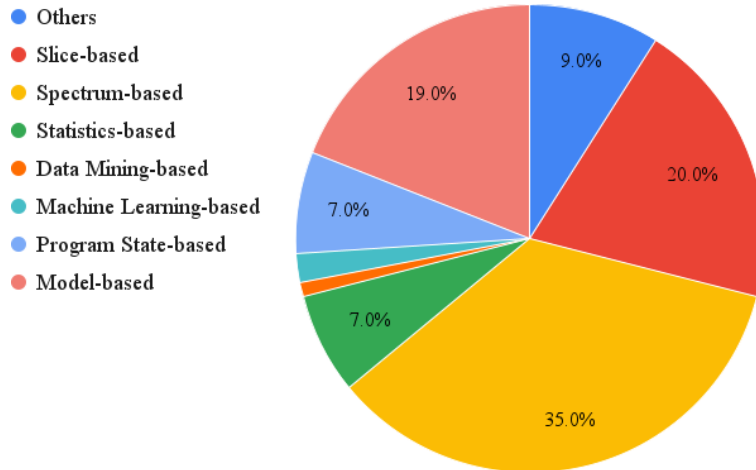


Figure 2.1: AFL papers found by Wong et al. [27]

A breakdown of the found research is shown in Figure 2.1, where the majority of found papers are focused on Spectrum-Based Fault Localization (SBFL). Overall this research provides a great starting point to find and compare the different types and

approaches of AFL. Another benefit of these resources is that Wong et al. [27] expands on the types of SBFL and reviews key literature that contributes to show the benefits and drawbacks of each approach.

Another insightful survey paper is by Idrees Sarhan et. al [21] where he analyzes the main challenges in SBFL and some of the major obstacles that other literature attempts to tackle. AFLuent addresses two of the issues listed in this survey study, specifically element ties and division by zero. But many other challenges such as test flakiness, single vs. multiple bugs, and many others remain unaddressed.

2.1.1 SBFL Approaches

Similarity Coefficient Based Technique

One of the most relevant SBFL techniques described by Wong et al. [27] is similarity coefficient based ones. Generally, these approaches seek to quantify how close “the execution pattern of a statement is to the failure pattern of all test cases”, where the closer they are the more likely that this statement to contain the error. In order to create a measurement of closeness, several equations have been developed and evaluated by past literature. Figure 2.2 shows some of the equations reviewed by Wong et al in, however, more popular formulas have been developed that required the developer to analyze less code before finding the fault. Despite the existence of many Similarity Coefficient formulas, Yoo et al. [31] concludes in an extensive evaluation study that there is no formula that outperforms all others in all circumstances. With that in mind, AFLuent gives the user the option to choose which approach to use and includes a performance evaluation of each.

Tarantula

Starting off with the Tarantula formula (figure 1.3), it’s one of the foundational equations that was introduced to attempt to visualize suspicious statements using color ranges [12, 13]. A tool using this equation was also implemented in Java in order to scan code and assign colors to statements based on their suspiciousness. The equation for Tarantula is made of two main ratios, the first one being the number of failed tests that cover the element divided by the total number of failed tests ($\frac{N_{CF}}{N_{CF} + N_{UF}}$). The other ratio used is number of passing tests that cover the element divided by total number of passing tests ($\frac{N_{CS}}{N_{CS} + N_{US}}$). The equation is then assembled as seen in figure 1.3. To better understand how the equation works, we can look at the important terms that have the largest influence in changing the output. The number of failed tests that cover the element is clearly the main influencer here because it’s what causes the numerator to grow larger. This means that an increase in failed tests that cover the element cause an increase in suspiciousness. Additionally, a decrease in the number of failing tests that do not cover the element also increases suspiciousness. Considering these two points, Tarantula gives a better indicator of suspiciousness when there are fewer failures in tests covering elements not under inspection. In addition to the logical analysis of the equation previous works provide an empirical evaluation of Tarantula in comparison to other formulas. Jones et al. [13] compares the effectiveness and efficiency of Tarantula to techniques such as Set Union, Set Intersection, and Nearest

Coefficient		Algebraic Form	Coefficient		Algebraic Form
1	Braun-Banquet	$\frac{N_{CF}}{\max(N_{CF}+N_{CS}, N_{CF}+N_{UF})}$	17	Hamonic Mean	$\frac{(N_{CF} \times N_{US} - N_{UF} \times N_{CS})((N_{CF}+N_{CS}) \times (N_{US}+N_{UF}) + (N_{CF}+N_{UF}) \times (N_{CS}+N_{US}))}{(N_{CF}+N_{CS}) \times (N_{US}+N_{UF}) \times (N_{CF}+N_{UF}) \times (N_{CS}+N_{US})}$
2	Dennis	$\frac{(N_{CF} \times N_{US}) - (N_{CS} \times N_{UF})}{\sqrt{n \times (N_{CF}+N_{CS}) \times (N_{CF}+N_{UF})}}$	18	Rogot2	$\frac{1}{4} \left(\frac{N_{CF}}{N_{CF}+N_{CS}} + \frac{N_{CF}}{N_{CF}+N_{UF}} + \frac{N_{US}}{N_{US}+N_{CS}} + \frac{N_{US}}{N_{US}+N_{UF}} \right)$
3	Mountford	$\frac{N_{CF}}{0.5 \times ((N_{CF} \times N_{CS}) + (N_{CF} \times N_{UF})) + (N_{CS} \times N_{UF})}$	19	Simple Matching	$\frac{N_{CF}+N_{US}}{N_{CF}+N_{CS}+N_{US}+N_{UF}}$
4	Fossum	$\frac{n \times (N_{CF} - 0.5)^2}{(N_{CF}+N_{CS}) \times (N_{CF}+N_{UF})}$	20	Rogers & Tanimoto	$\frac{N_{CF}+N_{US}}{N_{CF}+N_{US}+2(N_{UF}+N_{CS})}$
5	Pearson	$\frac{n \times ((N_{CF} \times N_{US}) - (N_{CS} \times N_{UF}))^2}{N_{CF} \times N_{US} \times N_{CS} \times N_{UF}}$	21	Hamming	$N_{CF} + N_{US}$
6	Gower	$\frac{N_{CF}+N_{US}}{\sqrt{N_{CF} \times N_{CS} \times N_{US} \times N_{UF}}}$	22	Hamann	$\frac{N_{CF}+N_{US}-N_{UF}-N_{CS}}{N_{CF}+N_{US}+N_{CF}+N_{US}}$
7	Michael	$\frac{4 \times ((N_{CF} \times N_{US}) - (N_{CS} \times N_{UF}))}{(N_{CF}+N_{US})^2 + (N_{CS}+N_{UF})^2}$	23	Sokal	$\frac{2(N_{CF}+N_{US})}{2(N_{CF}+N_{US})+N_{UF}+N_{CS}}$
8	Pierce	$\frac{(N_{CF} \times N_{US}) + (N_{UF} \times N_{CS})}{(N_{CF} \times N_{UF}) + (2 \times (N_{UF} \times N_{US})) + (N_{CS} \times N_{US})}$	24	Scott	$\frac{4(N_{CF} \times N_{US} - N_{UF} \times N_{CS}) - (N_{UF} - N_{CS})^2}{(2N_{CF}+N_{UF}+N_{CS})(2N_{US}+N_{UF}+N_{CS})}$
9	Baroni-Urbani & Buser	$\frac{\sqrt{(N_{CF} \times N_{US}) + N_{CF}}}{\sqrt{(N_{CF} \times N_{US}) + N_{CF} + N_{CS} + N_{UF}}}$	25	Rogot1	$\frac{1}{2} \left(\frac{N_{CF}}{2N_{CF}+N_{UF}+N_{CS}} + \frac{N_{US}}{2N_{US}+N_{UF}+N_{CS}} \right)$
10	Tarwid	$\frac{(n \times N_{CF}) - (N_{F} \times N_{C})}{(n \times N_{CF}) + (N_{F} \times N_{C})}$	26	Kulczynski	$\frac{N_{CF}}{N_{UF}+N_{CS}}$
11	Ample	$\left \frac{N_{CF}}{N_{CF}+N_{UF}} - \frac{N_{CS}}{N_{CS}+N_{US}} \right $	27	Anderberg	$\frac{N_{CF}}{N_{CF}+2(N_{UF}+N_{CS})}$
12	Phi (Geometric Mean)	$\frac{N_{CF} \times N_{US} - N_{UF} \times N_{CS}}{\sqrt{(N_{CF}+N_{CS}) \times (N_{CF}+N_{UF}) \times (N_{CS}+N_{US}) \times (N_{UF}+N_{US})}}$	28	Dice	$\frac{2N_{CF}}{N_{CF}+N_{UF}+N_{CS}}$
13	Arithmetic Mean	$\frac{2(N_{CF} \times N_{US} - N_{UF} \times N_{CS})}{(N_{CF}+N_{CS}) \times (N_{US}+N_{UF}) + (N_{CF}+N_{UF}) \times (N_{CS}+N_{US})}$	29	Goodman	$\frac{2N_{CF} - N_{UF} - N_{CS}}{2N_{CF}+N_{UF}+N_{CS}}$
14	Cohen	$\frac{2(N_{CF} \times N_{US} - N_{UF} \times N_{CS})}{(N_{CF}+N_{CS}) \times (N_{US}+N_{CS}) + (N_{CF}+N_{UF}) \times (N_{UF}+N_{US})}$	30	Jaccard	$\frac{N_{CF}}{N_{CF}+N_{UF}+N_{CS}}$
15	Fleiss	$\frac{4(N_{CF} \times N_{US} - N_{UF} \times N_{CS}) - (N_{UF} - N_{CS})^2}{(2N_{CF}+N_{UF}+N_{CS}) + (2N_{US}+N_{UF}+N_{CS})}$	31	Sorensen-Dice	$\frac{2N_{CF}}{2N_{CF}+N_{UF}+N_{CS}}$
16	Zoltar	$\frac{N_{CF}}{N_{CF}+N_{UF}+N_{CS} + \frac{10000 \times N_{UF} \times N_{CS}}{N_{CF}}}$			

Figure 2.2: Coefficient Based Formulas [27]

Neighbor. The results demonstrate that Tarantula outperformed the other Techniques where it provided better guidance to the developer. Using Tarantula a developer would need to manually inspect fewer elements of the program compared to when using other approaches.

Another variation of Tarantula is also introduced in [9], where grouping of suspicious elements is used to provide a better guide for developers. In this modification, “statements that are executed by the same number of failed test cases are grouped together”, then suspiciousness scores are used to sort elements within each group. By doing so, two layers of sorting exist, the first one based on the number of failed tests covering the element, and then the suspiciousness scores. The empirical results in Debroy et al. [9] show a statistically significant improvement provided by this grouping technique where the developer needs to review less elements and more faults are accurately detected. While Debroy et al. only applied the grouping technique to Tarantula and a neural network-based approach, it could be extended to include other similarity coefficient based techniques.

Overall, while other formulas outperform Tarantula as will be discussed, incorporating this equation in AFLuent offers a starting point and a point of comparison to other equations. One of the goals of AFLuent is to give the user the ability to choose their most fitting approach to localize faults, and it’s useful to include Tarantula as one of the available options.

Ochiai

Ochiai is another similarity coefficient formula for SBFL that uses code coverage information and test output to produce a suspiciousness score. Originally used in computing genetic similarity in molecular biology and evaluated in Abreu et al. [6], the equation for this approach is shown in fig.1.4. Similar to Tarantula, the number of failing test cases that execute an element are the main factor in increasing suspiciousness. The formula uses similar terms as Tarantula such as total number of tests that cover the element, unlike Tarantula, however, it does not consider successful tests that do not cover the element. Papers such as [6, 5] also evaluate the performance of Ochiai in comparison to others such as Tarantula, AMPLE, and Jaccard. Another evaluation of Ochiai is done by Le et al. [15] where it was found to have a statistically significant improvement when compared to Tarantula. The paper demonstrates that on average developers only need to inspect 21.02% of the source code before finding the fault. AFLuent includes an implementation and evaluation of Ochiai to validate that it performs as expected compared to the Tarantula technique. Additionally, considering that Ochiai is considered a fairly accurate and effective formula to detect faults, AFLuent takes advantage of the performance it offers.

$$Ochiai2(element) = \frac{N_{CF} \cdot N_{US}}{\sqrt{(N_{CF} + N_{CS}) \cdot (N_{US} + N_{UF}) \cdot (N_{CF} + N_{UF}) \cdot (N_{CS} + N_{US})}} \quad (2.1)$$

Figure 2.3: Ochiai2 Equation[27]

In addition to the Ochiai formula in figure 1.4, another variation of Ochiai is identified and evaluated by [18]. The formula for Ochiai2 is shown in 2.3 and it takes into consideration all the possible outcomes of unit test coverage. Overall, this variation is included in AFLuent to compare the performance between Ochiai2 and the original formula and see if it provides an improvement in effectiveness. Both Ochiai and Ochiai2 are relevant similarity coefficient-based approaches that add variety to AFLuent and allow a representative evaluation of the tool that takes in many different approaches..

DStar

DStar (also written as D*) is another SPFL technique that utilizes code coverage information of a program to locate and rank faults. The equation for this approach can be found in figure 1.5. Wong et al. [26] introduce and extensively evaluate this approach in a 2014 paper that demonstrate its effectiveness compared to other formulas. In the process of constructing D*, the paper lists the factors involved in determining suspiciousness of an element. The principles are as follows:

1. Suspiciousness is directly proportional to the number of failed tests covering the element.
2. Suspiciousness is inversely proportional to the number of successful tests covering the element.

3. Suspiciousness is inversely proportional to the number of failed tests that do not cover the element.
4. The number of failed tests covering the element should have the most weight in determining suspiciousness.

Considering that multiplying N_{CF} by a constant to increase its weight will not affect the ranking of statements, the authors argue that raising N_{CF} to a value $*$ greater than or equal to 1 would be more appropriate in increasing the weight of this variable. The study continues by illustrating how increasing the value of $*$ produces more clear rankings that facilitate the debugging process by requiring the developer to examine less elements in both the best and worst case. However, the authors also point out that this benefit of increasing the value of $*$ levels off at a certain point depending on the size of the program under analysis. The paper concludes by reviewing performance results showing that D^* is more effective than the previously discussed formulas (Tarantula, Ochiai, and Ochiai2). With that in mind, D^* offers the latest and most effective formula to calculate suspiciousness compared to all others included in this research. AFLuent implements D^* to validate this step up in effectiveness in the context of Python projects and gives the user the ability to use it.

2.1.2 Combining Approaches

While AFLuent only relies on SBFL approaches in its implementations, it's useful to explore other methodologies that could assist in the debugging process. This creates a guide for potential extension of AFLuent and provides a way to fill in the shortcomings of AFLuent. Xuan et al. explores the possibility of combining several SBFL metrics of fault localization and introducing a machine learning model to assist with the ranking [30]. While AFLuent does not support this approach, Xuan et al. shows some promising results that could potentially uncover performance improvements in fault localization. There are many tricky aspects of this research, especially that it suggests training a machine learning model to assist with ranking. Depending on the data used to train the model, the results could be very different. Overall, while AFLuent does not use machine learning, this research provides a great idea for future work and improvements.

Another proposed extension to the existing SBFL work is studied by Wang et al. [24] where similar to the previous work, a "Search-Based Composition Engine" is trained using a set of coverage information as well as locations of existing bugs. Additionally, it creates a composite equation using multiple coefficient-based formulas and their weights as part of the engine. The resulting evaluation shows that this approach does, in fact, outperform standard formulas like Tarantula and Ochiai.

Overall, while these works are somewhat out of scope of AFLuent since they use different techniques, they have the same intention in optimizing the debugging process and reducing its cost. The research discussed in this section could potentially get used to extend the functionality of AFLuent where a model/engine is trained through the feedback of users which allows them to use it later in their debugging efforts.

2.1.3 Acknowledging Problems

With the multitude of approaches and formulas to use in SBFL, various criticisms are brought up for each proposed research. Some research even suggests that SBFL and AFL in general is not effective for all developers [19]. In a survey study, Wong et al. [27] identifies a series of issues and concerns surrounding SBFL in general. The main one being the central problem of giving failed and successful tests accurate weights in order to produce a meaningful suspiciousness score and reduce the potential for ties. Another identified concern is the assumption that a well written and extensive test suite exists for the program under examinations but more specifically the assumption SBFL approaches tend to make by considering passing test cases indicators of error absence. In many instances, test cases themselves could contain bugs causing them to produce inaccurate results. Existing literature on both of these concerns exist and it's required to consider this criticism in the development of AFLuent.

One of the brought up concerns of SBFL is the inclusion of passed program spectra in calculating suspiciousness of an element. Xie et al. [28] argue that while a failed program test case does indicate the presence of an error in a passed program spectra/test data, "is not guaranteed to be absolutely free of any faulty statement". With that in mind, passed test information alone does not give reliable results on elements suspiciousness. The proposed approach to mitigate this problem is to organize program entities into two main groups, those who have been "activated" at least once by a failed program spectra, and "clean" ones, which have not at all. The research continues by experimenting with this approach and presenting results that showed some signs of improvement on existing SBFL formulas. Overall, this research provides a way to address inaccuracies with AFLuent and assists in expanding the project beyond simple calculations based on formulas.

Another concern with the use of SBFL to debug programs is the possibility of having equal suspiciousness scores assigned to multiple statements. These ties hinder the debugging process and present the developer with a dilemma. Which element should be inspected first? they're equally suspicious! This problem becomes more significant when only one of the tied elements actually contains the fault. A study by Xu et al. [29] recognizes this problem and expands on the different outcomes. In the best case, the developer picks the statement containing the fault as their first choice and finds the error right after. However, the worst case would require the developer to examine every tied element before reaching the one containing the fault. The research continues by showing that ties in calculated suspiciousness scores are frequent no matter the chosen formula. Other contributions of this research include strategies to break ties and facilitate suspiciousness ranking between elements. The research concludes by presenting that the tie breaking strategies were impactful in reducing the number of found ties in Tarantula and Ochiai approaches.

2.2 Existing Tools

Some tools that perform SBFL have already been proposed and implemented. While they tend to be standalone applications that operate differently than AFLuent, it's important to explore the way these applications interact with the user and show out-

put. Zoltar [11] is a standalone tool for C/C++ programs that offers a graphical user interface for users to view program spectra and calculate suspiciousness scores using various techniques. The tool also uses the hue concept developed by an early study that proposes Tarantula [12]. By color coding statements based on their rank and suspiciousness scores, Zoltar simplifies the user interaction by making it clear which statement is the most likely to be faulty. While AFLuent lives and runs in a terminal window, it does utilize color coding elements in its output to increase output readability and facilitate user interaction. Additionally, AFLuent solves some of the problems present with Zoltar by existing as a Pytest plugin, which gives it easy access to program spectra such as test results and statement coverage.

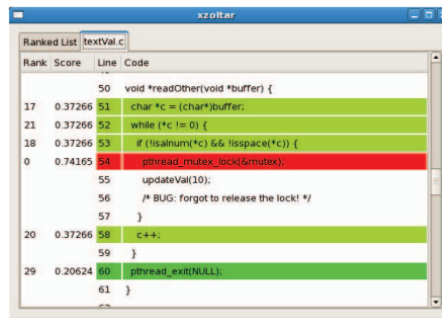


Figure 2.4: Example of Zoltar Interface [11]

In contrast to Zoltar, which was used to analyze programs written in C and C++, CharmFL, is a Python tool that performs SBFL to identify faults. Sarhan et al. [22] introduce CharmFL in a study that discusses its features and implementation. The study recognizes the need for SBFL tools to assist Python developers since it has become a very popular language. Additionally, it presents CharmFL as a plugin for the popular IDE and text editor PyCharm. Similar to AFLuent, it uses the Pytest framework to collect program spectra and calculates suspiciousness scores using Tarantula, Ochiai, and DStar approaches. Overall, CharmFL has many similarities with AFLuent, but it's also less accessible considering that it's a PyCharm plugin which is not used by every developer. Overall, the implementation of CharmFL provides an inspiration for AFLuent and encourages improvements where CharmFL may fall short.

2.3 Usability and Accessibility

Considering that the focus of AFLuent is not to simply be a fault localization tool but rather an accessible one for beginner and novice developers, it's important to better understand how the tool can be catered to that audience. This can be done in many different ways, one of which is to increase the clarity and verbosity of output messages from the tool. Instead of simply displaying the ranked scores of statements, it would be more user friendly to explain the meaning of the output to guide the user into beginning the debugging process. Kohn [14] explores the experience of beginners with Python errors with different severity and various Python interpreter error output. The results

```

7 def addToCart(product):
8     if(product not in cart.keys()):
9         cart[str(product)] = 1
10    else:
11        cart[str(product)] = cart[str(product)] + 2_# bug -> should be 1
12
13
14 def removeFromCart(product):
15     if(product in cart.keys()):
16         if(cart[str(product)] > 1):
17             cart[str(product)] = cart[str(product)] - 1
18         elif(cart[str(product)] == 1):
19             del cart[str(product)]
20     else:
21         print("Something's fishy")

```

Figure 2.5: Example interface of CharmFL [22]

confirm that more clear error messages tend to have a higher percentage of students finding and fixing the error. This connection between error output and the ability for beginner developers to fix faults is very crucial in the case of AFLuent. And while a user survey is out of scope of this research, Kohn provides encouragement to account for the different use cases in AFLuent and attempts to provide a clear output that describes the fault and guides the developer for the next step.

Another aspiration of AFLuent is to assist beginners in debugging their code in ways that go beyond simply looking at the suspiciousness ranking of elements. By identifying popular python errors in Python among beginners, cause of faults can more quickly be pointed out after statement ranking has been produced. These steps require additional analysis of the suspicious statements by analyzing their syntax to identify potential causes. The goal of AFLuent would then become more than simply locating the fault, but also giving an educated guess regarding the reason behind the error. Cosman et al. [8] create a tool named PABLO that uses a trained classifier to identify common bugs and faults in beginner written Python programs. PABLO is also evaluated using an empirical and human study to analyze how accurate it is in detecting mistakes and how helpful users find the tool. Overall, PABLO is found to be “helpful, providing high-accuracy fault localization that implicates the correct terms 59-77% of the time”. The ability of PABLO to describe the cause of error is very valuable to incorporate in AFLuent, especially that AFLuent is only able to conclude that there is an error and where it’s possibly located. PABLO provides another layer of fault localization that can only enhance the user experience.

Chapter 3

Method of Approach

This chapter describes the implementation of AFLuent and the experiment setup and execution process. More specifically, the reasoning behind design decisions and the result are the main focus. Additionally, charts and diagrams are used to demonstrate the algorithms, structure, and flow of execution.

3.1 Development Environment and Toolset

In order to begin discussing how AFLuent is implemented, a ground-up overview of the tools used and their roles is necessary to establish definitions and facilitate the understanding of how dependencies are connected. By being a Python package AFLuent can rely on a wide variety of helpful and popular tools. Some of the most important tools and dependencies are discussed below.

3.1.1 Poetry

Poetry is a Python virtual environment management tool that allows developers to set up an isolated environment for their projects. Furthermore, it manages the installation of Python dependencies on the virtualenv and updates them when necessary. Poetry has a crucial role in the implementation of AFLuent since it's used to make the development process simpler, its role also goes beyond that to the packaging and publishing of AFLuent to the Python Package Index (PyPI). Poetry's ability to abstract and simplify all the details in creating a Python package, makes it essential for development. However, from a user's perspective, Poetry is not necessary to run or use the tool for fault localization.

3.1.2 Pytest

Pytest is a Python testing framework used to write and execute a variety of tests, especially unit tests. Many developers in the community have contributed to Pytest through plugins that extend its functionality and allow it to accomplish new beneficial tasks. As mentioned previously, AFLuent is a Pytest plugin that adds automated fault localization features. By definition, AFLuent relies on Pytest in order to function

properly. More details on how AFLuent is integrated with pytest can be found in Section 3.2.

3.1.3 Coverage.py

Spectrum-based fault localization requires data on code coverage and test results in order to calculate and rank the suspicious elements in the code. Coverage.py[7] is a Python tool that provides an easy to use application programming interface to collect that data. The tool also provides various configurations for the user to skip certain files or directories from being considered. AFLuent relies on this tool to calculate what's known as per-test coverage. This data describes the lines of code covered by a single test case and organized in an accessible way to find out the number of passing and failing test cases that executed each line. Automated fault localization approaches require this information in order to calculate suspiciousness scores and Coverage.py can provide that relatively easily.

3.1.4 Radon

Radon is a Python tool used to calculate metrics of code complexity. Specifically, it's used in AFLuent to calculate cyclomatic complexity of some elements to break ties in rankings. When the user requests that AFLuent utilizes cyclomatic complexity to break ties, Radon is used to create a dataset of complexity scores. Since single lines cannot have a cyclomatic complexity score, they are assigned the score of the function they're a part of. Furthermore, the dataset is organized to facilitate efficient searching by the line number in question.

3.1.5 Libcst

In order to provide additional methods to break ties between element rankings, Libcst is used to create an abstract syntax tree of the code in question. This approach allows AFLuent to detect error prone syntax and formulate a score to use to break ties between lines if the need arises. Additional details on the use of Libcst are discussed in section 3.6: Tie Breaking.

3.2 AFLuent as a Pytest Plugin

Pytest provides simple and intuitive ways of extending its functionality through hooks. These hooks break up the standard Pytest execution steps and allow external code to run at these points. A plugin is essentially a collection of packaged hooks that fit in the workflow of Pytest. AFLuent makes use of five different hooks to implement automated fault localization. Figure 3.1 shows a general overview of the steps changed in the workflow of Pytest. Additionally, the section below describes in detail how each step was modified.

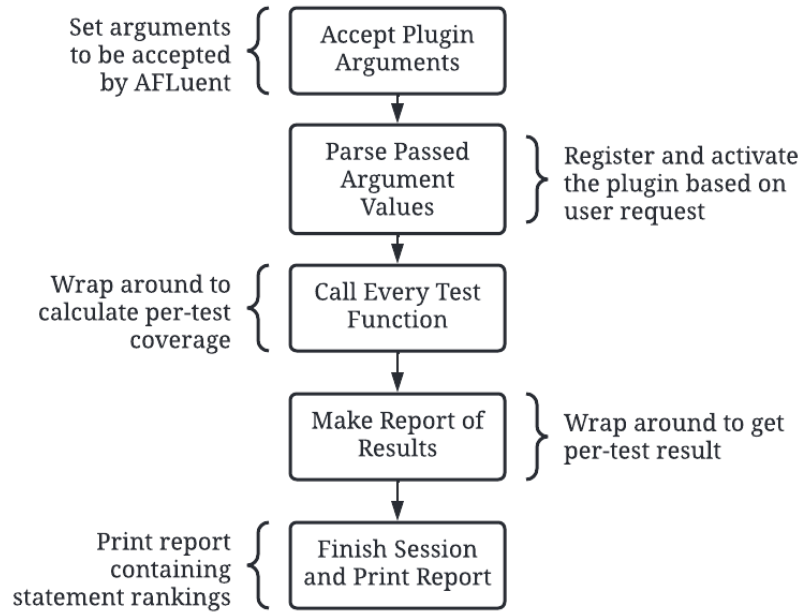


Figure 3.1: Pytest Tasks Flowchart

3.3 Installing AFLuent

As part of the goal of creating a novice friendly tool, it's very important that AFLuent can be installed and set up easily on any system with little to no complications. To achieve that, AFLuent is published to the Python Package Index (PyPI), which makes it installable through the `pip install afluent` command. Once this command runs successfully, AFLuent is automatically integrated with Pytest as a plugin and will be run with every pytest session when the user specifies. AFLuent's dependency on Coverage.py creates a small but avoidable conflict. In the case that AFLuent and another plugin that utilizes Coverage.py is active in the same Pytest session, various errors might occur where coverage information is not calculated correctly, thus affecting the automated fault localization process. This issue was prevalent when specifically using `pytest-cov` plugin, which produces a coverage report at the end of a Pytest session. AFLuent handles this problem by checking for the presence of other Coverage.py reliant plugins and warns the user. Due to this issue, AFLuent will not run in the session unless specified using the command-line arguments discussed in the following section.

3.3.1 Adding Command-Line Arguments

Pytest already supports a multitude of command line arguments that allow the user to pass configurations that change how the test suite is executed and reported. Similarly, AFLuent requires user passed arguments to complete a variety of tasks. The hook `pytest_addoption` allows adding new arguments in a fashion similar to the `argparse` Python library. The `argparse` library supports features that allow easy parsing and

validation of command-line arguments. Furthermore, it facilitates working with input with various data types. The added arguments accept values that specify the following:

- If AFLuent should be enabled for the current Pytest session.
 - Flags: `--afl-debug` or `--afl`
- The techniques for suspiciousness score calculations
 - `--tarantula`: enable fault localization using Tarantula approach
 - `--ochiai`: enable fault localization using Ochiai approach
 - `--ochiai2`: enable fault localization using Ochiai2 approach
 - `--dstar`: enable fault localization using DStar approach
 - `--dstar-pow`: used to set the power to use in the DStar approach. Defaults to 3
 - Multiple approaches can be used at the same time, however, results will be sorted by the approach passed first.
- The number of results to display in the report
 - `--afl-results`: number of top results to show in the terminal window. Defaults to 20
- The directories and files to be ignored while calculating coverage
 - `--afl-ignore`: Used to pass directories and files to ignore. Example: `tests/*` will ignore all files and directories inside the `tests` folder.
- The types of file reports to create after the Pytest session is over
 - `--report`: accepts `json` or `csv` and generates reports with the passed format.
 - `--per-test-report`: requires that a per-test coverage report is produced. This report is only generated in JSON format.
- The tie breaking approaches to resolve ties in rankings
 - `--tiebreaker`: allowed values are `random`, `cyclomatic`, `logical`, or `enhanced`
 - Defaults to `random` if no approach is passed.

3.3.2 Activating AFLuent

After arguments are passed, the next steps parse through some of them to check if AFLuent was enabled and to validate some of their values. The Pytest hook `pytest_cmdline_main` gives access to the collected configuration. In this hook, checks are conducted to see if there are other active plugins that utilize Coverage.py. These checks are necessary due to conflicts in collecting data when multiple plugins use Coverage.py simultaneously. The problem is resolved through warnings to the user that ask to disable these plugins in order for AFLuent to run properly. Lastly, if AFLuent is enabled, an `Afluent` object is initialized with the passed configuration and registered as a plugin for the current Pytest session.

3.3.3 Calculating Per-test Coverage and Test Result

Per-test coverage is defined here as the collection of lines executed by each individual test case. In order to collect this data, AFLuent adds a wrapper that executes code before and after each test case function is called. The `pytest_pyfunc_call` hook is used in this scenario. The execution steps are as follows:

1. `cov.start()` begins recording coverage
2. The hook yields back control to Pytest which calls the individual test case
3. `cov.stop()` stops recording coverage
4. The collected data is then organized in a simpler structure defined as the program spectra
5. The process repeats for every test case

Once per-test coverage is completed, the `pytest_runtest_makereport` hook is used to get the outcome of each test case and add it to the program spectra structure. There are three possible test outcome in Pytest, 'Passed', 'Failed', and 'Skipped'.

3.3.4 Reporting Results

The last step in AFLuent execution as part of Pytest is to report the fault localization outcome. The `pytest_sessionfinish` hook is used to detect the exit code of the session and display output on the console accordingly. An exit code of 0 means that all tests have passed and there is no need to perform fault localization, therefore, a message would display that to the user before finishing the Pytest run. On the other hand an exit code of 1, would indicate that at least one test case failed during the pytest session. In this scenario, AFLuent initializes a `Spectrum` object, which is responsible for parsing through the program spectra structure, ranking elements based on suspiciousness, and producing an output to the console.

3.4 AFLuent's Components

Considering all the steps involved in calculating and reporting suspiciousness of elements, AFLuent is implemented in an object oriented style that focuses on the readability and maintainability of the code. By splitting up AFLuent to a collection of objects organized in a hierarchy, the testing approach becomes more clear and debugging becomes easier. This section discusses the objects oriented structure starting with the least complex.

3.4.1 Line Object

A line is the simplest unit in a large project or program. With that in mind, a **Line** object describes the attributes and implements the functionalities associated with a single line in the context of automated fault localization. Important attributes of a line include identifiers such as the line number and file path it belongs to. Other line-specific information include a list of failed test cases that executed this line, and another for tests that passed. This information is crucial in calculating suspiciousness scores. These scores are also stored as attributes of each Line object. Another stored attribute is a collection of scores used to break ties between lines when their suspiciousness is the same. In general, these scores measure how error prone is the code with the current syntax structure. Further discussion on breaking ties using these scores can be found in Section 3.6 Tie breaking.

As for functions, a Line object contains an implementation of the four equations, Tarantula shown in Figure 1.3, Ochiai shown in Figure 1.4, Ochiai2 shown in Figure 2.3, and DStar shown in Figure 1.5, where suspiciousness scores are calculated and rounded up to four decimal places. The mathematical implementations of these functions also handle and account for division by zero. Depending on the equation, there could be multiple instances where division by zero is possible. The following subsections discuss how this issue was handled for each equation.

Division by Zero: Tarantula

In the Tarantula equation, division by zero occurs if either the number of total passing tests or number of total failing tests is zero. In the case that there are no failing test-cases in the whole test suite, then the suspiciousness of all covered statements is zero (not suspicious) since there was no fault. The other edge case occurs when there are no passing tests in the whole test suite, in which every covered line has the maximum suspiciousness of 1. It is important to acknowledge that the outcomes reached here only apply to code that has been covered by the test suite. Faulty lines, which are not covered by any test case will not be investigated since there is no data to calculate their suspiciousness. Using values plugged into Figure 1.3, the examples in Figure 3.2 and Figure 3.3 show the two possible cases where division by zero might occur in the Tarantula equation.

Figure 3.2 shows the case where there are zero total failed test cases, covering AND not covering the element, in which a score of 0 is assigned to the element. On the other

When $\mathbf{N}_{CF} = \mathbf{0}$ and $\mathbf{N}_{UF} = \mathbf{0}$

$$Tarantula(element) = \frac{\frac{\mathbf{0}}{\mathbf{0}}}{\frac{\mathbf{N}_{CS}}{\mathbf{N}_{CS} + \mathbf{N}_{US}} + \frac{\mathbf{0}}{\mathbf{0}}} \quad (3.1)$$

Figure 3.2: Tarantula Division by Zero Example 1

When $\mathbf{N}_{CS} = \mathbf{0}$ and $\mathbf{N}_{US} = \mathbf{0}$

$$Tarantula(element) = \frac{\frac{\mathbf{N}_{CF}}{\mathbf{N}_{CF} + \mathbf{N}_{UF}}}{\frac{\mathbf{0}}{\mathbf{0}} + \frac{\mathbf{N}_{CF}}{\mathbf{N}_{CF} + \mathbf{N}_{UF}}} \quad (3.2)$$

Figure 3.3: Tarantula Division by Zero Example 2

hand, Figure 3.3 shows the case where there are zero total passing test cases, in which a maximum score of 1 is assigned to the element.

Division by Zero: Ochiai

Division by zero occurs in the Ochiai formula when there is no test coverage information for a line or when the total number of failed tests is zero. The latter case indicates that the line is not suspicious since it did not cause any failures, therefore, zero is returned as the suspiciousness score. However, the former case is not considered at all since AFLuent does not include lines with no coverage information. Figure 3.4 and Figure 3.5 demonstrates that equation after plugging in the zero values from 1.4

When $\mathbf{N}_{CF} = \mathbf{0}$ and $\mathbf{N}_{UF} = \mathbf{0}$

$$Ochiai(element) = \frac{\mathbf{0}}{\sqrt{(\mathbf{0}) \cdot (\mathbf{0} + \mathbf{N}_{CS})}} \quad (3.3)$$

Figure 3.4: Ochiai Division by Zero Example 1

When $\mathbf{N}_{CF} = \mathbf{0}$ and $\mathbf{N}_{CS} = \mathbf{0}$

$$Ochiai(element) = \frac{\mathbf{0}}{\sqrt{(\mathbf{0} + \mathbf{N}_{UF}) \cdot (\mathbf{0})}} \quad (3.4)$$

Figure 3.5: Ochiai Division by Zero Example 2

Figure 3.4 is an example of when there are zero total failed test cases, resulting in a zero suspiciousness score. However Figure 3.5 shows an example where there are no failed or successful tests that cover the line. This scenario does not occur in AFLuent, which only looks at lines that have some coverage data through passing or failing tests. Therefore, it wasn't necessary to handle this possibility.

Division by Zero: Ochiai2

Considering the number of terms in the denominator of the formula to calculate score using Ochiai2, there are technically four instances where the denominator can be zero. In the first case, no tests, failing or passing, cover the statement. More formally when $\mathbf{N}_{CF} + \mathbf{N}_{CS} = \mathbf{0}$. Figure 3.6 shows the equation with the zero plugged in. While this is an issue mathematically, it does not occur in AFLuent because the tool only analyzes elements that have been covered at least once by a failing or a passing test. Therefore, there is no need to handle this possibility.

$$Ochiai2(element) = \frac{\mathbf{N}_{CF} \cdot \mathbf{N}_{US}}{\sqrt{(\mathbf{0}) \cdot (\mathbf{N}_{US} + \mathbf{N}_{UF}) \cdot (\mathbf{N}_{CF} + \mathbf{N}_{UF}) \cdot (\mathbf{N}_{CS} + \mathbf{N}_{US})}} \quad (3.5)$$

Figure 3.6: Ochiai2 Division by Zero Example 1

Moving on to the next possibility of division by zero, which occurs when $\mathbf{N}_{US} + \mathbf{N}_{UF} = \mathbf{0}$. In this case, there are no tests that do not cover the element being inspected, or in other words, all the tests (both failing and passing) do cover this element. Considering that \mathbf{N}_{US} is a zero in this case, it would cause the numerator to be a zero and result in a zero divided by zero. Due to both numerator and denominator being zero, the suspiciousness score output is also returned as zero.

$$Ochiai2(element) = \frac{\mathbf{N}_{CF} \cdot \mathbf{N}_{US}}{\sqrt{(\mathbf{N}_{CF} + \mathbf{N}_{CS}) \cdot (\mathbf{0}) \cdot (\mathbf{N}_{CF} + \mathbf{N}_{UF}) \cdot (\mathbf{N}_{CS} + \mathbf{N}_{US})}} \quad (3.6)$$

Figure 3.7: Ochiai2 Division by Zero Example 2

In the third possibility where a division by zero could happen, the third term $\mathbf{N}_{CF} + \mathbf{N}_{UF} = \mathbf{0}$. This would mean that there are a total of zero failed tests that both cover and not cover the element. Since AFLuent will only run when there is at least one failure, this situation is not possible during the tool's runtime. Figure 3.8 shows the zero as part of the full equation.

The last possibility where division by zero in Ochiai2 can occur is when $\mathbf{N}_{CS} + \mathbf{N}_{US} = \mathbf{0}$, in which there are zero total successful test cases. In this situation, there is no indicator of correct functionality and therefore every element is highly suspicious.

$$Ochiai2(element) = \frac{\mathbf{N}_{CF} \cdot \mathbf{N}_{US}}{\sqrt{(\mathbf{N}_{CF} + \mathbf{N}_{CS}) \cdot (\mathbf{N}_{US} + \mathbf{N}_{UF}) \cdot (\mathbf{0}) \cdot (\mathbf{N}_{CS} + \mathbf{N}_{US})}} \quad (3.7)$$

Figure 3.8: Ochiai2 Division by Zero Example 3

A score of 1.0 will always be returned in this scenario. Figure 3.9 shows the equation with the zero in the denominator.

$$Ochiai2(element) = \frac{\mathbf{N}_{CF} \cdot \mathbf{N}_{US}}{\sqrt{(\mathbf{N}_{CF} + \mathbf{N}_{CS}) \cdot (\mathbf{N}_{US} + \mathbf{N}_{UF}) \cdot (\mathbf{N}_{CF} + \mathbf{N}_{UF}) \cdot (\mathbf{0})}} \quad (3.8)$$

Figure 3.9: Ochiai2 Division by Zero Example 4

Division by Zero: DStar

In the DStar equation, division by zero takes place only in one case. When the number of passing test cases that cover the line AND the number of failed test cases that do not cover the line are both zero, the denominator evaluates to zero. This translates to the following: if there are no passing tests executing this line and no failing test executing other lines only, then this line should have the maximum suspiciousness score possible. However, since DStar has a numerator raised to a power set by the user, it has no numerical upper limit on the resulting score. To resolve this issue, the maximum integer value is assigned as the suspiciousness score of this line. Figure 3.10 shows an example of this scenario after plugging in values from Figure 1.5

When $\mathbf{N}_{CS} = \mathbf{0}$ and $\mathbf{N}_{UF} = \mathbf{0}$

$$Dstar(element) = \frac{(\mathbf{N}_{CF})^*}{\mathbf{0}} \quad (3.9)$$

Figure 3.10: DStar Division by Zero Example

3.4.2 ProjFile Object

ProjFile objects are designed to contain attributes that describe whole files. Additionally, they support functionality that apply to these files. The most important attribute of these objects is the `lines` instance variable, which stores a dictionary of contents of the file. Specifically, the keys in this dictionary are line numbers in the file and the values are the Line objects discussed previously. In addition to storing this data,

ProjFile implements an intuitive way to update it with a new result of a test case. `update_file` function takes in a list of numbers of covered lines executed by a test case as well as the result and the name of that test case. Following that, it iterates through the covered line numbers and updates the Line objects stored in the ProjFile dictionary. This approach creates a hierarchy between the objects by nesting Line objects inside of ProjFile objects.

3.4.3 Spectrum Object

This object contains some of the core functionalities that organizes the collected data and displays it to the console. Similar to how a ProjFile object stores a dictionary of line number and Line object pairs, Spectrum takes nesting a step further and stores a dictionary of file paths and ProjFile objects as key-value pairs. Additionally, it keeps track of running totals of passed, failed, and skipped test cases, which are used to calculate suspiciousness scores. Lastly, it's responsible for sorting Line objects in a descending order from highest suspiciousness scores. Once this task is completed, the Spectrum object produces a table formatted report with color coded results that demonstrate to the user the outcome of fault localization.

3.4.4 Afluent Object

While this Object does not implement any functionality regarding the calculation or sorting of suspiciousness scores, it plays an important role in integrating this tool with Pytest. By containing several of the Pytest hooks mentioned prior, this object is registered as the plugin when AFLuent is enabled through command line arguments. Following the end of the Pytest session, the AFLuent object initializes a Spectrum Object and forwards the arguments collected from the console command. Following that, it calls the Spectrum object to produce and print its report. Overall, this object acts as a pipeline that passes coverage information and command line arguments collected from the Pytest session to the infrastructure responsible for performing fault localization.

3.4.5 Objects Overview

Fig 3.11 provides a visual simplification of the different components of AFLuent and an overview of their roles in the functioning of the tool. Overall the nested structure creates several layers that facilitate development by isolating the different components and hiding unnecessary information from other objects in the hierarchy. By following these structures, unit tests can be written much easier and debugging becomes a simpler task.

3.5 AFLuent's Output

While the main purpose of AFLuent is to produce a report of the most suspicious statements, there are many additional features that can increase the readability and

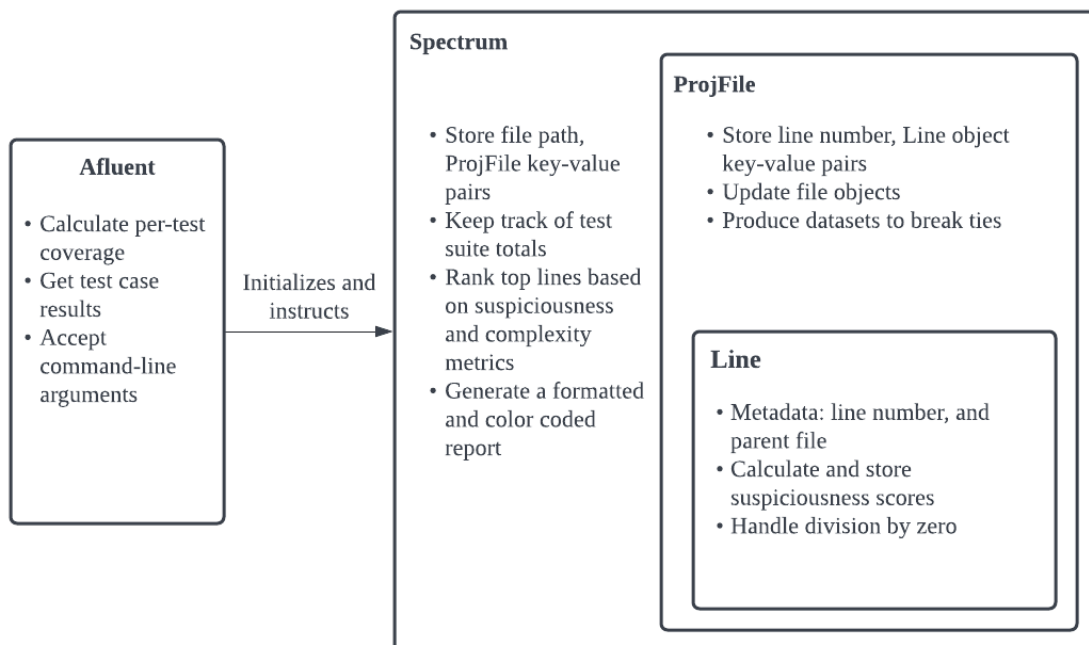


Figure 3.11: AFLuent Object Structure

user friendliness of the report. The input and workflow of AFLuent was discussed in the previous sections, this section focuses on the outcome and discusses how the user should interact with it.

3.5.1 Feedback Messages

Throughout the execution of AFLuent, unexpected things can occur. And while Python and Pytest error messages can be produced, AFLuent also shows console output that gives a hint regarding what went wrong. There are three tiers of messages produced by AFLuent, Error, Warning, and Success messages, each color coded using red, orange, and green respectively. Each one is designed to stand out to the user and provide a clear and concise description of the scenario.

Success Messages

This message is produced in the case that the test suite passes with no error or failures. Using bright green highlighted text with bold white letters, the message displays: **All tests passed, no need to diagnose using AFLuent**. Figure 3.12 demonstrates the success message output when AFLuent is run on the project's test suite.

```

noboshe@noboshe-Yoga-6:~/AFLuent/AFLuent
~/A/AFLuent > main !1 ?3 poetry run pytest --afl --afl-ignore tests/*
Test session starts (platform: linux, Python 3.9.6, pytest 6.2.5, pytest-sugar 0.9.4)
rootdir: /home/noboshe/AFLuent/AFLuent
plugins: sugar-0.9.4, afluent-0.1.3, cov-3.0.0
collecting ...
tests/test_line.py 39%
tests/test_proj_file.py 58%
tests/test_spectrum_parser.py 100%

All tests passed no need to diagnose using AFLuent.

Results (0.27s):
 36 passed
~/A/AFLuent > main !1 ?3

```

Figure 3.12: Example of Success Message

Error Messages

Unlike success messages, error messages are produced when things aren't going as expected. However, in this case, they aren't meant to denote complete failure of AFLuent, but rather to communicate a major alert to the user. This type of message is produced when the Pytest session exists with status code 1, which occurs when at least one test fails. Figure 3.13 shows the console output when a test failure was purposefully introduced. A bright red highlighted message with bold white letters indicate that fault localization using AFLuent is underway.

Warning Messages

The last type of messages produced during the runtime of AFLuent are warning messages. They communicate important information to the user but do not signify a major problem. Warning messages are displayed in three possible scenarios. In the case that the user passes the `-x` or `--maxfail` command line arguments to Pytest, the session can stop at the first test failure. This is problematic from a fault localization standpoint because it prevents further collection of coverage information and test outcomes from the rest of the test suite. In order to let the user be aware of this issue, the warning message shown in Figure 3.14 is displayed. Regardless, AFLuent will continue running with standard behavior, however, results are very likely to be inaccurate.

Another warning message is displayed when AFLuent is installed to the current Python environment but not enabled by the user through the `--afl` or `--afl-debug` flags. This message serves as a reminder to the user to enable the plugin if they're interested in utilizing fault localization. The message is shown in Fig 3.15.

```
noboshe@noboshe-Yoga-6:~/AFLuent/AFLuent
"sample_testcase",
]
covered_lines = [5, 6]
test_projfile.update_file(covered_lines, "skipped", "sample_testcase")
> assert test_projfile.lines[5].skipped_by == [
    "sample_testcase2",
    "sample_testcase3",
    "sample_testcase4",
    "sample_testcase",
]
E      AssertionError: assert ['sample_test...ple_testcase'] == ['sample_test..
.ple_testcase']
E          At index 2 diff: 'sample_testcase' != 'sample_testcase4'
E          Use -v to get the full diff

tests/test_proj_file.py:74: AssertionError

tests/test_proj_file.py x✓✓ 58%
tests/test_spectrum_parser.py ✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓ 100%
```

Failing tests detected. Diagnosing using AFLuent...

===== AFLuent Report =====

=====

Figure 3.13: Example of Error Message

```
noboshe@noboshe-Yoga-6:~/AFLuent/AFLuent
~ / A / AFLuent > poetry run pytest --afl --afl-ignore tests/* -x
Exit after failure detected. AFLuent gives more accurate results if the full test suite was ran.
Consider removing '-x' and/or '--maxfail' from CLI arguments.

Test session starts (platform: linux, Python 3.9.6, pytest 6.2.5, pytest-sugar 0.9.4)
rootdir: /home/noboshe/AFLuent/AFLuent
plugins: sugar-0.9.4, afluent-0.1.3, cov-3.0.0
collecting ...
tests/test_line.py 39%
tests/test_proj_file.py 58%
tests/test_spectrum_parser.py 100%

All tests passed, no need to diagnose using AFLuent.

Results (0.29s):
 36 passed
```

Figure 3.14: Maxfail Warning Message


```

noboshe@noboshe-Yoga-6:~/AFLuent/AFLuent
~ / A / AFLuent > main !2 ?2 poetry run pytest

AFLuent is disabled. Enable AFLuent by adding --afl or --afl-debug to the test command.

Test session starts (platform: linux, Python 3.9.6, pytest 6.2.5, pytest-sugar 0.9.4)
rootdir: /home/noboshe/AFLuent/AFLuent
plugins: sugar-0.9.4, afluent-0.1.3, cov-3.0.0
collecting ...
tests/test_line.py 39% ██████████
tests/test_proj_file.py 58% ██████████
tests/test_spectrum_parser.py 100% ██████████

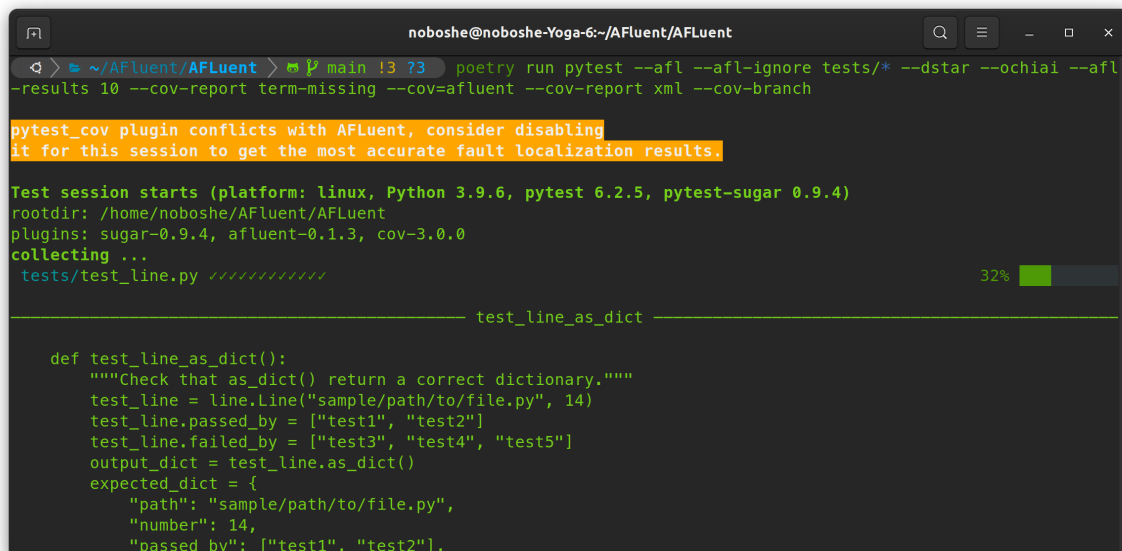
Results (0.07s):
 36 passed
  
```

Figure 3.15: AFLuent Disabled Warning Message

The third and last case that causes a warning message takes place when a potential conflict with Coverage.py is detected. In the possibility that another coverage collecting Pytest plugin is enabled in the same session as AFLuent, coverage data could be mixed up leading to inaccurate results or internal errors in Pytest. To avoid that, AFLuent scans the enabled plugins for the current session and checks if `pytest-cov` is running. If so, a warning message is displayed but the session continues as expected. It's recommended that when running AFLuent, that `pytest-cov` is disabled from the command line. Moreover, when running `pytest-cov`, it's safer to disable AFLuent. This can be done by adding `-p no:pytest-cov` to disable `pytest-cov` or `-p no:afluent` to disable AFLuent. There could potentially be many other Pytest plugins that conflict with AFLuent, however, `pytest-cov` is the only known one thus far.

3.5.2 Console Report

In the case that AFLuent was enabled for the Pytest session and at least one failure occurred, a report is generated and printed to the console to help the user look for the error. Based on the inputs passed in the command-line arguments, this report can be structured in many different ways. Generally, the report is a table with rows containing the path to the file, the line number, and suspiciousness score(s). The number of rows depends on the value of `--afl-results` argument, which is set to 20 by default. Additionally, the user can determine one or many equations to use for suspiciousness score calculations, where the results will be sorted using the first one mentioned in the command. For example, the command `poetry run pytest -afl -afl-ignore tests/* -ochiai -dstar -afl-results 10` will display the top 10 results sorted by the most suspicious using the Ochiai equation. However, the DStar suspiciousness



```
noboshe@noboshe-Yoga-6:~/AFLuent/AFLuent
> ~/AFLuent/AFLuent > main !3 73 poetry run pytest --afl --afl-ignores tests/* --dstar --ochiai --afl
--results 10 --cov-report term-missing --cov=afluent --cov-report xml --cov-branch

pytest_cov plugin conflicts with AFLuent, consider disabling
it for this session to get the most accurate fault localization results.

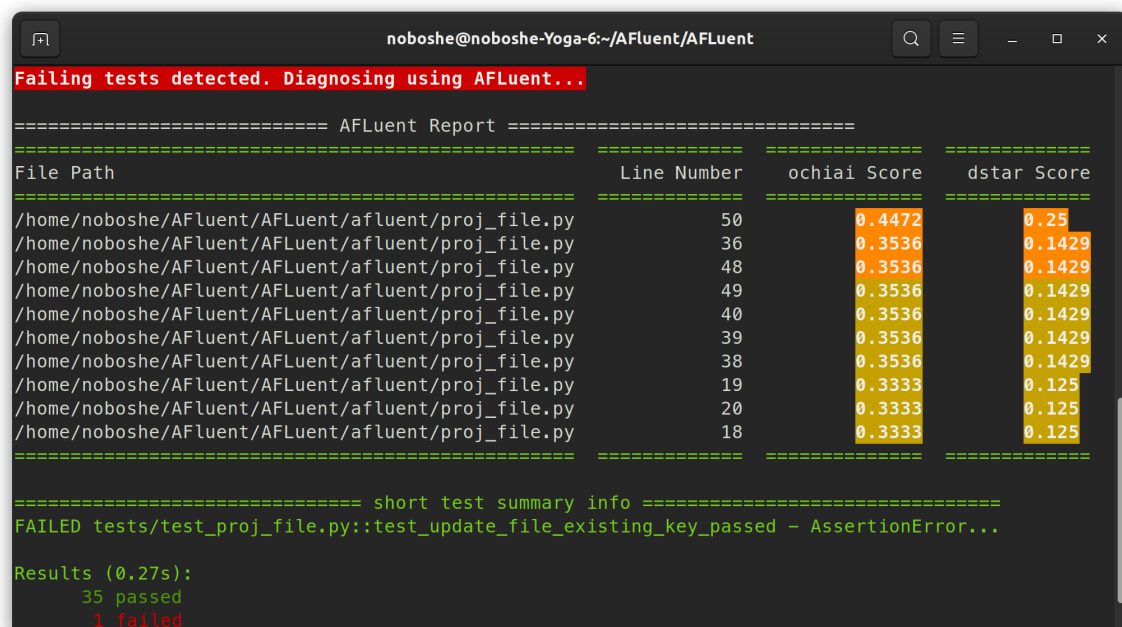
Test session starts (platform: linux, Python 3.9.6, pytest 6.2.5, pytest-sugar 0.9.4)
rootdir: /home/noboshe/AFLuent/AFLuent
plugins: sugar-0.9.4, afluent-0.1.3, cov-3.0.0
collecting ...
tests/test_line.py ////////////////////////////////// 32%

----- test_line_as_dict -----

def test_line_as_dict():
    """Check that as_dict() return a correct dictionary."""
    test_line = line.Line("sample/path/to/file.py", 14)
    test_line.passed_by = ["test1", "test2"]
    test_line.failed_by = ["test3", "test4", "test5"]
    output_dict = test_line.as_dict()
    expected_dict = {
        "path": "sample/path/to/file.py",
        "number": 14,
        "passed_by": ["test1", "test2"],
```

Figure 3.16: Conflicting Plugin Warning Message

will also be displayed. Figure 3.17 shows an example of the report when running the AFLuent test suite after introducing a bug in the code.



```
noboshe@noboshe-Yoga-6:~/AFLuent/AFLuent
Failing tests detected. Diagnosing using AFLuent...

===== AFLuent Report =====
=====
File Path                               Line Number   ochiai Score   dstar Score
=====
/home/noboshe/AFLuent/AFLuent/afluent/proj_file.py 50           0.4472         0.25
/home/noboshe/AFLuent/AFLuent/afluent/proj_file.py 36           0.3536         0.1429
/home/noboshe/AFLuent/AFLuent/afluent/proj_file.py 48           0.3536         0.1429
/home/noboshe/AFLuent/AFLuent/afluent/proj_file.py 49           0.3536         0.1429
/home/noboshe/AFLuent/AFLuent/afluent/proj_file.py 40           0.3536         0.1429
/home/noboshe/AFLuent/AFLuent/afluent/proj_file.py 39           0.3536         0.1429
/home/noboshe/AFLuent/AFLuent/afluent/proj_file.py 38           0.3536         0.1429
/home/noboshe/AFLuent/AFLuent/afluent/proj_file.py 19           0.3333         0.125
/home/noboshe/AFLuent/AFLuent/afluent/proj_file.py 20           0.3333         0.125
/home/noboshe/AFLuent/AFLuent/afluent/proj_file.py 18           0.3333         0.125
=====

===== short test summary info =====
FAILED tests/test_proj_file.py::test_update_file_existing_key_passed - AssertionError...

Results (0.27s):
  35 passed
   1 failed
```

Figure 3.17: Example Report Using Ochiai and DStar

In addition to printing the report, AFLuent color codes elements based on their rankings. The four possible colors are green, yellow, orange, and red, where each one represents the likelihood of finding the fault at that line. Suspiciousness scores

highlighted using green signify that the corresponding element is safe and has a zero suspiciousness score. On the opposite end of the spectrum, scores are highlighted with red when they equal the maximum suspiciousness score possible (usually 1), the color indicates that these elements are extremely likely to be the ones causing the fault. For the remaining results, the top 20% are highlighted using orange to show that they are at risk of being faulty. The remaining non-zero elements are highlighted using yellow. Figure 3.18 shows an example of how safe statements are displayed in the report.

```

noboshe@noboshe-Yoga-6:~/AFLuent/AFLuent
Failing tests detected. Diagnosing using AFLuent...

===== AFLuent Report =====
File Path                               Line Number  dstar Score  ochiai Score
=====
/home/noboshe/AFLuent/AFLuent/afluent/main.py      164          0.0278      0.1644
/home/noboshe/AFLuent/AFLuent/afluent/main.py      163          0.0278      0.1644
/home/noboshe/AFLuent/AFLuent/afluent/line.py       29           0           0
/home/noboshe/AFLuent/AFLuent/afluent/spectrum_parser.py  87           0           0
/home/noboshe/AFLuent/AFLuent/afluent/line.py      153           0           0
/home/noboshe/AFLuent/AFLuent/afluent/line.py      129           0           0
/home/noboshe/AFLuent/AFLuent/afluent/line.py       63           0           0
/home/noboshe/AFLuent/AFLuent/afluent/spectrum_parser.py  94           0           0
/home/noboshe/AFLuent/AFLuent/afluent/line.py       85           0           0
/home/noboshe/AFLuent/AFLuent/afluent/line.py       26           0           0
=====

===== short test summary info =====
FAILED tests/test_line.py::test_something - assert False

Results (0.29s):
  36 passed
   1 failed

```

Figure 3.18: Example Report With Safe Statements

3.5.3 File Report

Aside from the reports produced in the console, AFLuent can create report files of the rankings in both JSON and CSV formats. Additional reporting of per-test coverage is also possible. The command line arguments `-report` and `-per-test-report` enable this feature. Producing these reports facilitates the evaluation of AFLuent by allowing the storage of rankings in a file for further analysis later on. In general, the standard report contains the rankings of the lines using the approach requested by the user. Additionally, other columns in the report contain the scores of all the other approaches for comparison. On the other hand, the per-test report is a dictionary structured such that the main keys are the names of the different test cases in the test suite and the value is an object that stores the lines executed by that test case as well as the outcome of the test.

3.6 Tie Breaking

Ties are a frequent issue in spectrum-based fault localization. They often cause issues in identifying the source of the fault and render the scores useless. To mitigate this

problem, AFLuent implements three different approaches to resolve the inevitable ties in suspiciousness scores. These techniques can hopefully provide another metric to break the tie between two elements.

3.6.1 Random

Random tie breaking is the baseline approach for dealing with ties in suspiciousness scores. Other approaches are compared to random tie breaking in order to detect if there are any improvements. In random tie breaking, statements are ranked in a descending order by the chosen suspiciousness score first, however, the order between tied elements is random. This could lead to different rankings for each run of the tool.

3.6.2 Cyclomatic Complexity

One of the available ways to break ties between elements is to use cyclomatic complexity as a secondary score to consider when sorting. This score is proposed by McCabe [17] and can be easily calculated using the Radon library. It measures the number of available paths that execution could go through in a function. Since this type of score only applies to whole functions and not to individual elements, lines inherit the cyclomatic complexity score of the function they live in when being ranked. Essentially, if a function has a cyclomatic complexity score of 7, then all statements within the function also have that score. While this can help break many ties, it's not effective in breaking ties when the tied elements exist in the same function. However, further empirical evaluation of its performance will be discussed in the evaluation section.

3.6.3 Mutant Density: Logical Set

Another metric used to break ties between suspicious statements is mutant density [20]. More specifically, this score indicates how error prone the statement is by calculating the number of all possible mutants. For example, a statement that has many mathematical and logical operators to perform a calculation is more error prone than a statement that only has one or two of these operations. With that in mind, AFLuent uses this information to break ties between statements that have the same suspiciousness score but are syntactically different.

In the implementation of this approach, the focus was on mutants in logical operations such as arithmetic, boolean, comparison operators, and others. Table 3.1 includes all the possible mutants which are considered with support from Libcst. The number of these operators is calculated only for simple statements as defined by Libcst, therefore, it does not include `if` statements, function definition, `while` loops and so on.

3.6.4 Mutant Density: Enhanced Set

This approach to tie breaking also uses mutant density to evaluate how error prone a statement is. However, it seeks to provide a more holistic metric that also considers how error prone the constructs that a statement is nested in. For example, a statement inside a multi-level `if` statement, which is also nested in a loop, is more error prone than

Unary	Boolean	Binary	Comparison	Augmented Assignment
BitInvert	And	Add	Equal	AddAssign
Not	Or	BitAnd	GreaterThan	BitAndAssign
Minus		BitOr	GreaterThanEqual	BitOrAssign
Plus		BitXor	In	BitXorAssign
		Divide	Is	DivideAssign
		FloorDivide	LessThan	FloorDivideAssign
		LeftShift	LessThanEqual	LeftShiftAssign
		MatrixMultiply	NotEqual	MatrixMultiplyAssign
		Modulo	IsNot	ModuloAssign
		Multiply	NotIn	MultiplyAssign
		Power		PowerAssign
		RightShift		RightShiftAssign
		Subtract		SubtractAssign

Table 3.1: Mutants in the Logical Set

a statement which is outside these constructs. Since there is more room for errors in loops and if statement conditions, a statement nested in them takes on this risk of error. In order to measure this score, the list of mutants used in the logical set was extended to include additional constructs that might contain the error. Additionally, the tiebreaker looks through and scores each block that the statement is nested in. Table 3.2 shows the additional mutants that are looked for in the enhanced set. Additionally, Table 3.3 discusses how a score is assigned to each construct while parsing through the syntax tree and assessing how error prone a statement is. All of these approaches are used to calculate a score that gets used in breaking ties of suspicious statements while ranking. Figure 3.19 shows how each construct score is used in generating the final score of a statement.

Mutant	Description
Annotated Assign	Assignment to a variable that contains type hints
Assign	Simple assignment to a variable using the single equal sign
Function Call	A call to any function
Subscript	Indexing a list, dictionary, tuple or an object using brackets
Literal Tuple	Literal constructed tuple using parenthesis
Literal List	Literal constructed list using brackets
Literal String	Literal constructed string using quotation marks
Literal Dictionary	Literal constructed tuple using curly braces
Literal Integer	Literal integer (not a variable that stores one)
Literal Float	Literal float (not a variable that stores one)

Table 3.2: Mutants in the Enhanced Set

Construct	Scored Using
Function Definition	Number of arguments in the function signature
If statement	Number of enhanced list mutants in the test condition
Simple statement	Number of the enhanced list mutants in the statement
While loop	Number of enhanced list mutants in the test condition
For loop	Number of enhanced list mutants in the iterator and iterable
With statement	Number of the enhanced list mutants in the statement

Table 3.3: Construct Scoring in Enhanced Set

$$Score(statement) = \frac{M_{statement} + \frac{M_{c1} + M_{c2} + \dots + M_{ci}}{i}}{2} \quad (3.10)$$

Figure 3.19: Enhanced Score Equation

	Line	Cyclomatic	Logical	Enhanced
1:	def some_func(arg1, arg2, arg3, arg4: str):	3	0	2.0
2:	int1 = arg1 * arg2 / arg3	3	2	3.5
3:	int2: str = arg4 + "hello world"	3	1	3.5
4:	my_list = ["some item"]	3	0	3.5
5:	while int1 < 30:	3	0	1.5
6:	my_list.append(int2)	3	0	2.0
7:	if len(my_list) > 15:	3	0	1.5
8:	print("done")	3	0	2.5
9:	break	3	0	1.5
10:	with open("my_file.txt", "w+") as outfile:	3	0	1.0
11:	outfile.write(str(my_list))	3	0	2.0
12:		0	0	0
13:		0	0	0
14:	def some_other_func(arg1, arg2):	4	0	1
15:	if arg1 > arg2:	4	0	0.75
16:	if 2 * arg2 > arg1:	4	0	1
17:	print(arg1)	4	0	1.5
18:	elif 3 * arg2 > arg1:	4	0	1.125
19:	print(arg2)	4	0	1.625
20:	else:	4	0	0.75
21:	print(arg1 * 3.5)	4	1	2.25

Table 3.4: Tiebreaking Score Examples

3.6.5 Tiebreaking Overview

Using all the tie breaking approaches discussed previously, this subsection provides an overview and an example that demonstrates tie breaking on a sample program. Table

3.4 shows a sample program that implements two functions with some conditional logic and simple mathematical operations. It also contains three columns with each scoring approach. Starting with the cyclomatic scores, one can see that all statements in a function contain the same score. This number represents the cyclomatic complexity of the function resulting from using Radon. While this approach is expected to break ties, it could potentially fail at doing so if the tied statements are in the same function. As for the logical approach scoring, it only applies to simple statements such as lines 2,3,4,6,8,11,17,19, and 21. Using this approach, mutant density is calculated using logical mutants only. For example, since line 2 contains the two binary operators `*` and `/` for multiplication and division, the mutant density of the statement is 2, which is also its score. Note that variable assignment and literals do not affect logical tie breaking scores. Lastly, enhanced tie breaking takes into consideration a more extensive list of possible mutants. Additionally, it analyzes a wider range of statements that includes `if` statements, `while` and `for` loops, as well as function definitions. Using some examples from the sample program in Table 3.4, the enhanced score for line 3 would be calculated as follows:

$$\frac{3 + (4 / 1)}{2}$$

Where 3 is the number of enhanced list mutants in the line (annotated assign, plus, literal string), 4 is the total number of enhanced list mutants the statement is nested in, and 1 is the total number of constructs that the statement is nested in. Using another example on line 8, the score would be calculated as follows:

$$\frac{2 + (9 / 3)}{2}$$

Where 2 is the number of possible mutants in the statement itself (function call, and literal string), 9 is the total number of possible mutants in constructs that the statement is nested in (`len()`, `>`, `15`, `<`, `30`, four function arguments) and 3 is the number of constructs that the statement is nested in (`if` statement, `while` loop, and function). Using the same process the enhanced score is calculated for all the lines in the program and used for breaking ties in suspiciousness scores.

Chapter 4

Experimental Results

4.1 Experimental Design

Following the completion of AFLuent, an evaluation step is needed to understand how accurate the tool is in localizing faults. This section discusses the design of this experiment including early steps in choosing and filtering a sample to test AFLuent on.

4.1.1 Approach Overview

In order to evaluate AFLuent, several prerequisites are needed that enable collecting data for analysis. The primary requirement is a collection of Python programs, which are susceptible to becoming faulty. Additionally, this collection's complexity must be comparable to code typically written by novice developers, which AFLuent targets. Another crucial requirement before evaluation can begin is a test suite for the selected code python code. The test suite must be executable by Pytest and covers as much as possible of the code to maximize the ability to locate faults. Assuming that these prerequisites are met and are available, the evaluation of AFLuent can then be generally described by these four main steps:

1. Introduce a bug/fault in the code under test
2. Run the test suite with AFLuent
3. Produce a report of suspicious statements as ranked by AFLuent
4. Asses how far the in the produced ranking report was the location of the introduced bug

While these steps are great for planning and describing the intention in evaluating AFLuent, they fall short of providing a clear and detailed plan. The sections below will expand on the points discussed here, elaborate on how the prerequisites are met, and describe the systematic and automated approach in evaluation.

4.1.2 Research Questions

Before discussing the evaluation process, it's important to clearly state the questions to answer. Previously, Section 1.4 brought up a few research questions concerning the implementation and evaluation of AFL in Python. Related Work and Methods section addressed *RQ1.1* as well as *RQ2*, however, the answer *RQ1.2* remains unclear. While *RQ1.2* generally involved the efficiency and accuracy of AFLuent, there was no mention of the metrics or the comparisons needed to achieve an answer. In general, there are sixteen different approaches that will be evaluated and compared. For every AFL equation used, Tarantula, Ochiai, Ochiai2, and Dstar, there are four tie breaking approaches. Each equation-tiebreaker pair will be evaluated on the same dataset, where the resulting rankings will be used to produce a score to assess how close the produced rankings are to localizing the fault correctly. In addition to this score, the time taken to run each approach will be recorded to compare their time overhead.

4.1.3 Choosing a Sample

Since AFLuent is targeted towards novice developers with little experience programming in Python, the evaluation strategy should reflect that by running AFLuent on novice written code. However, considering that this study does not involve human subjects, novice developers cannot be asked to write code for the purposes of this evaluation. To simulate novice written code as best as possible, the GitHub repository `TheAlgorithms/Python`[4] is used as the sample. This educational repository is a collection of projects that implement several fundamental algorithms in Python that range from ciphers, sorting, graphs, string operations to financial and physics equations. It resembles novice written code because, in a sense, beginner developers write these programs while learning the basics of programming. However, a problem with using this code is the lack of unit tests that `pytest` can run, since most tests are written using Python's `doctest`. With that in mind, there are Python tools such as `Pynguin`[16] that can be used to automatically generate a test suite for these projects. With the many limitations in `Pynguin` and the long waiting times to generate tests, some filtering of the projects is needed.

Filtering Sample

There are many issues that could arise when automatically generating a test suite using `Pynguin`. To mitigate these problems, the following criteria was used to eliminate entire projects, delete modules, or trim some code.

- Projects that import external modules: In order to facilitate and expedite generating a test suite using `Pynguin`, several projects that use external packages were removed. These packages include `scikit-learn`, `Tensorflow`, `Matplotlib`, `Sympy`, and `PIL`. Generating data and creating unit tests for projects that use these packages can be difficult because they are time consuming or simply cannot be tested due to their graphical output.
- Functions that do not take input or return no results: some implemented functions do not accept arguments as input. This makes them difficult to test because

no data can be passed to them. Additionally, functions that do not return a result are also difficult because there is no output to be checked. Functions that are part of an object oriented structure that change the state of an object without returning a value were kept since their results can be tested. However, other functions such as `main()` which read input from the console and controlled the flow were removed.

- Functions with missing type hints in signature: generating data for functions with unknown input type would be challenging for Pynguin to perform. Therefore, these functions were removed. In some instances where the documentation explicitly stated the type of input for the function, type hints were manually added to avoid the removal of the function. This was especially frequent in the sorting function, in which the input was specified as integer values.
- Code snippets under `if __name__ == "__main__":`: In most instances the code under this if statement either ran the doctest tests, or called a separately implemented main function. Since there is no use for these snippets and to clean up the sample, these snippets were removed.

arithmetic_analysis	audio_filters	backtracking	bit_manipulation
blockchain	boolean_algebra	cellular_automata	ciphers
compression	conversions	data_structures	divide_and_conquer
dynamic_programming	electronics	financial	genetic_algorithm
geodesy	graphs	greedy_methods	hashes
knapsack	linear_algebra	maths	matrix
physics	scheduling	searches	sorts
strings			

Table 4.1: Remaining Projects Prior to Test Generation

Following the initial phase of filtering the codebase, general statistics and observations were recorded for the remaining sample thus far. Table 4.1 shows the name of the remaining project. Additionally, SLOCcount was used to calculate the number of non-comment lines of code included in the sample. The output from the tool is shown in Figure 4.1. Overall, there was 12199 lines of code remaining in the sample prior to automatically generating tests using Pynguin.

Generating a Test Suite

Once initial filtering of the codebase was completed, Pynguin was run to generate tests. However, additional issues came up in this process that required additional filtering to be done. The new content was filtered as follows:

- When attempting to generate tests, Pynguin failed in 54 modules and threw error messages indicating possible bugs in the tool. To limit the time taken to generate tests, 15 minutes were given per module, where no tests were generated for 73 modules due to getting timed out. Finally, there were 249 modules where

SLOC	Directory	SLOC-by-Language (Sorted)
2396	data_structures	python=2396
1856	ciphers	python=1856
1583	maths	python=1583
900	graphs	python=900
664	sorts	python=664
559	strings	python=559
543	conversions	python=543
412	linear_algebra	python=412
365	divide_and_conquer	python=365
360	matrix	python=360
359	dynamic_programming	python=359
308	backtracking	python=308
248	compression	python=248
236	searches	python=236
200	arithmetic_analysis	python=200
173	hashes	python=173
171	audio_filters	python=171
161	bit_manipulation	python=161
118	cellular_automata	python=118
109	boolean_algebra	python=109
97	scheduling	python=97
94	blockchain	python=94
86	electronics	python=86
67	genetic_algorithm	python=67
43	financial	python=43
40	knapsack	python=40
37	geodesy	python=37
11	greedy_methods	python=11
3	physics	python=3

Totals grouped by language (dominant language first):
python: 12199 (100.00%)

Figure 4.1: SLOCcount Output After Initial Filtering

tests were successfully generated. Modules with failed and timed out runs were removed due to the lack of tests that cover them.

- Some generated tests were faulty and caused errors when run, or indeterminately failed making them flaky. Those tests were removed in some instances or fixed when possible.
- Since AFLuent relies on a thorough test suite with high coverage in order to find bugs accurately, modules with below 85% coverage were removed.

SLOC	Directory	SLOC-by-Language (Sorted)
934	maths	python=934
826	ciphers	python=826
576	data_structures	python=576
496	sorts	python=496
465	conversions	python=465
454	graphs	python=454
356	strings	python=356
217	dynamic_programming	python=217
192	backtracking	python=192
173	hashes	python=173
133	bit_manipulation	python=133
101	searches	python=101
90	linear_algebra	python=90
69	electronics	python=69
61	divide_and_conquer	python=61
43	financial	python=43
42	scheduling	python=42
37	geodesy	python=37
27	knapsack	python=27
16	arithmetic_analysis	python=16
14	cellular_automata	python=14
11	greedy_methods	python=11
10	compression	python=10
3	physics	python=3

Totals grouped by language (dominant language first):
python: 5346 (100.00%)

Figure 4.2: SLOCcount Output After Initial Filtering

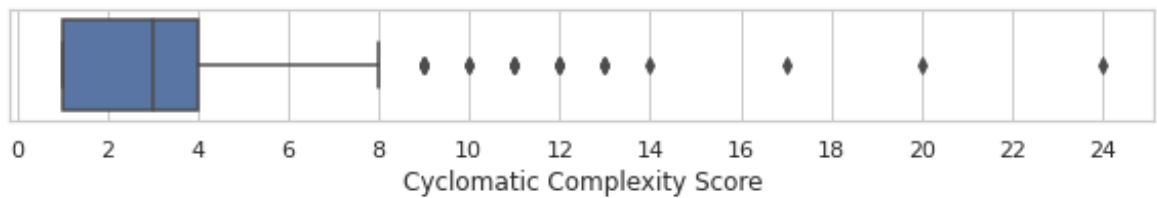


Figure 4.3: Cyclomatic Complexity of Sample

After the second filtering step SLOCcount statistics were collected again to get information on the sample, the output from the tool is shown in Figure 4.2. While the codebase was significantly reduced in size, the resulting test suite contains 1105 test cases with 99% coverage. In order to get a better understanding of the remaining

sample, additional data was collected on the cyclomatic complexity of the functions in the code. This information would give some ideas on the structure of the code regarding if statements, loops and other constructs. The cyclomatic complexity of the remaining 516 functions was calculated and plotted as shown on Figure 4.3. The figure shows a box and whiskers plot of this data, where the lower quartile and the minimum are equal causing a lower whisker of zero length. Additionally, the plot shows a median cyclomatic complexity score of 3 and a maximum of 8. Lastly, few data outliers go beyond and up to a score of 24. These results mean that the code is very linear, they also suggest that it resembles what novice developers write as simple only block function and minimal branching.

4.2 Evaluation

4.2.1 Data Collection

After choosing and filtering the sample, the evaluation of AFLuent would be conducted by repeating three main steps: (1) Introduce a single bug in the codebase, (2) Run AFLuent expecting the tests to fail, and (3) collect the reports produced by AFLuent for analysis later on. With the large sample and the possibility of inserting bugs in many possible locations, there was a need for an automated approach to collect this data. Some existing tools such as *mutmut* [3] already perform similar steps for mutation testing and it could be easily repurposed to generate the bugs/mutants and run AFLuent after inserting a mutant into the codebase. Furthermore, *mutmut* supports a hook function that facilitates collecting results after each run and before the next mutant is applied. Lastly the test suite run command can be modified to run AFLuent in evaluation mode and collect fault localization data. Overall, for each bug/mutant, the following data would be collected to use for evaluation:

- Statement rankings produced by AFLuent for the 16 approach-tiebreaker combination. Note that a value of 3 was used for * in the DStar equation.
- Per-test coverage report
- Timing report containing the time taken to run the test suite before fault localization and the time to perform fault localization and get rankings.
- Information about the mutant such as the file and line number it was added to as well as the mutated code.

AFLuent ranking and timing data was collected on five different machines each with Ubuntu operating system, Intel Xeon E3-12xx CPU, and 2GB of memory. Docker containers were created on each machine, where Python 3.10.2 was installed along with AFLuent and *mutmut*. After running *mutmut* on the filtered algorithms, it was able to generate 7767 mutants in which 4123 were killed, 3490 survived, and 154 were timed out by *mutmut*. The mutants were classified using the following criteria:

- Killed mutants: cases where the mutated code caused a failure in the test suite, as expected, prompting AFLuent to localize the fault and generate ranking reports. Every mutant that generated the ranking reports is considered killed.

- Survived: cases where the mutation did not cause a failure in the test suite. AFLuent was not run in these cases since no test case failed. Runs that only generated per-test coverage and timing reports but no ranking reports are for mutants that survived.
- Timed out: special cases where *mutmut* reported to run a mutant but no report was produced by AFLuent. The log produced by *mutmut* confirms that these mutants were timed out after taking too long to complete the test run.

When evaluating AFLuent’s effectiveness in ranking lines of code, only the results from killed mutants are relevant since that’s when reports were generated. As for efficiency evaluation, data can be split up to two categories, time taken by AFLuent to execute tests while calculating per-test coverage, and time taken to produce rankings to find faults. Both types of mutants, killed and survived, produced data points on the time taken to run the tests. However, only killed mutants can produce data points on the time taken to perform localization. Therefore, a different number of data points will be used for each category.

4.2.2 Results

The collected data is visualized in this subsection to compare the different equations to each other, additionally, tie breaking techniques are compared to determine the most effective. In order to quantify the effectiveness of each approach, an *EXAM* [25] score is calculated for each report produced from a killed mutant. This score represents the percentage of reported statements that a developer must parse through before reaching the correct line where the bug exists. To produce this score, the reported results were filtered by project so that they only include lines from that project. For example, when the *EXAM* score is calculated for a fault in the `bit_manipulation` project, the percentage of lines belonging to that project that a developer must read before finding the fault is the score. The lower the exam score for an approach, the more effective it is because the developer would have to analyze less lines before finding the fault.

To visualize the data, box plots are used to show the distribution of scores for each equation-tiebreaker combination. This form of visualizing the data allows each point to be represented in the graph and gives a clear indicator of where outliers are. Additionally, heat maps are used to compare each pair to the other fifteen after completing statistical tests.

Data Visualization

To compare suspiciousness score equations when the tie breaking approach is the same, Figures 4.4, 4.5, 4.6, 4.7 group *EXAM* scores by tie breaking technique for each killed mutant. On each plot, the x-axis contains the name of the approach and the y-axis the percentage of score to be analyzed before finding the mutant (also referred to as the *EXAM* score). Additionally, the value of the median is labeled on each boxplot. Lastly, outliers are shown using the black circles that go beyond the limit of the upper whisker on each plot.

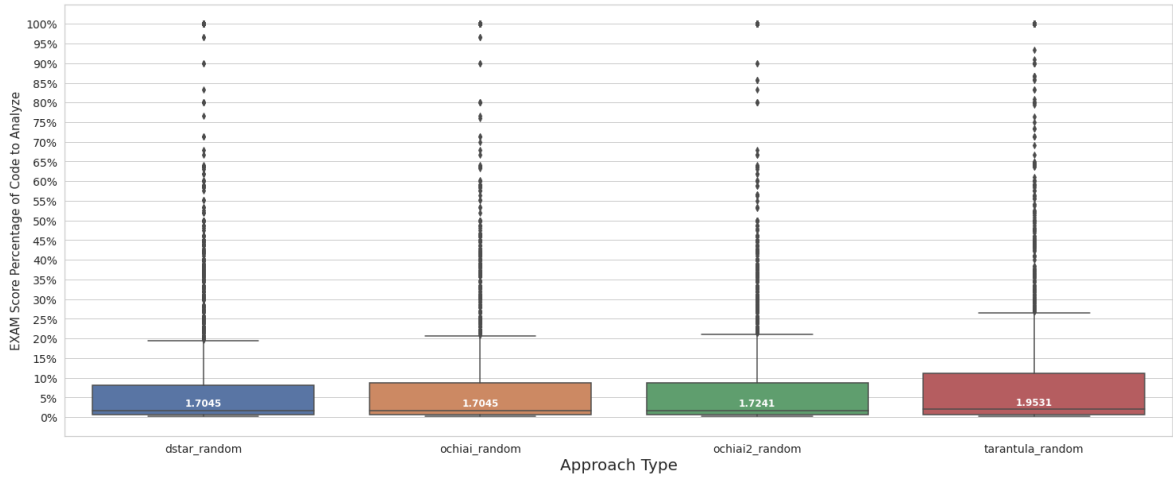


Figure 4.4: *EXAM* Scores Using Random Tiebreaking

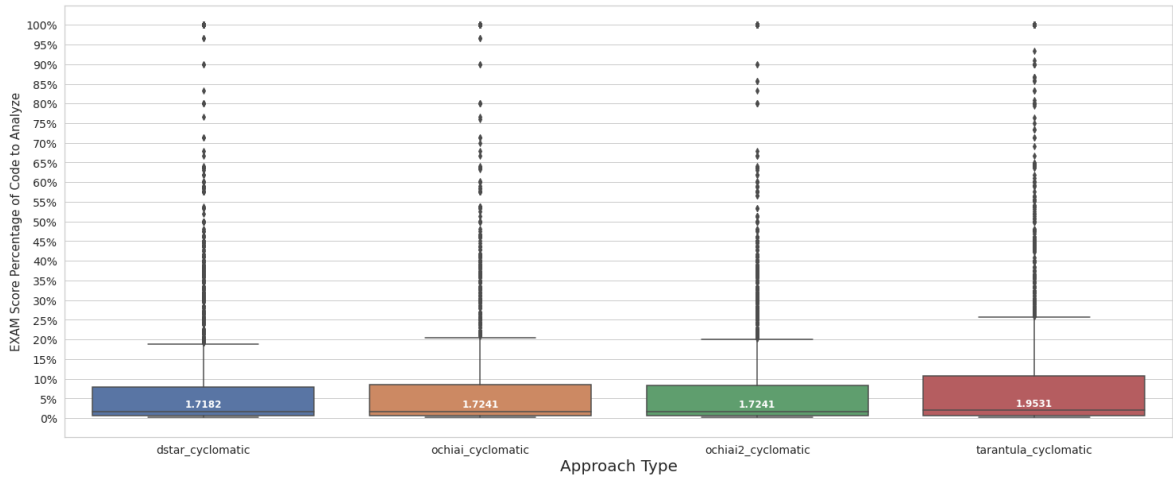


Figure 4.5: *EXAM* Scores Using Cyclomatic Tiebreaking

In Figure 4.4, which compares the four equations using random tie breaking, one noticeable difference between the equations is in the Tarantula approach which has the highest median of 1.9531. Additionally, it has the highest maximum *EXAM* score around 26%. Out of the remaining equations in the plot, DStar and Ochiai have the same median, however, DStar has a lower maximum making it slightly the better equation. From visually inspecting the data in Figure 4.4, DStar appears to be the most effective, while Tarantula is the least effective in fault localization. So far, this only applies when ties are handled randomly.

Moving on to Figure 4.5, improvements in the median are not noticeable. In fact, the median for DStar and Ochiai is higher using cyclomatic tie breaking when compared to random tie breaking. The general trend in the effectiveness of the equation stays the same from the previous figure, where DStar was the most effective and Tarantula was the least effective. However, Figure 4.5 could suggest that breaking ties using

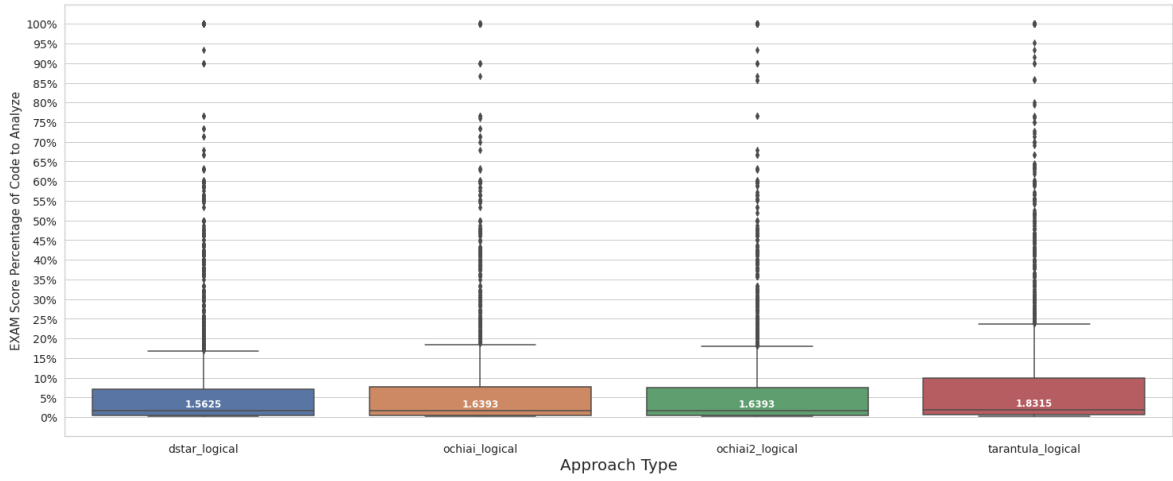


Figure 4.6: *EXAM* Scores Using Logical Tiebreaking

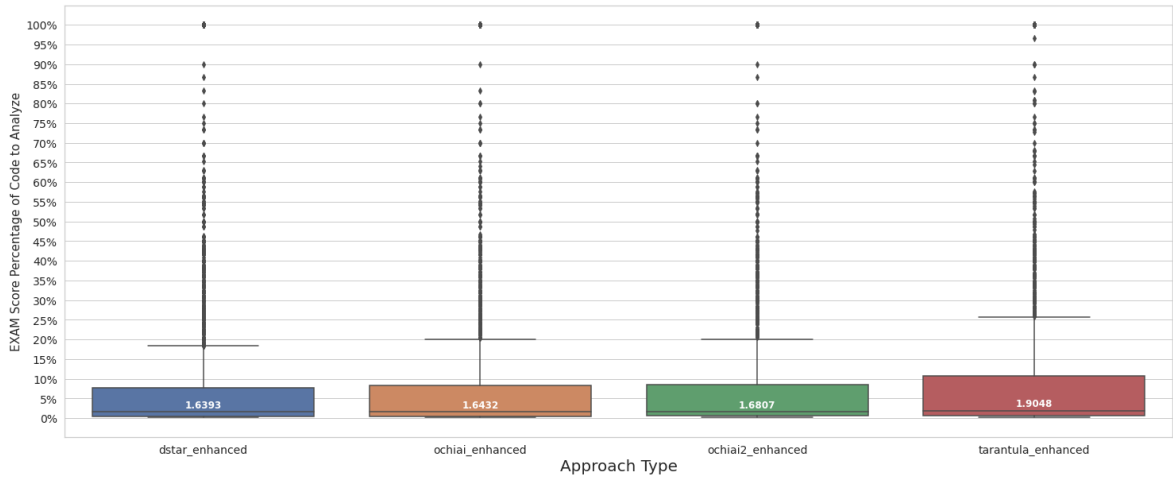


Figure 4.7: *EXAM* Scores Using Enhanced Tiebreaking

cyclomatic complexity is no better than randomly doing so. But, further statistical analysis is needed to come to that conclusion.

When analyzing Figure 4.6, there is a noticeable visual difference to the previous plots. Specifically, the median *EXAM* score for all approaches using logical tie breaking is smaller when compared to random and cyclomatic tie breaking. While this implies an improvement, it's still unclear if it's statistically significant. In addition to smaller score medians, approaches using logical tie breaking have smaller maximums. As for outliers, some differences are present, where there are small shifts downwards in their values, further implying the improvement that logical tie breaking provides. As for the rankings between the equations, Tarantula still appears to be the worst performing while DStar has the lower scores overall.

Lastly, *EXAM* scores while using enhanced tie breaking are shown in Figure 4.7. When comparing the results of this figure to those in Figure 4.6, the median values

as well as maximums appear to be higher. This indicates that enhanced tie breaking might not be as effective as logical tie breaking. However the enhanced approach still seems as an improvement to the random and cyclomatic ones. The rankings of the approaches remains the same as the previous three figures, where DStar has the lowest overall scores and Tarantula has the highest.

Throughout the last four figures, it appears that Ochiai and Ochiai2 have generally performed the same, with Ochiai2 having slightly higher median in some cases. This is quite surprising considering that Ochiai2 is considered an improvement on Ochiai. Overall, both of them have had the same maximum and similar distribution in outliers. From visual analysis of the plots, it's unclear which is more effective.

In addition to the box plots that showcase the medians, and the quartiles, it's worth looking at the average exam score for each approach. Figure 4.8 plots the *EXAM* score averages for all categories and lists the value of the mean at the top of each bar. This bar plot further demonstrates the low performance of Tarantula, in which it has the highest averages out of all other equations. Based on the averages in the plot, the best performing approach is DStar with logical tie breaking with an average of 18.27% *EXAM* score. It's followed by Ochiai logical and Ochiai2 logical with very close averages.

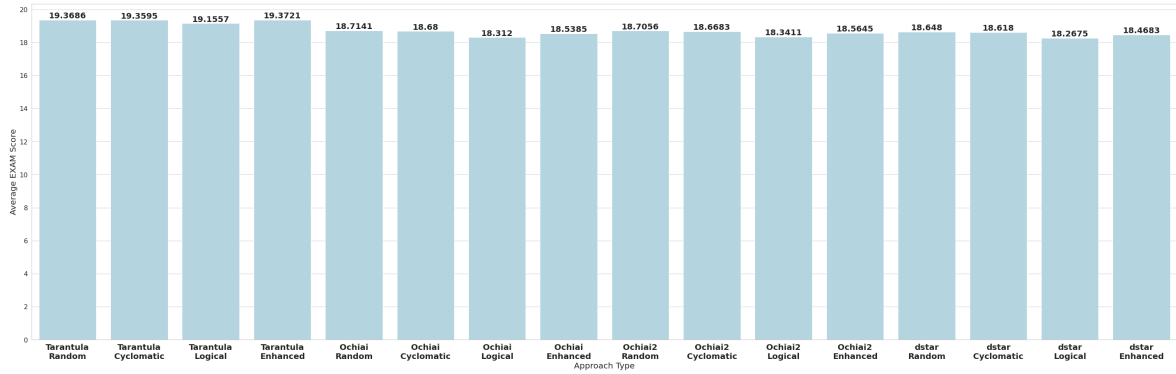


Figure 4.8: Average *EXAM* Scores for all Approaches

Statistical Tests: Mann-Whitney

Simple visual analysis of the data is not enough to reach a justifiable result regarding the most effective approach. Therefore, this subsection discusses how two statistical tests were conducted to compare all sixteen different approaches to each other. Starting with the Mann-Whitney U Test, it is used to test whether the distributions of two independent samples are equal or not. In the two-sided Mann-Whitney test, the hypotheses are as follows:

Null hypotheses: $\mathbf{H}_0: \mathbf{A}_x = \mathbf{A}_y$

Alternative hypotheses: $\mathbf{H}_1: \mathbf{A}_x \neq \mathbf{A}_y$ for $p \leq 0.05$

Where \mathbf{A}_x is the approach on the x-axis and \mathbf{A}_y is the approach on the y-axis

The generated *p-values* from the test represent the probability that H_0 is true. Therefore, when *p-values* are less than or equal to the statistically significant level of 0.05, the test indicates that the two approaches are statistically different. However, it does not conclude which approach had the higher or lower exam scores. Figure 4.9 plots a heat map of the resulting *p-values* from the two sided Mann-Whitney test. It compares each approach to the fifteen remaining ones. *P-values* for each comparison are included in each square, where squares with lower *p-value* are darker color than those with larger value.

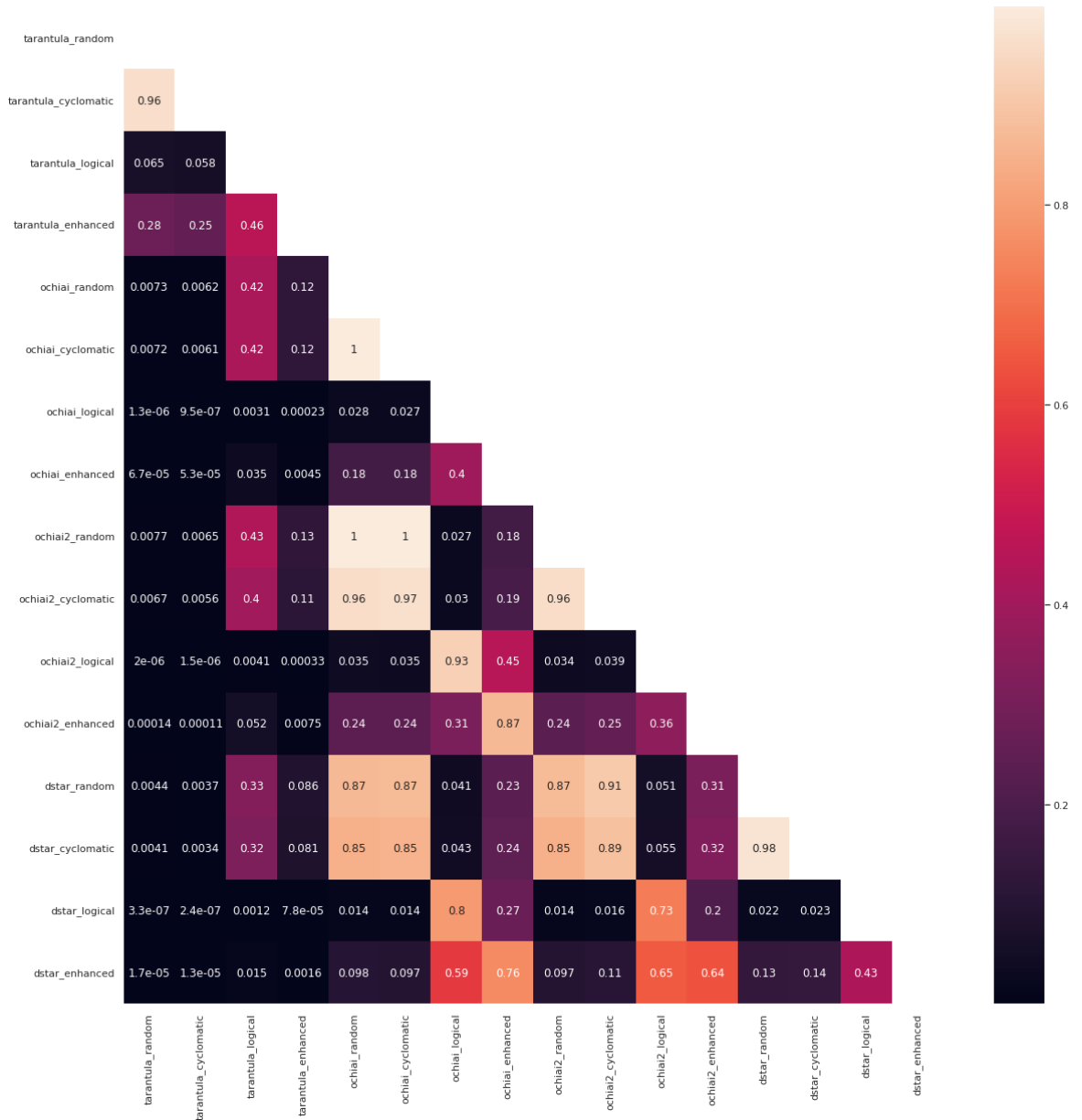


Figure 4.9: *p-values* of Two Sided Mann-Whitney Test

Many conclusions can be reached from the *p-values* displayed in the heat map, some of which further affirm the observations made earlier from the plots, and others show

a completely different result. Starting with the Tarantula approaches, the *p-values* for Tarantula clearly show that these sets are different from all non-Tarantula approaches to a very statistically significant level. Therefore, the null hypothesis is rejected here. There are only a few exceptions, where Tarantula Enhanced and Tarantula Logical are not different from the random and cyclomatic variants of Ochiai, Ochiai2, and DStar, so the null hypothesis is accepted for these exceptions. These cases represent an interesting case where using the logical and enhanced tiebreakers was a significant enough improvement to bring Tarantula in line with the random and cyclomatic variants of other equations. When comparing Tarantula variants amongst each other, they aren't statistically different, therefore, the null hypothesis is accepted in this case.

For the Ochiai, Ochiai2, and Dstar equations, the heat map shows an interesting relationship between them. The Random and Cyclomatic variants were very similar, both across and among each other. For all 15 of these pairs shown in the light squares in the center of the plot, *p* was above 0.85 suggesting that the null hypothesis should be accepted. This result suggests that the different tie breaking techniques were not effective in giving an equation the edge over another. As for the remaining variants of Ochiai, Ochiai2, and DStar, namely logical and enhanced, they are not different from each other on a statistically significant level. Therefore, the null hypothesis is accepted for these combinations.

While the results of this test are helpful to show if statistically significant differences exist between the different approaches, it does not clarify which one is better than the other. To have a better understanding of that relationship, a one sided `less` Mann-Whitney test is performed with the hypotheses are as follows:

Null hypotheses: $\mathbf{H}_0: \mathbf{A}_x = \mathbf{A}_y$

Alternative hypotheses: $\mathbf{H}_1: \mathbf{A}_x < \mathbf{A}_y$ for $p \leq 0.05$

Where \mathbf{A}_x is the approach on the x-axis and \mathbf{A}_y is the approach on the y-axis

In contrast to the two-sided test, this one gives a more clear idea on which approaches has a lower *EXAM* scores when compared to others. A *p-value* less than or equal to 0.05 suggests that the approach on the x-axis is significantly more effective than its counterpart on the y-axis. On the other hand, a *p-value* greater than 0.95 indicates the opposite, where the approach on the y-axis is significantly more effective than the one on the x-axis. Figure 4.10 plots a heat map of the resulting *p-values* from this test.

Starting with the Tarantula variants, none of them resulted in a statistically significant *p-value* to suggest that they are more effective than the other non-Tarantula approaches, thus, the null hypothesis is accepted. However, among Tarantula variants, the logical tie breakers has the lowest *p-value*, suggesting that it was the most effective but not to a significant level. To further illustrate how Tarantula falls behind the other equations, every non-Tarantula approach significantly outperformed the random and cyclomatic variants of Tarantula. The logical and enhanced variants were also outperformed by others, however, it was less frequently.

As for other equations, the Random and Cyclomatic variants of Ochiai, Ochiai2, and DStar all had similar results between each other, where none of them outperformed another on a significant level. Therefore, for these approaches, the null hypotheses is

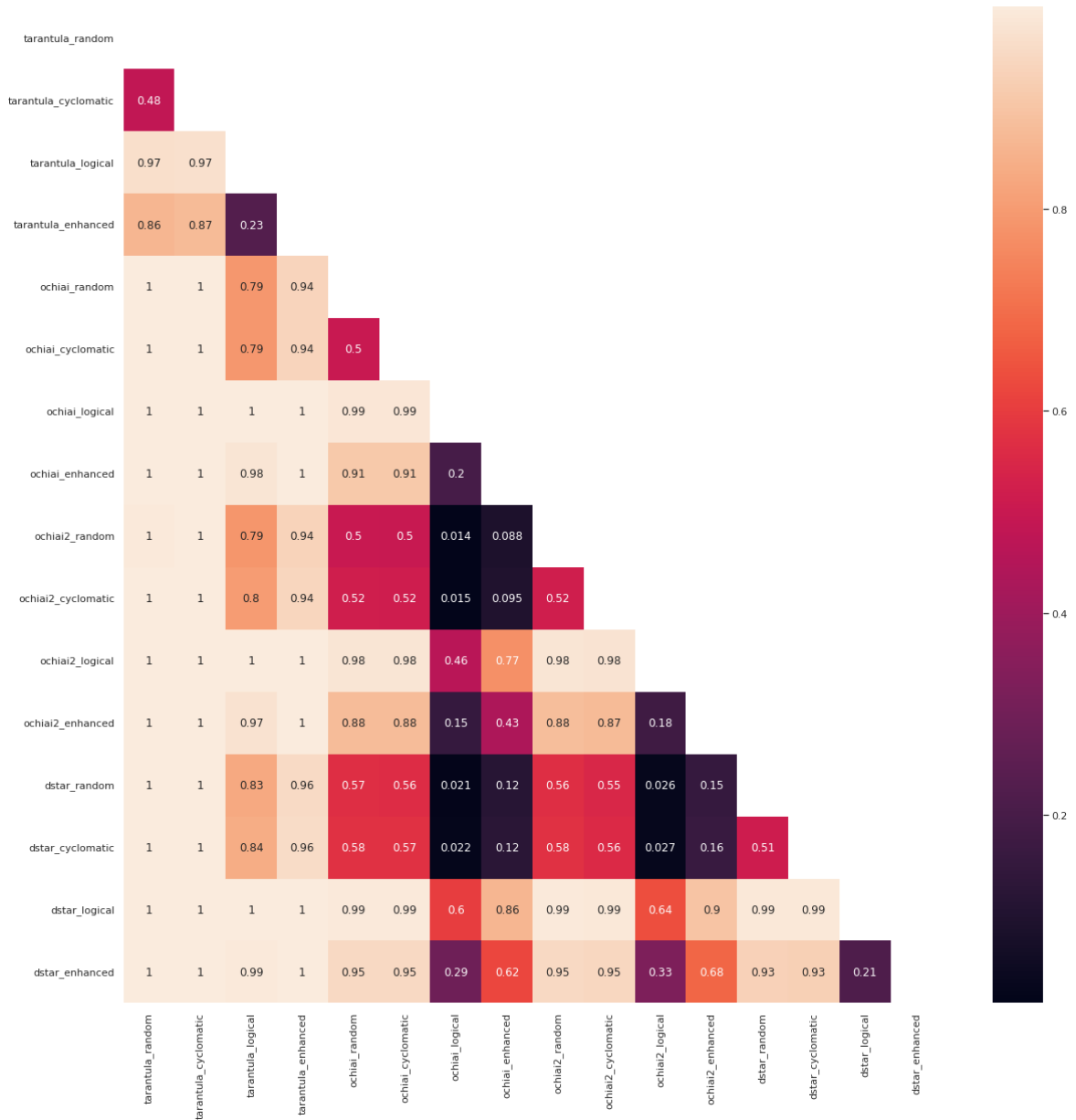


Figure 4.10: *p-values* of One Sided Mann-Whitney Test

accepted. This result matches the outcome of the two-sided test where these approaches were very similar.

Some notable large differences in performance can be seen in the logical variant of Ochiai, Ochiai2, and DStar. These three significantly outperformed every random and cyclomatic variant of all other equations. On the other hand, the enhanced variant of these slightly outperformed the cyclomatic and random ones, but not on a statistically significant degree. This result suggests that the logical tie breaker is the most effective in assisting equations to produce the lowest *EXAM* scores. Furthermore, it demonstrates that while Ochiai and Ochiai2 do not outperform each other, both show a significant improvement to almost every other combination.

One of the surprising outcomes of this data is the performance of enhanced tie breaking. Despite the fact that it adds on the mutant density metric provided by logical, and considers wider possibility while generating its score, it did not outperform the logical tie breaker. In fact, the *p-value* was leaning more to the other outcome, but not in any significant way.

Statistical Tests: Cohen’s D Effect Size

While the Mann-Whitney test checks if the distribution of the approaches is different to others on a statistically significant level, it does not give an idea of the magnitude of this difference. In order to get that information, the non-parametric Cohen’s D effect size is calculated for each pair of approaches to understand the difference between their average *EXAM* score. Figure 4.11 shows the heat map of the results of this calculation. The values in each square represent the difference between the approach on the x-axis and the one on the y-axis, where negative values would indicate that the approach from the x-axis had, on average, a lower *EXAM* score than its counterpart. The opposite is also true, where positive values indicate that the approach on the y-axis was more effective.

The general trend in this plot further confirms what was discussed in the Mann-Whitney test. The logical variant of Ochiai, Ochiai2, and DStar remains as the most effective out of all others, however, this plot can help in deciding which of these three was the best. At the intersections of Ochiai logical and Ochiai2 logical with DStar logical, the values 0.0013 and 0.0021 respectively show that DStar had an improvement over the other two. This improvement is very small and, as established in the previous test, not statistically significant. As for the comparison between Ochiai logical and Ochiai2 logical, an even smaller value indicates a very small difference in favor of Ochiai logical. Again, this difference is not statistically significant to reach a conclusion of which is the better approach. Overall, the performed statistical tests allowed the filtering of 16 different approaches to determine the top 3 ones that yielded the lowest *EXAM* scores.

Time Efficiency

In order to provide a time comparison that accounts for the different cases of AFLuent runs, three different categories of time data are included: (I) Time to execute tests without AFLuent, (II) Time to execute tests with AFLuent enabled, (III) Time to locate faults and perform tie breaking using all equation-tiebreaker combinations. Generally all these times were calculated when all 1105 test cases in the project’s suite were run.

Figure 4.12 compares the time taken to execute all test functions with and without using AFLuent. The baseline includes 4000 data points where the tests were run without AFLuent. On the other hand 7613 data points are plotted for when AFLuent generated a timing report. It’s important to mention that every point in the boxplot on the right represents the combined time to run all equation-tiebreaker combinations after one another. While this creates an unfair comparison, limitations of the experiment prevented the collection of more detailed data. In general, the figure shows a clear

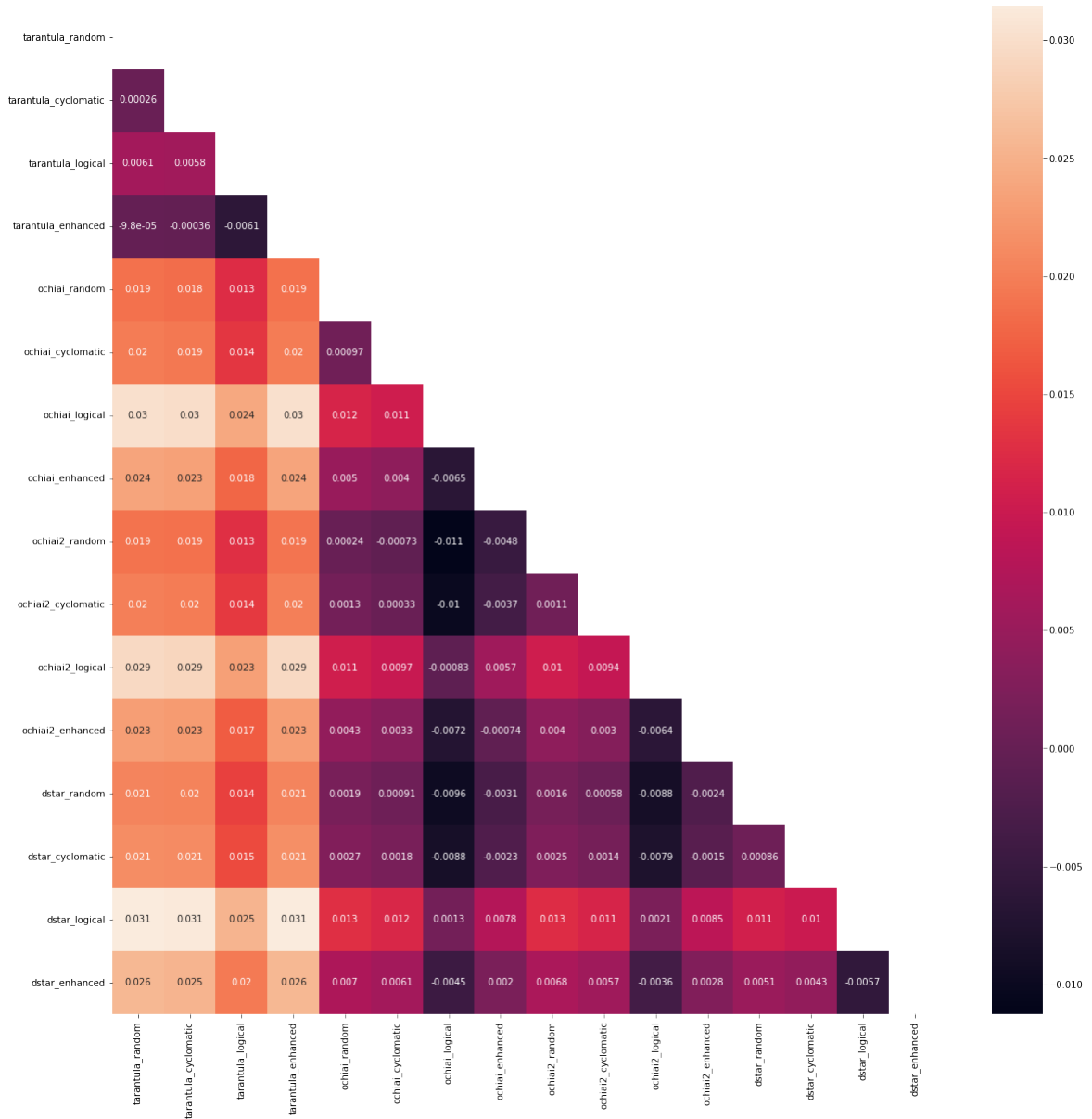


Figure 4.11: Results of the non-parametric Cohen D Effect Size

difference between the two box plots, where the median run with AFLuent took longer time to complete than the median run for the baseline. Additionally, the long upper whisker indicates that including AFLuent increases the chances that the tests could take substantially longer to run. This increase is somewhat expected since AFLuent introduces wrapper functions around each test execution to calculate per-test coverage. And while this increase is quite small, considering the large size of the test suite, minimizing the additional time cannot be done through AFLuent since the tool relies on Coverage.py to calculate per-test coverage. After performing a two-sided Mann-Whitney Test on both sets, the resulting *p-value* of $8.035528e-157$ confirms that there is statistically significant difference between the two sets, where the null hypothesis is

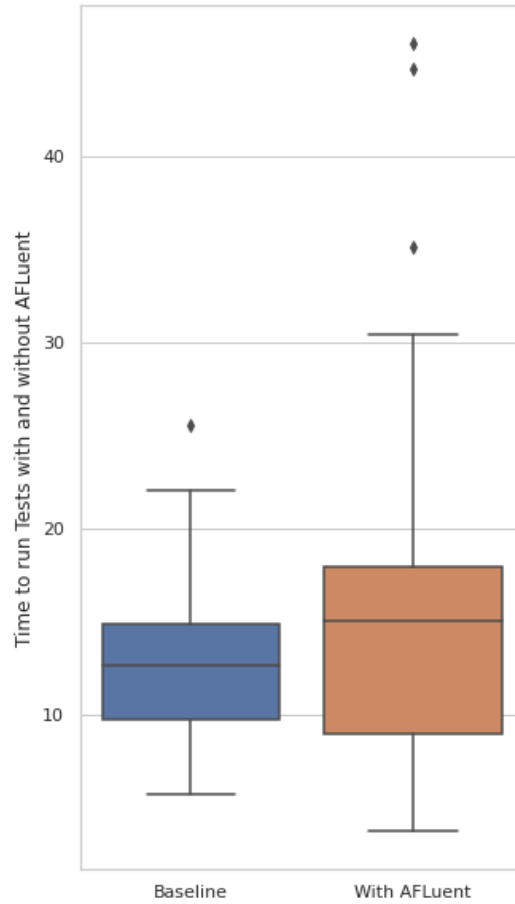


Figure 4.12: Test Execution Time

rejected.

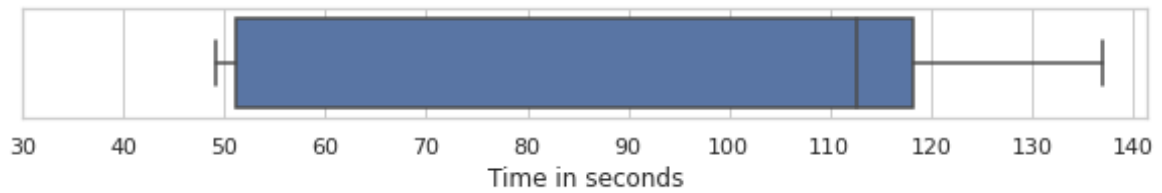


Figure 4.13: Fault Localization Time

Another useful time metric for developers hoping to use AFLuent is the time it took to locate faults. And while comprehensive time data for each equation and tie breaking approach was not collected, some conclusions can be drawn from the most time consuming case. Figure 4.13 shows the time taken to generate a statement ranking following a test failure. This includes the time to calculate suspiciousness scores using all equations and to retrieve and evaluate tie breaking values using all approaches. In general, this is the worst possible scenario for AFLuent localization time. While

the median time is quite large, around 115 second, so is the test suite. For every run, AFLuent generates and parse abstract syntax trees using LibCST and calculates cyclomatic complexity fo all files covered in the suite. In the case of this evaluation, this includes all files in the codebase. Further discussion on how this time can be minimize is found in Section 5.2: Future Work.

4.3 Threats to Validity

There are some aspects of this study where the method of collecting data and performing evaluation could be criticized. In this relatively small-scale evaluation of SBFL equations, the sample chosen to test AFLuent might not adequately represent code written by novice developers. While this brings up further questions on how would AFLuent perform on a different type of code base, it does not completely invalidate the results reached here. The repeated shrinking of the sample from the original `TheAlgorithms/Python` to a very filtered version can raise several red flags and questions regarding why specific code was removed. Given the constraints of this study, however, and the intention to use Pynguin to generate unit tests automatically, some code had to be removed in order for Pynguin to run in a timely fashion and produce appropriate tests. It could be argued that the tests generated by Pynguin were not very thorough, especially that almost 50% of the mutants survived while running `mutmut`. This argument brings up a valid criticism and suggests that a well rounded evaluation of AFLuent requires that the sample be developed and tested by novice developers. Additionally, it indicates that an automatically generated test suite does not adequately mimic one written by novice developers. Lastly, one possible concern could be brought up regarding the calculation of th *EXAM* scores. Since mutant information was collected from `mutmut`, and coverage information was collected from `Coverage.py`, this could create a possible mismatch between the actual location of the mutant and how lines were ranked in AFLuent’s report. For example, some instances were observed where `mutmut` introduced a mutant to a global constant, and while this constant was used and tested, `Coverage.py` does not consider the variable lines to be covered. And since it supposedly wasn’t covered, a suspiciousness score was not calculated for it. This results in the *EXAM* score being at its maximum of 100% because the faulty line wasn’t present in the ranking at all. There could be other possible causes of mismatches that were not observed so this creates uncertainty regarding the validity of the data. However, these cases only affect AFLuent negatively by producing a higher *EXAM* score than expected, therefore, this could be an underestimate evaluation of AFLuent’s performance.

Chapter 5

Discussion and Future Work

5.1 Summary of Results

The results of this study can be split up into two main contributions, as the title suggests, implementation and evaluation. AFLuent represents a first-of-a-kind SBFL tool directly integrated to the Pytest framework. Furthermore, its emphasis on ease of use and readability makes it a great fit for novice developers. AFLuent as a tool is maintainable, readable, and extendable in several ways that can potentially serve for more thorough evaluations of SBFL algorithms in the future. In addition to implementation technical details, an evaluation of AFLuent on two main metrics, effectiveness and efficiency, is presented. After a visual and statistical analysis of *EXAM* scores for each equation and tie breaker pair implemented in AFLuent, much was revealed regarding the best and worst performances. One observation that stood out is the significant underperformance of Tarantula when compared to all other equations. Additionally, the statistically significant edge that logical tie breaking achieves to outperform its random and cyclomatic counterparts was surprising. Overall, Ochiai, Ochiai2 and DStar had very strong performances when the same tie breaker was used across all of them, however, the data did not suggest that one outperformed the other significantly. Additional experiments may yield different results and uncover new information about these equations. As for the most effective tie breaking technique, the logical approach emerged as the most successful in creating a statistically significant difference followed by the enhanced tie breaker. Additionally, the presented time data demonstrate the time overhead that developers need to take on if they choose to use AFLuent. While further evaluation is needed to determine the time efficiency of each approach independently from others, the worst case time data suggest that calculating per test coverage and performing fault localization will lead to a significant increase in runtime.

5.2 Future Work

Extensions can be added to AFLuent to explore more approaches in the SBFL field as well as increase the tool's ability to find faults. These types of additional work can be split up to three main sub-categories: effectiveness, efficiency, and evaluation.

Effectiveness

Many SBFL equations have been studied and evaluated in previous literature, however, AFLuent’s inclusion of only the four, Tarantula, Ochiai, Ochiai2, and DStar is only the first step in analyzing how fault localization can perform in Python projects. Additionally, setting a limit is needed due to time constraints. Possible extensions of this work can implement more SBFL equation options for developers to utilize, which could be especially helpful if new techniques prove to be more effective than those currently implemented. Another feature of AFLuent that significantly influenced the collected performance metrics is the approach used for tie breaking. Additional enhancements and techniques can be added to improve existing ones, especially the ‘enhanced’ tie breaker. This could be completed by expanding the list of possible mutants and improving the analysis of the abstract syntax trees. These potential improvements can be easily added to the existing code by implementing new functions to calculate suspiciousness or break ties in the object oriented structure. This extension would allow developers to add their favorite SBFL equation into AFLuent easily.

Efficiency

One of the concerning outcomes of this study is the long time AFLuent takes to produce fault localization output. Developers usually want fast and optimized tools, which could render AFLuent unusable in the eyes of many. And while AFLuent’s reliance on several tools reduce the ability to control the time it takes to run, there are some measures that could mitigate this problem. Throughout manual testing of the tool, it was generally observed that the most time consuming feature of AFLuent involves generating the tie breaker datasets. In order for AFLuent to adequately understand the code under test, it must generate an abstract syntax tree for every file covered in the test suite. Since the debugging process usually involves making small changes at a time and rerunning the tests, a time improvement is possible by caching all generated syntax trees and only re-generating ones for files that have been edited since the last run. This solution will not reduce the runtime for the first time but it could have a significant effect on the runs that follow. Overall, this change introduces a quality of life improvement by reducing the time developers need to wait before receiving the report from AFLuent. Once these improvements have been made, a more detailed evaluation of AFLuent’s performance is also needed. By comparing each equation and tie breaking approach combination independently, a better understanding can be reached regarding how AFLuent performs based on the used approach.

Evaluation

While the evaluation section of this study managed to filter out many ineffective combinations of equations and tie breakers, it could not reach a definite conclusion on the best approach to perform SBFL. Furthermore, the evaluation was conducted on a small scale by focusing on small projects with low complexity. Time and feasibility constraints were very prevalent, which led to adopting a strategy to simulate novice written code and unit tests rather than including human subjects to write, test, and

debug the code. Even though the used strategy has some flaws, it is justifiable considering the scope of this study. To improve on the current understanding of AFLuent's performance, additional experiments are needed on a larger sample of Python projects that includes a variety of programming styles and utilizes a diverse set of Python constructs. Furthermore, extending the study to include novice developers by asking them to write and debug their code using AFLuent would provide great insight on their perspective on the tool and how they plan to use it. For example, a potentially more thorough evaluation of AFLuent would consist of these steps:

1. Select computer science students with various levels of experience
2. Assign the students to write and test a collection of programs
 - Students can be split into groups to complete different assignments
 - Chosen assignments should allow for test driven development
3. Allow some groups of students to use AFLuent and assess its effectiveness in helping them locate faults
4. Collect direct feedback regarding the students' experience while using AFLuent

While these steps do not provide a comprehensive plan for an alternative approach of evaluating AFLuent, it seeks to take into consideration the human experience aspect of using and interacting with the tool. This metric is missing from the current evaluation, and AFLuent, as well as the SBFL field, could benefit from more data surrounding it.

5.3 Ethical Implications

Aside from answering research questions in the form of implementation and evaluation, this study includes an interesting ethical question regarding the potential misuse of a tool such as AFLuent. One major difference between AFLuent and a traditional debugger is the way a developer interacts with each. In order to use a traditional debugger, the developer must be well versed in the code base and understand where breakpoints should be placed. Additionally, they must be able to identify what suspicious behavior looks like. On the other hand, AFLuent simplifies this process and provides a statement ranking for the developer to parse through and check. By doing so, AFLuent serves its goal in making the debugging process simpler and more efficient, but it could lead to over reliance, especially from novice developers. If developers become too dependent on AFLuent, or any automated fault localization tool, it could negatively impact their development skills by taking away the experience of manually analyzing the code and understanding its expected behavior and why failures occur. This could be especially harmful if AFLuent was overused in educational environments since students could utilize its functionality without fully understanding how to fix the code themselves, and thus negatively impact their learning process.

Bibliography

- [1] 2020 State of API Report Postman. <https://www.statista.com/statistics/1083189/worldwide-api-tasks-time-allocation/>, 2020.
- [2] AFLuent/AFLuent. <https://github.com/AFLuent/AFLuent>, 2022.
- [3] boxed/mutmut. <https://github.com/boxed/mutmut>, 2022.
- [4] TheAlgorithms/Python. <https://github.com/TheAlgorithms/Python>, 2022.
- [5] ABREU, R., ZOETEWIJ, P., GOLSTEIJN, R., AND VAN GEMUND, A. J. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792. SI: TAIC PART 2007 and MUTATION 2007.
- [6] ABREU, R., ZOETEWIJ, P., AND VAN GEMUND, A. J. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)* (2006), pp. 39–46.
- [7] BATCHELDER, N. Coverage.py, 2022.
- [8] COSMAN, B., ENDRES, M., SAKKAS, G., MEDVINSKY, L., YANG, Y.-Y., JHALA, R., CHAUDHURI, K., AND WEIMER, W. Pablo: Helping novices debug python code through data-driven fault localization. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2020), SIGCSE '20, Association for Computing Machinery, pp. 1047–1053.
- [9] DEBROY, V., WONG, W. E., XU, X., AND CHOI, B. A grouping-based strategy to improve the effectiveness of fault localization techniques. In *2010 10th International Conference on Quality Software* (2010), pp. 13–22.
- [10] HAILPERN, B., AND SANTHANAM, P. Software debugging, testing, and verification. *IBM Systems Journal* 41, 1 (2002), 4–12.
- [11] JANSSEN, T., ABREU, R., AND VAN GEMUND, A. J. Zoltar: A toolset for automatic fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering* (2009), pp. 662–664.
- [12] JONES, J., HARROLD, M., AND STASKO, J. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002* (2002), pp. 467–477.

-
- [13] JONES, J. A., AND HARROLD, M. J. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (New York, NY, USA, 2005), ASE '05, Association for Computing Machinery, pp. 273–282.
 - [14] KOHN, T. The error behind the message: Finding the cause of error messages in python. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2019), SIGCSE '19, Association for Computing Machinery, pp. 524–530.
 - [15] LE, T.-D. B., THUNG, F., AND LO, D. Theory and practice, do they match? a case with spectrum-based fault localization. In *2013 IEEE International Conference on Software Maintenance* (2013), pp. 380–383.
 - [16] LUKASCZYK, S., AND FRASER, G. Pynguin: Automated unit test generation for Python. 44th International Conference on Software Engineering Companion (ICSE '22 Companion).
 - [17] MCCABE, T. A complexity measure. *IEEE Transactions on Software Engineering SE-2*, 4 (1976), 308–320.
 - [18] NAISH, L., LEE, H. J., AND RAMAMOHANARAO, K. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)* 20, 3 (2011), 1–32.
 - [19] PARNIN, C., AND ORSO, A. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (New York, NY, USA, 2011), ISSTA '11, Association for Computing Machinery, pp. 199–209.
 - [20] PARSAI, A., AND DEMEYER, S. Mutant density. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops* (June 2020), ACM.
 - [21] SARHAN, Q. I., AND BESZÉDES, Á. A survey of challenges in spectrum based software fault localization. *IEEE Access* (2022).
 - [22] SARHAN, Q. I., SZATMÁRI, A., TÓTH, R., AND BESZÉDES, A. CharmFL: A fault localization tool for Python. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)* (2021), IEEE, pp. 114–119.
 - [23] STACKOVERFLOW. 2021 StackOverflow Developer Survey, 2021.
 - [24] WANG, S., LO, D., JIANG, L., LUCIA, AND LAU, H. C. Search-based fault localization. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)* (2011), pp. 556–559.

-
- [25] WONG, E., WEI, T., QI, Y., AND ZHAO, L. A crosstab-based statistical method for effective fault localization. In *2008 1st International Conference on Software Testing, Verification, and Validation* (2008), pp. 42–51.
 - [26] WONG, W. E., DEBROY, V., GAO, R., AND LI, Y. The dstar method for effective software fault localization. *IEEE Transactions on Reliability* 63, 1 (2014), 290–308.
 - [27] WONG, W. E., GAO, R., LI, Y., ABREU, R., AND WOTAWA, F. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.
 - [28] XIE, X., CHEN, T. Y., AND XU, B. Isolating suspiciousness from spectrum-based fault localization techniques. In *2010 10th International Conference on Quality Software* (2010), pp. 385–392.
 - [29] XU, X., DEBROY, V., ERIC WONG, W., AND GUO, D. Ties within fault localization rankings: Exposing and addressing the problem. *International Journal of Software Engineering and Knowledge Engineering* 21, 06 (2011), 803–827.
 - [30] XUAN, J., AND MONPERRUS, M. Learning to combine multiple ranking metrics for fault localization. In *2014 IEEE International Conference on Software Maintenance and Evolution* (2014), pp. 191–200.
 - [31] YOO, S., XIE, X., KUO, F.-C., CHEN, T. Y., AND HARMAN, M. No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. *RN* 14, 14 (2014), 14.