Can runtime profiling information provide a useful signal in determining whether two implementations are "similar" in some form?

Iyamintan Awodola, BAI

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

Supervisor: Vasileios Koutavas

November 2024

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Iyamintan Awodola

April 22, 2024

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Iyamintan Awodola

April 22, 2024

Can runtime profiling information provide a useful signal in determining whether two implementations are "similar" in some form?

Iyamintan Awodola, Master of Science in Computer Science
University of Dublin, Trinity College, 2024

Supervisor: Vasileios Koutavas

It's becoming significantly easier to develop, write and maintain code with the addition of natural language learning models (such as Chat-GPT). This combined with the ability to obfuscate your codebase makes it more difficult to detect similarity between programs. The research problem was to discover whether through regression testing via fuzz testing and profiling classes that based on the data that is extracted a similarity score between programs could be determined.

The quest to effectively detect code similarities has prompted innovative approaches beyond traditional static analysis methods. This research investigates the viability of utilizing runtime information as a metric for code similarity detection, with the aim of reshaping current paradigms in this domain. Various methodologies were explored, including leveraging Java Quick Fuzzing (JQF) in conjunction with tools like VisualVM and Java Command-Line Management (JCMD), to capture runtime data such as CPU usage and memory behavior. Despite encountering challenges in accurately profiling target applications and extracting desired data formats, these initial attempts provided valuable insights into potential metrics for consideration.

The evaluation phase involved analyzing runtime data extracted from a diverse dataset of user submissions. The dataset was organized and formatted into a spreadsheet for comprehensive analysis. Through meticulous examination, both similarities and differences between user submissions were uncovered. Notable findings include instances of code segments exhibiting similar structures and variable naming conventions, indicative of potential code reuse or plagiarism. Moreover, comparisons were made between the runtime scores obtained from the dataset and similarity scores generated by the compare50 tool.

Results revealed discrepancies in the closest pairs identified by runtime scores and those identified by compare50, underscoring the need for further investigation and refinement of code similarity detection methodologies.

In conclusion, this research contributes to the ongoing discourse on code similarity detection by proposing runtime information as a promising avenue for exploration. By shedding light on the challenges and opportunities associated with leveraging runtime data, this study lays the groundwork for future advancements in this field. Ultimately, the findings advocate for a holistic approach to code similarity detection, one that incorporates runtime metrics alongside existing static analysis techniques to enhance the accuracy and effectiveness of plagiarism detection tools.

Acknowledgments

I would like to thank my supervisor, Dr. Vasileios Koutav for his contributions and effort throughout this project. This was a long journey that could not have been completed without his help, we decided on this project which took a novel approach and didn't have much literature to use as a starting point. His weekly meetings with me, combined with his recommendations and advice really pushed me to complete this project to the best of my ability.

Iyamintan Awodola

University of Dublin, Trinity College
November 2024

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.0.1 Background

In the ever-expanding landscape of software development, the integrity and originality of code are paramount. Particularly in academic settings, where assignments and projects serve as fundamental components of learning and assessment, ensuring the authenticity of code submissions is crucial. Plagiarism detection mechanisms play a vital role in maintaining academic integrity, allowing educators to identify instances of code reuse or unauthorized collaboration among students. Traditional methods for detecting plagiarism primarily rely on static analysis techniques, which examine code structures and syntactic patterns to identify similarities between code fragments. While these methods have been effective to some extent, they often struggle to capture nuanced similarities and can be prone to false positives. This in combination with the addition of large language models and MOSSAD, it has become significantly easier to avoid the detection of traditional plagiarism tool.

1.0.2 Motivation

The limitations of static analysis methods have spurred on this research to explore alternative approaches for code similarity detection. One promising avenue is the utilization of runtime information, which provides insights into how a program behaves during execution. Unlike static analysis, which examines code at rest, runtime analysis offers dynamic insights into program execution, including CPU utilization, memory usage, and function call patterns. By leveraging runtime data, this research aim to uncover hidden similarities between code fragments that may elude traditional static analysis techniques. The motivation for this research also stemmed from the involvement of using random input to test programs using fuzz testing (Fuzzing). The research problem itself is unique and

tackling this question provided an opportunity to learn more about how programs utilize the CPU and fuzz testing.

1.0.3 Research Gap

Despite the potential benefits of using runtime information for code similarity detection, this approach remains relatively under-explored in the academic literature. Existing studies primarily focus on static analysis methods, leaving a gap in our understanding of the effectiveness and practicality of runtime-based approaches. Moreover, the challenges associated with capturing, processing, and interpreting runtime data present significant hurdles that must be addressed to realize the full potential of this approach.

1.0.4 Research Objectives

In light of the aforementioned gaps and challenges, this research seeks to address the following objectives:

1. Investigate the feasibility of utilizing runtime information as a metric for code similarity detection.

2. Explore methodologies for capturing and analyzing runtime data, including tools and techniques for profiling program execution.

3. Evaluate the effectiveness of runtime-based approaches in detecting code similarities compared to traditional static analysis methods.

4. Identify challenges and limitations associated with runtime-based approaches and propose strategies for mitigating them.

1.0.5 Research Methodology

To achieve these objectives, a multi-faceted research methodology will be employed. The study will begin with a comprehensive review of existing literature on code similarity detection, focusing on both static analysis and runtime-based approaches. Subsequently, a series of experiments will be conducted to capture runtime data from a diverse dataset of code samples. Tools such as Java Fuzzing (JQF) and Java CoMmanD (JCMD) will be utilized to profile program execution and extract relevant runtime metrics. The collected data will then be analyzed and compared against results obtained from traditional static analysis methods.

1.0.6 Significance of the Study

This research holds significant implications for academia, industry, and the broader software development community. By advancing our understanding of runtime-based approaches to code similarity detection, the study aims to contribute to the development of more robust and accurate plagiarism detection tools. Furthermore, the findings of this research may inform pedagogical practices in computer science education, helping educators design more effective strategies for teaching and assessing programming skills. There have been many papers, Gipp and Meuschke (2011) which have attempted to improve solving more elaborate attempts at copying and modifying work. Additionally, insights gained from this research may have applications in software quality assurance and code review processes in industry settings.

1.0.7 Structure of the Thesis

The remainder of this thesis is organized as follows: Chapter 2 provides a detailed review of existing literature on code similarity detection, highlighting key concepts, methodologies, and research findings. Chapter 3 outlines the research methodology employed in this study, including data collection procedures, experimental design, and analysis techniques. Chapter 4 presents the results of the experiments conducted, including comparisons between runtime-based and static analysis methods. Chapter 5 discusses the implications of the findings, identifies limitations of the study, and suggests directions for future research. Finally, Chapter 6 offers concluding remarks and summarizes the key contributions of this research.

Chapter 2

Literature Review

In the realm of programming assignments, being able to detect similarities in work is important especially at the university level. Students are required to demonstrate their understanding of concepts and ability to solve problems. In an ideal world we could trust that each and every student upholds that goal of showcasing their knowledge, skills and experience gained throughout their time spent working on the module and course. That isn't the case all of the time and as such, there have been tools developed in order to prevent students submitting work that has not be properly cited or has been passed off as their own. To start off this literature review, I will be discussing a number of tools that are being used by academic institutions all around the world.

2.1 MOSS - Plagiarism Detection

MOSS is a tool that is used for automatic plagiarism detection, it was developed by Stanford in the mid 1990's. There is a paper on the ideas behind MOSS, Schleimer et al. (2003). MOSS relies on the winnowing algorithm which is one of many finger-printing algorithms available to find matching sequences in code. The winnowing algorithm is efficient and highly-scalable. MOSS takes a structure-based approach towards detecting similarities, which works even when comments on the program or the lines of the code have been rearranged. The system goes through a bundle of programs that been submitted and generates similarity scores, it is important to bear in mind that a pair of programs can have a high similarity score without plagiarism occurring. Currently, it can detect similarities in code for more than 20 different programming languages. In the paper MOSS is based of, it discusses the winnowing algorithm which MOSS makes use of in order to detect similarities between programs. **The process of how MOSS works can be broken down into four processes**.

1. Remove white space and identifiers from the document

2. Produce hash sequences from the k-grams derived from the pre-processed document.
   What is a K gram? k-grams refer to contiguous sequences of k characters within a document. These k-grams are used as the basis for generating hash sequences, which are then utilized to identify and compare similarities between documents. The k is an number, often ranging between [3 - 10].

3. Select a subset of these hashes to use as the document's fingerprint

4. Documents that have many matching fingerprints should be matched together and flagged.

The **third** part of the process involves the winnowing algorithm which is fingerprint selecting algorithm that applies a sliding window of size w to the hashes. If the smallest rightmost value has not been recorded, it records it then slides on until the end of the sequence of hashes. After completing this the hash values that have been recorded, form the document's fingerprint. Figure 2.1 shows an example of document fingerprinting.

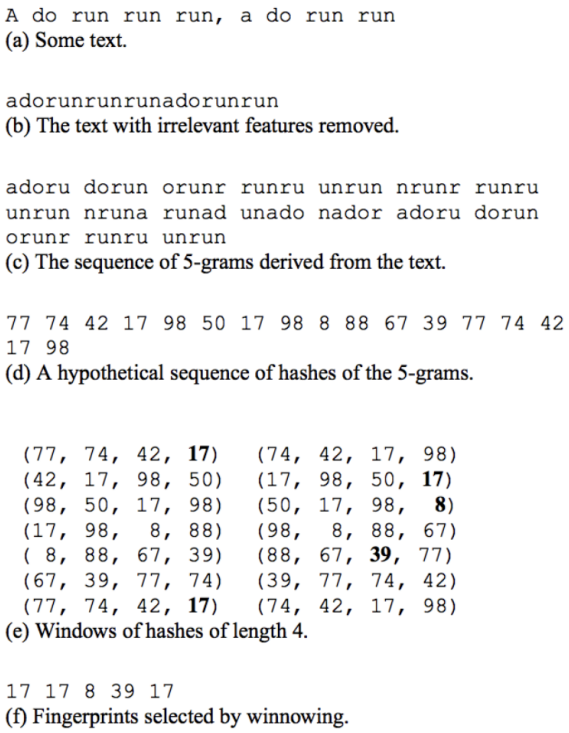
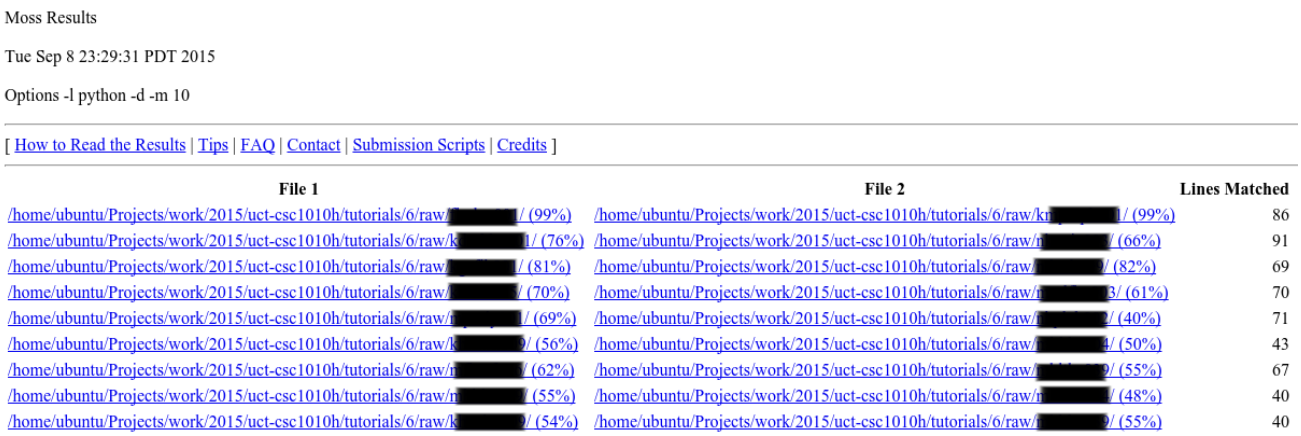


Figure 2.1: The process of document fingerprinting using the winnowing algorithm

Source: https://yangdanny97.github.io/blog/2019/05/03/MOSS

2.1.1 Applications of MOSS:

From an academic standpoint, a lecturer or a teaching assistant having over two hundred assignments to mark it can be difficult to notice the similarities while grading a bundle of assignments. In this case, the tool can be used to detect similarities between students and also past assignments given. From my research I have found that there are multiple ways to use MOSS, some involve using the command line, others use a graphical interface which can be more intuitive to those with no prior experience scripting. MOSS is provided as an internet service and requires you to email in order to obtain an account. After creating an account you can submit queries to MOSS, and the server will respond with a URL. Figure 2.2, displays the results of a query that has been submitted to MOSS. This shows the files that are most similar, to what percentage and the number of lines that match. After viewing the results of the query, users have the option to click on any pair of programs display and this will open up a side by side comparison of the pair of programs which can be used for further inspection to determine if the similarity between programs was due to work being passed off.



Moss Results

Tue Sep 8 23:29:31 PDT 2015

Options -l python -d -m 10

[How to Read the Results | Tips | FAQ | Contact | Submission Scripts | Credits]

File 1	File 2	Lines Matched
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (99%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (99%)	86
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (76%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (66%)	91
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (81%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (82%)	69
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (70%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]3/ (61%)	70
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (69%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]2/ (40%)	71
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (56%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]4/ (50%)	43
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (62%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]2/ (55%)	67
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (55%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (48%)	40
/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]/ (54%)	/home/ubuntu/Projects/work/2015/uct-csc1010h/tutorials/6/raw/[redacted]0/ (55%)	40

Figure 2.2: The results of MOSS query from the URL provided by the server

Source: Link to Figure 2.2 and 2.3

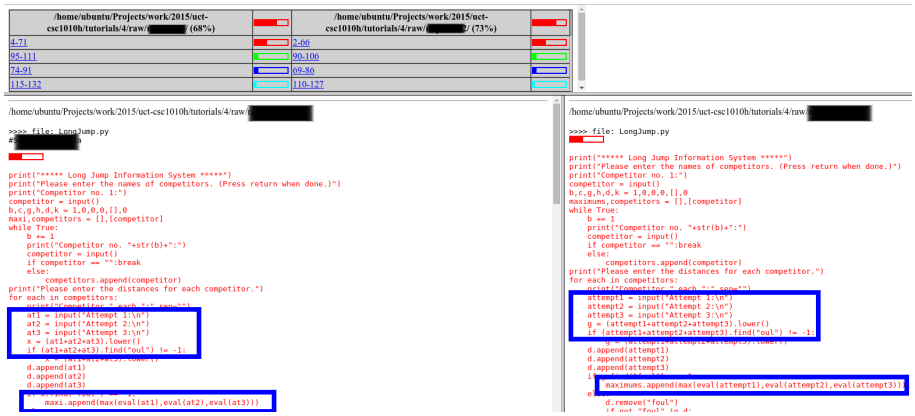


Figure 2.3: Page shown after clicking a pair of programs with a high similarity percentage

2.2 Compare50

Compare50 is another plagiarism tool on the market and serves as an alternative to MOSS. Compare50 is a tool that was developed for a computer science module (CS50) in Harvard Malan et al. (2021). It was one of many open-source tools created in order to facilitate to learning of students and lecturer's ability to grade and check assignments for plagiarism. Similarly to MOSS, it can detect similarity for different programming languages. As it stands, it can detect up to 300 different programming and templating languages. It is used from the command-line and can be installed using the python installer **"pip install compare50"**. The commands on the terminal are intuitive, making it a lot simpler to use for students and lecturers alike. Compare50 has five different comparison methods: **structure, text, exact, no comments and misspellings**, the first four make use of the winnowing algorithm. For the misspellings, they compare words in a document to the a English dictionary. It's the first of its kind and is used to aid discovering similarities in code. Combining these five different comparison methods gives a different variety to the ways you can highlight similarities in code and provides a more detailed insight into pairs of programs.

2.2.1 Applications of Compare50:

The applications of Compare50 exceed that of MOSS, which is due to the fact that it can compare similarities for not only programming languages but also templating languages. Compare50 can be used in educational institutions to evaluate the similarities between student's assignments not only for programs but dissertations, thesis' .etc. There are many online courses provided today and one of disadvantages of online courses is that it

is a lot easier to plagiarism code. Compare50 can be implemented to allow for the

detection of plagiarism in the submitted work. Since there are many languages available for comparison in Compare50, there are many applications for Compare50 aside from an educational aspect, such as using it to compare documents in a legal capacity. In order to evaluate the effectiveness of Compare50, it was tested locally and a comparison was ran between a bundle of binary search tree files that were written in java. On the terminal, Compare50 was installed and this command was ran in the directory that contained the bundle of files: **compare50 * -x "*" -i "*BST.java"**.
Fig 2.4 displays the output of terminal after running this command

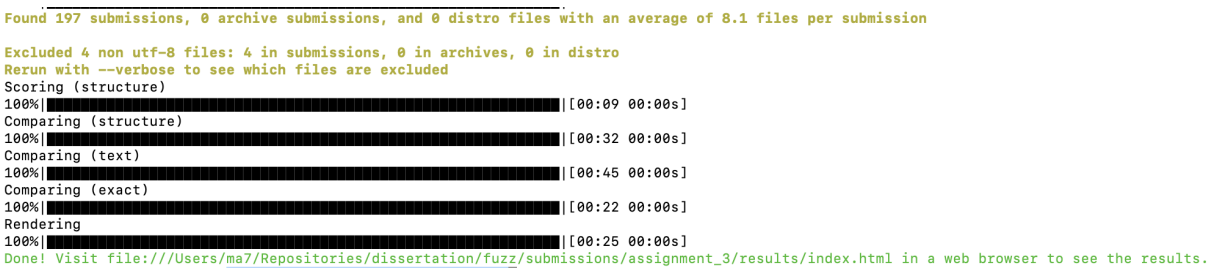
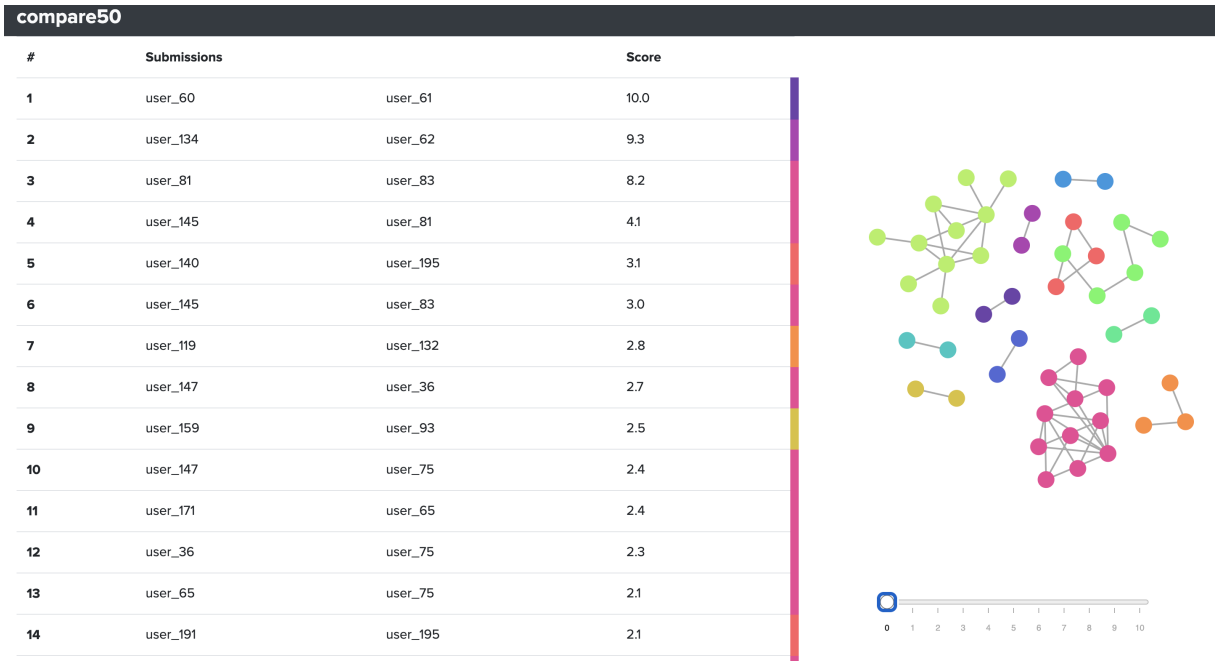


Figure 2.4: A screenshot of Compare50 running locally in the terminal

Source: Compare50 docs

After the execution of this command, a HTML file is created that can be opened in your browser that visualises the comparison between the bundle of files inputted. The next image 2.5 show's this visualisation



compare50

#	Submissions		Score
1	user_60	user_61	10.0
2	user_134	user_62	9.3
3	user_81	user_83	8.2
4	user_145	user_81	4.1
5	user_140	user_195	3.1
6	user_145	user_83	3.0
7	user_119	user_132	2.8
8	user_147	user_36	2.7
9	user_159	user_93	2.5
10	user_147	user_75	2.4
11	user_171	user_65	2.4
12	user_36	user_75	2.3
13	user_65	user_75	2.1
14	user_191	user_195	2.1

Figure 2.5: Outputted HTML file displayed on browser with comparison scores

There are a number of details given in this HTML file:

1. On the left is a list of user's whose submissions have the highest similarity scores in descending order

2. On the right is a graphical display of nodes with edges, the node signify a user, whilst the edge serves to display that nodes that are connected have some sort of similarity between the user's submissions

3. Below that graph is bar that filters the minimum value of the comparison score to display on the graph and list. If you changed the value to ten, then we would only see one comparison score the first one as that is the only one with a similarity score of ten.

This HTML file is not the only file outputted from the console, but for each match we get a match.HTML file which shows what lines of the code are similar. With many programs and documents, it is not absurd to see lines that are similar especially if it there is already a layout or structure that has been defined by a lecturer before giving the assignment to his students. An example of what this looks like can be seen below in Fig 2.6

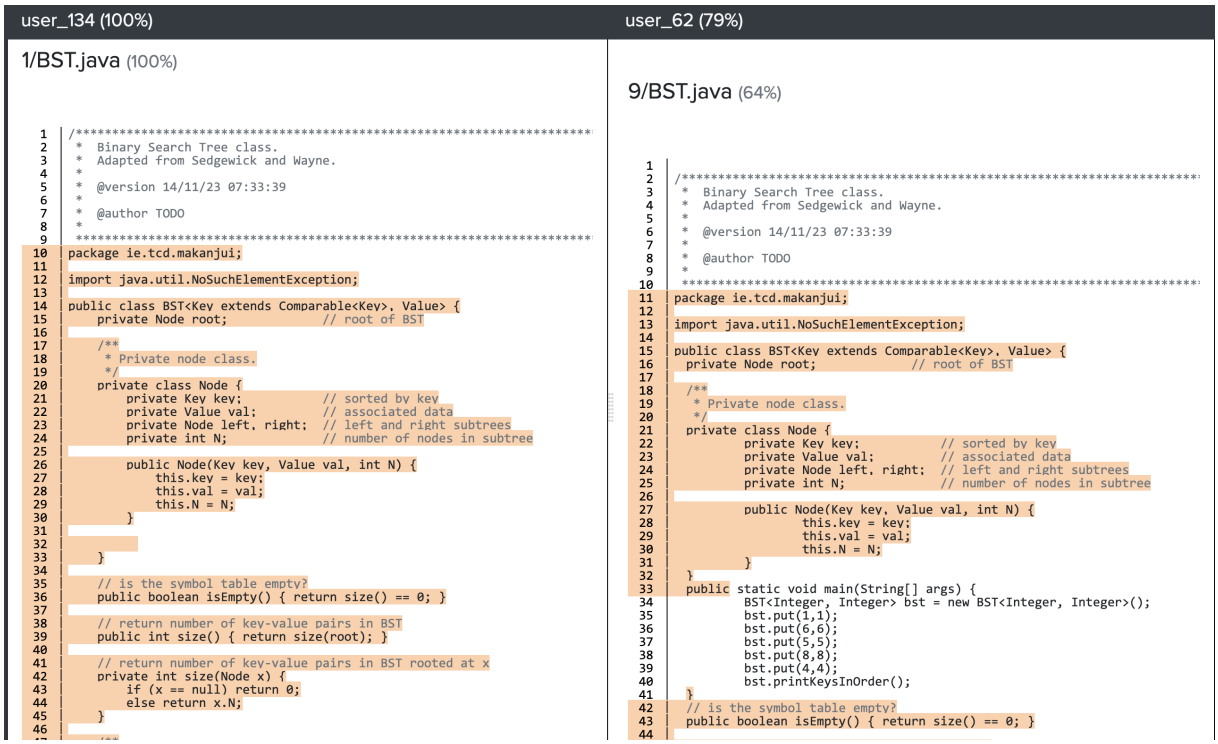


Figure 2.6: Matching sequences of code between two users

2.3 JPlag

JPlag is another tool used for detecting similarities between programs, it was "developed in 1996 by Guido Mahlpohl and others at the chair of Prof. Walter Tichy" at Karlsruhe Institute of Technology (KIT). It does not support as many programming languages as the previous tools but has the main languages that most programmers would use especially at an academic level. The jar file can be downloaded from the JPlag repository, which you can then call on the command line. Unlike MOSS and Compare50, JPlag uses a different type of algorithm for detecting similarities. The **Greedy String Tiling** algorithm is the architecture behind JPlag, this algorithm identifies similar areas between two strings by finding the longest common sub strings iteratively. It scans the source string and compares it with the target string, and selects the longest common sub-string at each step. This

process continues until the entire source is scanned. The algorithm outputs a set of matches or similarities between the two strings. JPlag has some features that separate it from other plagiarism tools aside from making use of a different type of algorithm they also allow for bundle submissions of files to have a **base code file**, this tells the architecture to ignore similarities between two files that share the same similarities with that base code file. This feature ensures that the similarity score is not impacted by code that had been pre-written i.e. a main function to test their (student's) code. The paper that was written on this tool was Prechelt and Malpohl (2003). JPlag can be applied in other areas aside from an academic standpoint having been used lawfully in intellectual property cases where it has been used by expert witnesses according to their site

2.3.1 Applications of JPlag:

There are a number of features available that are unique to JPlag. As mentioned before after downloading the jar file you can make use of it on the command line, but there is also the option of using JPlag programmatically. Using this API, it provides ways for users to creatively apply similarity detection for submissions of code. The image provided below is an example of how you may make use of this API.

Listing 2.1: **Example of API call in Java**

```
JavaLanguage language = new JavaLanguage();
language.getOptions(); //Use this to set language specific options
Set<File> submissionDirectories = Set.of(new File("/path/to/rootDir"));
File baseCode = new File("/path/to/baseCode");
JPlagOptions options = new JPlagOptions(language, submissionDirectories,
Set.of()).withBaseCodeSubmissionDirectory(baseCode);

try {
    JPlagResult result = JPlag.run(options);

    // Optional
    ReportObjectFactory reportObjectFactory = new
    ReportObjectFactory(new File("/path/to/output"));
    reportObjectFactory.createAndSaveReport(result, "/path/to/output");
} catch (ExitException e) {
    // error handling here
}
```

Source: JPlag Github

To execute the JPlag jar file on the command line interface, we use the command: **java -jar jplag.jar path/to/the/submissions**. Which can be set with additional parameters, for setting the programming language, the base code file, the mode .etc. After the execution has been completed successfully a zipped result file is returned which can be viewed using JPlag's report viewer on **jplag.github.io/JPlag/**. This zipped result file contains JSON formatted files that detail the overview of the submission, the identification numbers given to each file in the submission, the options chosen either on the command line or programmatically, a JSON file for each comparison and a vue file for a visual display of the overall report. The next figure 2.7 shows an example of what the JPlag report can look like.

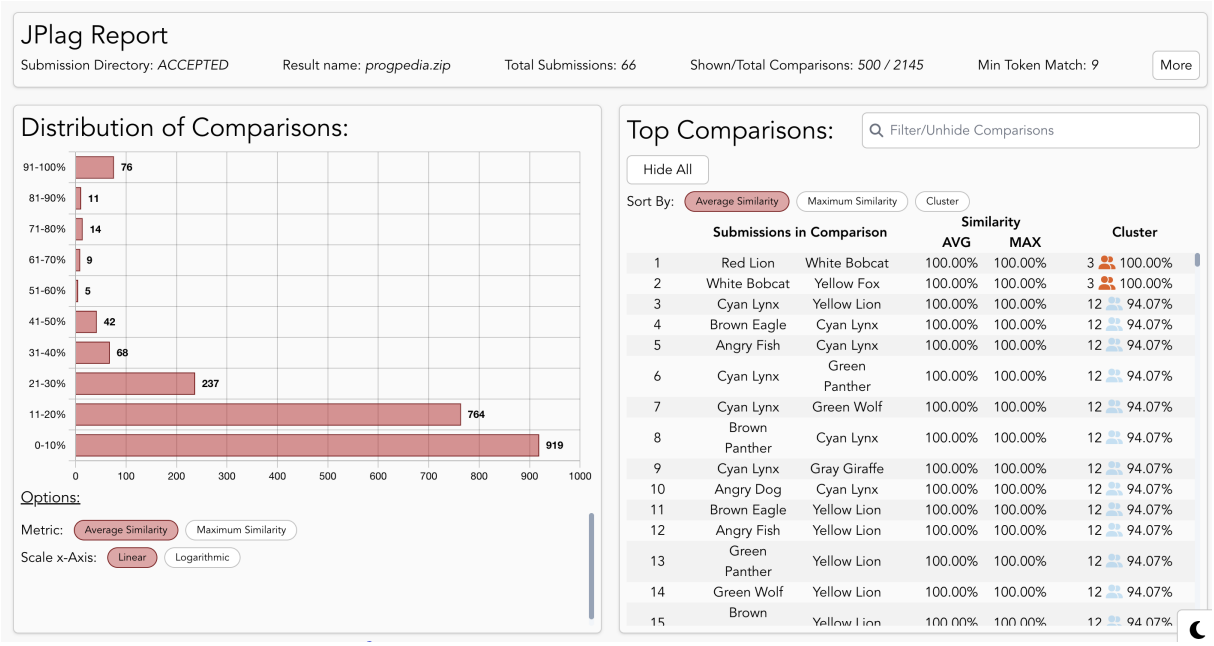


Figure 2.7: An example of what a JPlag report can look like

This report provides options to change the metrics from average similarity to maximum similarity, in most cases the average similarity would be used but if you had many programs with different length the maximum similarity will provide a better insight into the pairs of programs. You can change the x-axis scale from linear to logarithmic depending on your requirements. On the right, it shows the top comparisons, with the name of the students, the similarity scores for the average and maximum similarity and the cluster. The cluster details the group of submissions that share the same score. At the top of the report, you can see the total submissions, the number of comparisons shown against the total number of comparisons and a minimum token match, which is a parameter that can be tuned that adjusts the sensitivity of the comparisons made by JPLag.

What separates JPlag from the other plagiarism tools that have been discussed, lies in its ability to detect disguised plagiarism which can take on the form of:

1. Inserting comments or empty lines

2. Changing function or variable names

3. Addition of useless lines of code or changing lines of code

4. Restructuring the program flow

5. Changing control structures, if statements to switch cases .etc.

6. Modifying expressions (X ¿ Y) to !(X ¡= Y)

7. Statements in code being merged or split

Using the end-to-end testing, JPlag can detect these changes of code that may deceive other plagiarism tools. This requires user's to develop a testing strategy and can be structured using the JPlag API which would require a certain level of understanding of how the JPlag architecture works and programming knowledge.

Listing 2.2: Original Code

```
private final <T> void swap(T[] arr, int i, int j) {
    T t = arr[i];
    arr[i] = arr[j];
    arr[j] = t;
}
```

Listing 2.3: Plagiarized code

```
private final <T> void paws(T[] otherArr, int i, int j) {
    T t = otherArr[i];
    otherArr[i] = otherArr[j];
    otherArr[j] = t;
}
```

The above listings show an example of plagiarism code, that was changed by changing both the variable and function names. The output shows that the result of the match is 100% signifying the tool's ability to detect disguised plagiarism. Listing 2.4 shows the results in the format of a JSON.

Listing 2.4: Results from JSON file

```
"SortAlgo-SortAlgo1" : {
  "minimal_similarity" : 100.0,
  "maximum_similarity" : 100.0,
  "matched_token_number" : 56
},
```

2.4 Effectiveness and Limitations

2.4.1 Effectiveness

Plagiarism checkers of this type are highly effective due to their ability to detect potential instances of code plagiarism with a good degree of accuracy. By providing in-depth analysis of the code structure and matching code segments, these tools enable educators and researchers to identify similarities between different code submissions, even in cases where the code has been slightly modified. This capability is particularly valuable in educational settings, where maintaining academic integrity is paramount. One of the key advantages of these plagiarism checkers is their support for a wide range of programming languages. This versatility allows users to compare code written in different languages, making the tools applicable across various programming courses and disciplines. Whether students are working with Java, Python, C++, or any other programming language, these tools provide a consistent and reliable means of detecting code similarities. Moreover, these plagiarism checkers play a crucial role in upholding the integrity of courses and academic programs. By identifying instances of code plagiarism, educators can address academic misconduct and ensure that students are evaluated fairly based on their own original work. This helps maintain the credibility and reputation of educational institutions and promotes a culture of academic honesty and integrity among students. One of the most significant benefits of these tools is their ability to save users a significant amount of time that would otherwise be spent manually checking code submissions for similarities. By automating the process of plagiarism detection, these tools streamline the evaluation process and allow educators to focus their time and resources on other aspects of teaching and research. By making use of the implementation of advanced algorithms such as the winnowing algorithm. This algorithm has proven to be highly efficient and effective in identifying similarities between code segments, even in cases where the code has been intentionally modified. By utilizing such algorithms, these tools are able to achieve a high level of accuracy and reliability in detecting code plagiarism. In particular, JPlag stands out for its unique architecture and detection mechanisms. Unlike other plagiarism

checkers, JPlag employs a diverse range of detection methods, allowing it to detect various forms of plagiarism with precision. This versatility makes it significantly more difficult for students to circumvent detection and copy work from others, thereby enhancing the effectiveness of plagiarism prevention efforts. Importantly, the capabilities of these plagiarism checkers are not limited to academia alone. While they are commonly used in educational settings, their effectiveness and utility extend to other areas also. For example, these tools can be applied in software development environments to detect code reuse and identify potential copyright infringement issues. By leveraging the capabilities of these tools in diverse contexts, organizations and individuals can safeguard their intellectual property and ensure compliance with legal and ethical standards. In summary, the effectiveness of plagiarism checkers in detecting code similarities lies in their comprehensive analysis capabilities, support for multiple programming languages, and utilization of advanced detection algorithms. These tools play a critical role in upholding academic integrity, saving time for educators, and promoting fairness in the evaluation process. Moreover, their versatility and applicability extend beyond academia, making them valuable assets in various professional and organizational contexts. Sources taken from Novak et al. (2019) and Ahadi and Mathieson (2019).

2.4.2 Limitations

Even with the abundance of benefits, there are some limitations and disadvantages. The algorithms employed can flag content that is unique as plagiarized, paraphrasing (changing variable names) can cause the algorithm to miss plagiarized content. False positives occurring from code that the user may not have written themselves but possibly came with the assignment specification. Another challenge arises from the presence of false positives resulting from code the user may have obtained from legitimate sources. These false positives undermine the credibility of plagiarism detection results and complicate the process of assessing academic integrity. There are tools available online that are able to defeat these plagiarism tools, an example of this is MOSSAD which has developed an algorithm that takes advantage of MOSS' in order to manipulate the comparison score. These tools leverage sophisticated algorithms to manipulate comparison scores and evade detection, posing a significant challenge to the effectiveness of existing plagiarism detection methods. The development of such tools highlights the battle between plagiarism checkers and individuals seeking to circumvent them, underscoring the need for continued innovation and improvement in this field, MOSSAD is further discussed in this paper Devore-McDonald and Berger (2020). In addition to the general limitations of plagiarism checkers, here are also some limitations specific to JPlag, which include the fact that it

does not have much support for detecting similarities for ordinary text, Tuli (2016), which limit it's capabilities in certain contexts. With the many features available to JPlag, in order to best make use of the tool you require an understanding of the architecture and the ability to extend the tool using their API (Application Programming Interface) call. This limits the effectiveness of the tool to user's who understand the tool deeply and know how to best make use of it. This reliance on technical expertise may hinder the tool's accessibility and usability for users who are not familiar with its inner workings.

Looking at the current landscape of code plagiarism, with the emergence of artificial language learning models such as Chat-GPT and its variants, they have the capability to generate code and alter existing code in ways that can deceive traditional plagiarism checkers. A review on the different tools in relation to the emergence of artificial language learning models was spoken about in Omar et al. (2024) Multiple sources online show how these learning models can be provided code and with certain creative prompts will create and alter code to avoid detection by plagiarism tools. This presents a significant challenge for plagiarism detection efforts, as it is becoming increasingly difficult to distinguish genuine code and code that has generated or been manipulated. As a result, this has called for more innovative approaches to similarity detection.

Source: Chat-GPT beating MOSS

Chapter 3

Methodology

3.1 Overview

The current state of architecture for similarity tools focus on employing various algorithms and techniques to identify similarities between text and code. These can range from tokenization to string matching. These methods while proven to work are not immune to false positives and code can be altered to avoid detection. As programs execute, when this information is profiled i.e. determining where a component method or line is consuming the most resources this information reflects the behavior and characteristics of the program's execution. Unlike code itself, runtime information is more difficult to alter or obfuscate, as it is generated dynamically during program execution. This formed the basis of the research undertaken in this study, to investigate the possibility of leveraging runtime information from programs to detect similarities amongst programs. This is a departure from traditional methods of identifying code similarities. By exploring the potential of runtime data, the aim is to redefine the applications of existing similarity detection tools, thereby enhancing their effectiveness and applicability in diverse contexts. The primary objective of this chapter is to discuss the methods employed for extracting the runtime information and to highlight the attempt of utilising this data for identifying similarities among programs. The data that is extracted at runtime offers insights into the actual behavior and execution dynamics of the program, providing a more total view of a program's functionality and behavior. This chapter will delve into previous attempts made at leveraging runtime information for similarity detection and the shortcomings of those methods. There has been an attempt to utilize runtime-based information for program characterization in an attempt to discover similarities in code, Jhi et al. (2015), but the methodology approach taken here is novel as far as we know. In essence the methodology, will serve as a foundational exploration of the novel approach proposed by

the research question and lay the groundwork for future discussions on this topic.

3.2 Data Collection

Given that plagiarism detection tools are mainly used in academic settings, it was important to utilize student assignments as dataset for testing this methodology. The dataset was taken from Trinity College, specifically sourced from the "Algorithms and Data Structures" module. This aim of the module is for students to learn how to write effective and efficient programs using common data structures and algorithms In total, the dataset comprised over 250 assignments, each representing a student's implementation of functions for a binary search tree. The use of these assignments was deliberate as it provided a common ground for comparison, as all assignments were tasked with achieving the same objective although through potentially different approaches. The focus on Java assignments was crucial since java programs generate a lot of runtime data during execution and since the automation tool "Maven" was to be used this was the perfect choice.

3.2.1 Data Pre-processing

To uphold data protection standards and ensure the anonymity of students, the data was anonymized discarding names and numbers of students associated with each assignment submission. To achieve this, a python script was created to systemically anonymize the identifying information in the dataset. The script iterated through each submission, replacing the original student names with identifiers in the format "user_n", where 'n' represented a unique number for each student. For instance the 50th iteration of the anonymization process would assign the identifier "user_50" to the corresponding assignment submission. It's important to note that some students may have submitted multiple submissions for this assignment. To accommodate this scenario, the script was designed to handle multiple submissions from the same student so that each submission would be regarded as being made by the same user.

3.3 Maven

Maven is a build automation tool used for java projects. It enables developers to automate essential tasks such as project compilation, testing, packaging and deployment. One of the key advantages of Maven lies in its ability to manage project configurations and dependencies through a standardized format defined in the project's pom.xml file, the pom file used during this research is shown in .4. For this project we created the

maven project by installing maven and running this command in the terminal **mvn archetype:generate -DgroupId=com.mycompany.app -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DarchetypeVersion=1.4 -DinteractiveMode=false**. These parameters have an impact on the file structure for where the classes and test classes are stored, the package named associated with the classes is determined by "-DgroupId", in this case the top of each java class would begin with the following package name.

Listing 3.1: -DgroupId

```
package com.mycompany.app;
```

Using the command **mvn package** in the terminal at the directory of the maven folder, will compile your java code, run any tests, and finish by packaging the code up in a JAR file within the target directory. Maven's plugin architecture provides an extension to the functionality of the project as per the requirements of the project determined by the user. This research employed a Fuzz testing library mentioned in the next section, which required the addition of this snippet 3.2 below in the pom.xml file in order to add the library as a dependency to the project.

Listing 3.2: Fuzz testing POM dependency

```
<dependency>
  <groupId>edu.berkeley.cs.jqf</groupId>
  <artifactId>jqf-fuzz</artifactId>
  <version>2.0</version>
</dependency>
```

3.4 Fuzz Testing

Upon collecting the bundle of assignments, the next pivotal phase involved rigorously testing the programs to assess their functionality and performance. This process had an imperative need for randomized testing inputs. The paper Doi (2023) discusses regression testing via fuzz testing making use of the library to be mentioned. The random inputs were important to verify that the function calls made by the program exhibited unpredictable behavior. Randomized testing inputs was a means to evaluate the robustness and versatility of the programs. Using these rigorous tests ensured that the runtime

Source: Learn Maven

information given for each program was large, large enough to analyse and discuss further. In order to accomplish this we made use of a fuzz testing framework called **JQF + Zest: Coverage-guided semantic fuzzing for java**. A java class was created in order to generate random inputs for the assignments. Padhye et al. (2019) discusses the techniques proposed of how to implement these generator classes and how to use them, exploring this paper detailed how to create a generator class and the capabilities of JQF. The binary search tree that students were asked to implement was generic, meaning the tree could be instantiated with any object and for this research we focused on an integer key and an integer value which is shown by listing 3.3 below.

Listing 3.3: **New Binary Search Tree**

```
BST<Integer, Integer> bst = new BST<>();
```

The generator had been created to instantiate an array that can hold a thousand objects and return it with random values. With this array, we loop through each position in the array and add a random integer value ranging from [-10, 10]. The purpose of this is to ensure while the values in the array are random but are not wildly varying to maintain some sort of control for each of the programs. Using the maven plugin, the programs can be tested using the command **mvn jqf:fuzz**. This can be altered and additional commands can be added to control the fuzzing session such as the duration of time the fuzzing runs for; the number of executions of each program; an option to save the random inputs generated which can be used again for other fuzzing sessions. Combining these commands, the seed of the inputs were saved to ensure all the programs were tested by the same randomly generated inputs which served as another control and the number of executions of each program were limited to a hundred trials

Listing 3.4: **Array Generator Function**

```
@Override
public Integer[] generate(SourceOfRandomness random,
GenerationStatus __ignore__)
{
    Integer[] array = new Integer[1000];
    for(int i = 0; i < array.length; i++)
    {
        array[i] = (Integer) random.nextInt(-10, 10);
    }
    return array;
```

Source: JQF Library

}

3.4.1 Test Class

A test class was created which employed the executions of tests by generating arrays populated with random input data. These arrays fed randomized values into the binary search tree. The objective of this strategy was to thoroughly check the functionality of the binary search tree across a diverse range of inputs. To elaborate further on the testing process, lets delve into the mechanics. Each array generated by the test class which was obtained from the array generator mentioned earlier contains a sequence of randomly generated values. These values are then processed iteratively by the binary search tree with each iteration invoking a distinct method based on the remainder obtained from diving the current array element by three. The division operation yields a remainder of either zero, one or two, which will dictate the specific method to be called on the binary search tree. When the remainder is zero, the test class triggers the call of the "contains" method on the binary search tree. This function evaluates whether the current array element exists within the binary search tree and returns a boolean value indicating whether it is in the tree or not. When the remainder is one, the test class calls the "put" method on the binary search tree, this will insert the current array element into the tree structure. This method modifies the internal state of the tree and does not return any value. When the remainder is two, the test class will call the "get" method of the binary search tree. This method retrieves the value associated with the specified key (in this case, the array element) from the binary search tree, enabling the retrieval of stored values based on their corresponding keys. After each iteration the array, the test class increments the array index by three. This ensures that the subsequent method calls operate on distinct elements within the array. This process continues until all elements in the array have been process. By using this testing approach, each implementation of the binary search tree will leave performance information that is extracted for further inspection

3.5 Profiling

The fuzzing process serves a crucial role in generating runtime information for subsequent analysis. After each fuzzing session, we need to extract the runtime information that had been generated. To begin, prior to initiating each fuzzing session, the paths of the program submissions are recorded in a designated text file. Methods to profile are discussed in this paper Hasan et al. (2016), which were made use of in the previous methodologies

attempted. This file provides a way to associate the runtime information generated during the fuzzing process with the respective program submissions. To ensure consistency and accuracy, each the path in the file is updated dynamically, with each fuzzing session overwriting the paths of the previously processed submissions. Having this information, the next step involved extracting and preparing each of the java class files which contained the implementation of the binary search tree for each user. This file is extracted from the submissions folder, where all user programs are stored and copied into the designated maven project folder. This step allowed for each binary search tree implementation to be readily accessible and integrated into the maven project environment for subsequent fuzzing. Once the files are in place, the fuzzing process is initiated by executing the test class against the binary search tree class using the maven command **mvn jqf:fuzz**. This command is altered with addition parameters to customize the behavior of the fuzz framework, which included specifying the use of a random seed input for all programs and setting the number of executions for each trial to a hundred. These parameters were essential to ensure consistency and repeatability across the sessions, allowing for accurate profiling and analysis of program behaviors. To streamline and automate the entire fuzzing process, a bash script is employed. This script begins the sequential execution of the previously mentioned steps, automating these tasks enhanced reproducibility and ensure minimal manual intervention was required.

3.5.1 Extracting runtime information

The runtime information was created by starting a timer before a function call to the binary search tree was made, this was done for all methods observed (contains, get, put) and after the method call we stopped the time. This observed time was then stored inside a csv file, after every iteration execution of the test class i.e A single trial (Listing 1). Each entry in the csv file comprised three distinct components: the path indicating the origin of the program within the assignment bundle, a numerical identifier ranging from 0 to 2 corresponding to the specific method call (contains, get, put) and an accumulated average score representing the execution time of each method call. With the execution of 100 trials for each submission and a bundle of assignments containing over 250 submissions, the cumulative output resulted in an extensive dataset comprising over 40,000 lines within the csv file. Various scripts were ran in order to derive meaningful insights from the raw runtime data. This method of capturing and analyzing the runtime metrics lead to potential implications for identifying similarities between programs.

3.5.2 Post-processing runtime information

As previously mentioned after the entire fuzzing session had complete and the runtime information had been extract we were left with voluminous dataset. A post-processing step was essential to condense the data and extract useful insights from it. This phase involved reducing the dimensions of the dataset and calculating the average metrics to facilitate subsequent analysis. To achieve this, a python script was employed to streamline the data processing workflow. The initial step involved reading the raw data containing the runtime information. The script iteratively processed each entry in the dataset and populated a dictionary to aggregate the execution times of method calls for each program iteration. For every iteration of a program, the script calculated the average execution time for each method call, this resulted in a condensed representation of the runtime information. Upon completion of this process, the script generated a csv file containing the averaged method call numbers for the total number of trials. Since some users had submitted multiplied assignments, the dataset still comprised multiple entries for these users. To address this, a subsequent python script was created to iteratively process the intermediate csv file and calculate the average performance metrics for each user. This involved determining the number of submissions contributed by each user and computing the average performance metrics across all their submissions. This script generated a refined dataset that encapsulated the average method call numbers for each user.

3.6 How it was evaluated

After completing the fuzz testing and having extracted the runtime information, this data need to be analysed in order to determine if there was any signal given by the runtime data that would indicate similarity between the programs. There were two distinct approaches employed to evaluate the extracted runtime information. Firstly, an examination was conducted on the functions within the binary search tree programs of users who exhibited similar runtime scores. This approach aimed to identify any discernible patterns in code structures of users. By examining the code snippets of these users, the goal was to ascertain whether similarities in runtime behavior corresponded to similarities in code implementation. A comparative analysis was also conducted on programs from users with dissimilar runtime scores. This served as a control mechanism to establish a baseline for understand the relationship between runtime scores and code similarity. The aim was to discern whether variations in runtime behavior were reflective of differences in code implementation.

In addition to the internal analysis of runtime data, an external evaluation was also

taken using the Compare50 tool. This involved subjecting the bundle of assignments to the Compare50 algorithm to identify similarities between programs based on traditional approaches. Similarly, the similarities detected by Compare5o were juxtaposed with the findings derived from the runtime-based methodology. This comparative assessment aimed to discover the efficacy of the runtime-based approach by validating its outcomes obtained against the ones through conventional similarity detection mechanisms.

By adopted a multi-faced approach, a comprehensive evaluation on the effectiveness and reliability of leveraging runtime information for identifying code similarities was created. The overall aim was to gain deeper insight into the potential of runtime information based methodologies in the area of code similarity detection. This approach lead to highlighting strengths and limitations of the proposed methodology, informing future research directions on this topic.

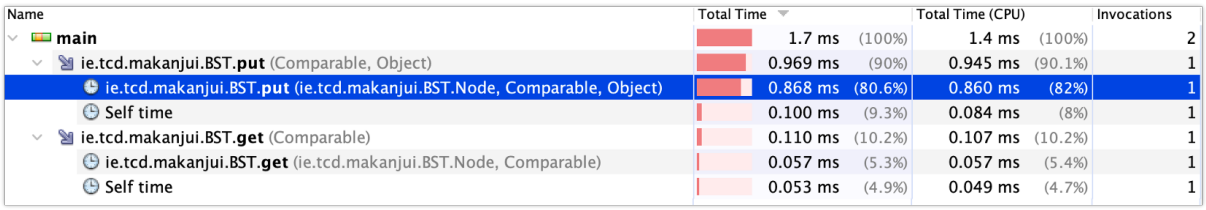
3.7 Other Methods Tried

3.7.1 JQF + VisualVM

A study on improving component coupling information by dynamic profiling was undertaken which made use of VisualVM's capabilities on eclipse, Kakarontzas and Pardalidou (2018). This was the first attempt at approaching the research question, it involved making use of the fuzzing framework library mentioned before and VisualVM which is a visual interface for viewing information about java applications. Detailed information about java programs were monitored and this could be extracted as a snapshot of the runtime information of the program. The information showed memory leaks, the heap data, the garbage collector and the profiling information obtained from the CPU. The profiling information from the CPU was what was focused on, it showed the percentage of the time of the CPU each method call used. After beginning a fuzzing session, the interface would then be used to begin capturing the java application and after the application terminated the interface would capture a snapshot. With this method of approach, the issue that occurred was that fuzz testing framework was being profiled instead of the binary search tree application. This meant that the snapshots did not reflect the usage of the CPU by the users program, but the fuzz testing session that was running to test the users program. This did give an initial idea on how to approach the problem and gave more shape to what metrics we might be interested in and the others that we could discard.

3.7.2 Hard-Coded JUnit Input + VisualVM

Building on the first approach, knowing that the fuzz testing library was obstructing VisualVM's capability to profile information. The framework was scrapped and instead we relied on using JUnit testing with hard coded input that would replace the array generator created by JQF. This did allow for VisualVM to correctly profile the binary search tree application, and resulted with a snapshot of the program shown in figure 3.1.



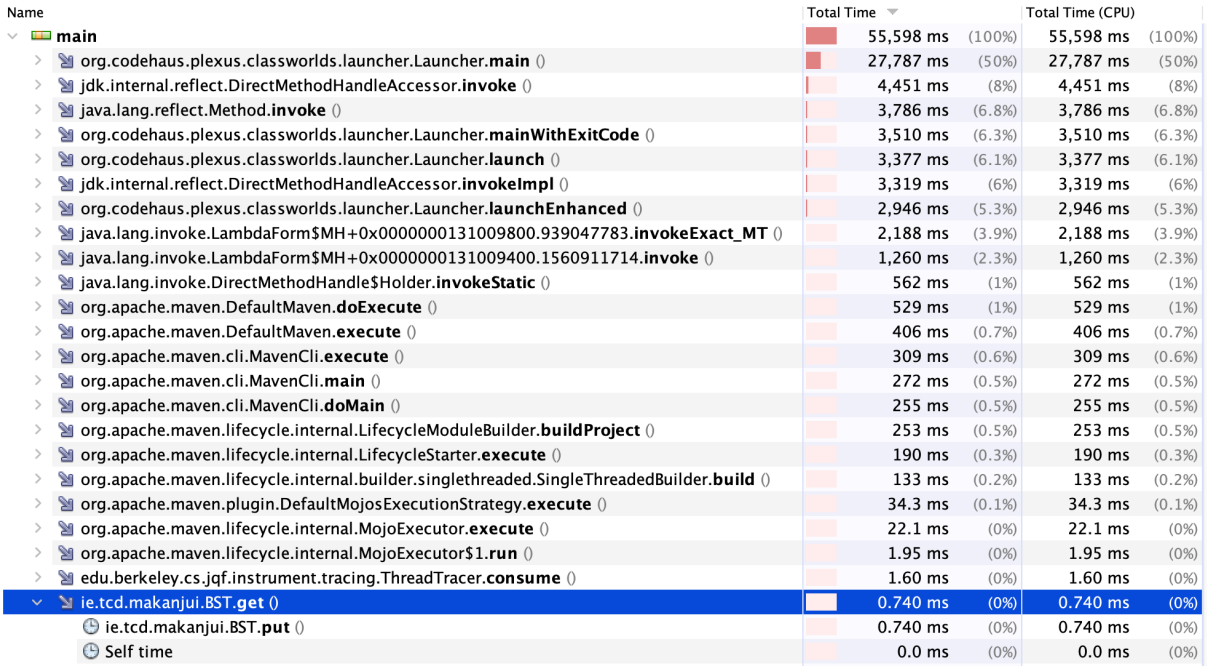
Name	Total Time		Total Time (CPU)		Invocations
main	1.7 ms	(100%)	1.4 ms	(100%)	2
ie.tcd.makanjui.BST.**put** (Comparable, Object)	0.969 ms	(90%)	0.945 ms	(90.1%)	1
ie.tcd.makanjui.BST.**put** (ie.tcd.makanjui.BST.Node, Comparable, Object)	0.868 ms	(80.6%)	0.860 ms	(82%)	1
Self time	0.100 ms	(9.3%)	0.084 ms	(8%)	1
ie.tcd.makanjui.BST.**get** (Comparable)	0.110 ms	(10.2%)	0.107 ms	(10.2%)	1
ie.tcd.makanjui.BST.**get** (ie.tcd.makanjui.BST.Node, Comparable)	0.057 ms	(5.3%)	0.057 ms	(5.4%)	1
Self time	0.053 ms	(4.9%)	0.049 ms	(4.7%)	1

Figure 3.1: A screenshot of a snapshot of the CPU profiling of the BST class

This image shows what methods are called by the class and what percentage of the CPU is being used by each method call. With this information, the next step was to then automate the process for every program in the file and then extract the profiled runtime information from VisualVM. This was not possible with the interface, and the idea to hand-write each number was considered briefly but quickly turned down as it was not feasible for the number of programs that needed to be profiled. In an attempt to figure out a way to solve this issue, an extension of VisualVM was used GraalVM but to no effect. This resulted in this approach being abandoned.

3.7.3 JQF + JCMD

Taking the failure of both approaches into consideration, multiple profiling tools were considered in order to be able to discover a way to profile information and extract that information. With this, we would be able to automate this process for all the programs. Ideas were obtained from the Yin et al. (2018), where they applied different profiling methods to obtain useful data from their applications and they extended the functionality of JCMD. JCMD is a command line tool that sends diagnostic requests to the java virtual machine (JVM) which enables the execution of java byte-code essentially an interpreter of java programs for the hardware of a machine. This was the command used in order to run the jcmd on the terminal**jcmd ¡process id/main class¿ ¡command¿** [**options**], it had additional parameters that could be added to the command. In order to utilise this command, a fuzzing session would be started followed by using the jcmd command on terminal with the process number of a the java application. This would begin the profiling process and once the java application terminated, the jcmd command terminates too. After a Java Flight Recorder (jfr) file would be outputted to the folder containing the java class. This file is then read on VisualVM, which displayed the profiling information shown in figure 3.2. This information was similar to the There were two issues with this approach, the first being it took in the CPU usage of the fuzzing library which has an effect on how the data is shaped although this may not have been a massive issue since all the programs would share the same problem. The second issue was that the data could not be extracted in the form desired such as on a spreadsheet, csv file or even a text file. All these approaches led to the methodology discussed above, although not ideal it did provide a starting point to answering the research question posed.



Name	Total Time		Total Time (CPU)	
main	55,598 ms	(100%)	55,598 ms	(100%)
org.codehaus.plexus.classworlds.launcher.Launcher.**main** ()	27,787 ms	(50%)	27,787 ms	(50%)
jdk.internal.reflect.DirectMethodHandleAccessor.**invoke** ()	4,451 ms	(8%)	4,451 ms	(8%)
java.lang.reflect.Method.**invoke** ()	3,786 ms	(6.8%)	3,786 ms	(6.8%)
org.codehaus.plexus.classworlds.launcher.Launcher.**mainWithExitCode** ()	3,510 ms	(6.3%)	3,510 ms	(6.3%)
org.codehaus.plexus.classworlds.launcher.Launcher.**launch** ()	3,377 ms	(6.1%)	3,377 ms	(6.1%)
jdk.internal.reflect.DirectMethodHandleAccessor.**invokeImpl** ()	3,319 ms	(6%)	3,319 ms	(6%)
org.codehaus.plexus.classworlds.launcher.Launcher.**launchEnhanced** ()	2,946 ms	(5.3%)	2,946 ms	(5.3%)
java.lang.invoke.LambdaForm$MH+0x0000000131009800.939047783.**invokeExact_MT** ()	2,188 ms	(3.9%)	2,188 ms	(3.9%)
java.lang.invoke.LambdaForm$MH+0x0000000131009400.1560911714.**invoke** ()	1,260 ms	(2.3%)	1,260 ms	(2.3%)
java.lang.invoke.DirectMethodHandle$Holder.**invokeStatic** ()	562 ms	(1%)	562 ms	(1%)
org.apache.maven.DefaultMaven.**doExecute** ()	529 ms	(1%)	529 ms	(1%)
org.apache.maven.DefaultMaven.**execute** ()	406 ms	(0.7%)	406 ms	(0.7%)
org.apache.maven.cli.MavenCli.**execute** ()	309 ms	(0.6%)	309 ms	(0.6%)
org.apache.maven.cli.MavenCli.**main** ()	272 ms	(0.5%)	272 ms	(0.5%)
org.apache.maven.cli.MavenCli.**doMain** ()	255 ms	(0.5%)	255 ms	(0.5%)
org.apache.maven.lifecycle.internal.LifecycleModuleBuilder.**buildProject** ()	253 ms	(0.5%)	253 ms	(0.5%)
org.apache.maven.lifecycle.internal.LifecycleStarter.**execute** ()	190 ms	(0.3%)	190 ms	(0.3%)
org.apache.maven.lifecycle.internal.builder.singlethreaded.SingleThreadedBuilder.**build** ()	133 ms	(0.2%)	133 ms	(0.2%)
org.apache.maven.plugin.DefaultMojosExecutionStrategy.**execute** ()	34.3 ms	(0.1%)	34.3 ms	(0.1%)
org.apache.maven.lifecycle.internal.MojoExecutor.**execute** ()	22.1 ms	(0%)	22.1 ms	(0%)
org.apache.maven.lifecycle.internal.MojoExecutor$1.**run** ()	1.95 ms	(0%)	1.95 ms	(0%)
edu.berkeley.cs.jqf.instrument.tracing.ThreadTracer.**consume** ()	1.60 ms	(0%)	1.60 ms	(0%)
ie.tcd.makanjui.BST.**get** ()	0.740 ms	(0%)	0.740 ms	(0%)
ie.tcd.makanjui.BST.**put** ()	0.740 ms	(0%)	0.740 ms	(0%)
Self time	0.0 ms	(0%)	0.0 ms	(0%)

Figure 3.2: A screenshot of a JFR snapshot of the CPU profiling of the BST class

Chapter 4

Evaluation

4.1 Results

4.1.1 Dataset

The extracted runtime information was formatted on a spreadsheet which was then organised into different spreadsheet tabs for further analysis. The first spreadsheet tab was the input data obtained from the post-processed csv file, shown in Table 4.1

User_Number	Method 0	Method 1	Method 2
6	0.002069	0.002058	0.00204837237
1	0.009589	0.00950548278	0.009521
2	0.007963	0.00789023994999999	0.007888
3	0.02365628741	0.023402	0.02341647106

Table 4.1: First four rows on the spreadsheet

The next spreadsheet tab contained pairs of submissions with their runtime score difference, this tab was created in order to allow for searching of pairs of submissions by users to determine what the difference in their runtime score was. This provided more context as to which pairs shared a low or high runtime score and upon recognizing the difference provided an opportunity to manually check the code for further inspection.

The final tab outputted the pair of user submissions with their runtime score difference for the 25 closest pairs. This showed which users had a runtime score most similar.

4.1.2 Comparison of average runtime scores

After processing the data, it was formatted onto a spreadsheet for analysis, and the subsequent findings are presented below. The spreadsheet was used to find the 25 closest

User_1	User_2	Difference
2	1	0.00487426156
3	1	0.04185947303
3	2	0.04673373459
4	1	0.02229259022

Table 4.2: First four rows on the spreadsheet

pairs of submissions, the users with the smallest difference according to their runtime scores were user 194 and user 136. The difference in their runtime score was 0.0000210625 seconds. Their code was then closely inspected to observe if there were any obvious similarities in their design.

Listing 4.1: **User 194 contains function**

```
public boolean contains(Key key)
{
    return get(key) != null;
}
```

Listing 4.2: **User 136 contains function**

```
public boolean contains(Key key) {
    return get(key) != null;
}
```

The similarities in code are quite observable and for this case, they both use the same variable names and formatting of return. The next step was to observe the get functions and how they were formatted.

Listing 4.3: **User 194 get function**

```
public Value get(Key key) { return get(root, key); }

private Value get(Node x, Key key)
{
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) return get(x.left, key);
    else if (cmp > 0) return get(x.right, key);
    else              return x.val;
}
```

Listing 4.4: **User 136 get function**

```
public Value get(Key key) { return get(root, key); }

private Value get(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) return get(x.left, key);
    else if (cmp > 0) return get(x.right, key);
    else              return x.val;
}
```

Similar results as the contains function, both their get methods use the same variable names and the conditional statements are set up in the same way. Lastly, the put function for both users needed to be looked at

Listing 4.5: **User 194 put function**

```
public void put(Key key, Value val)
{
    if (val == null) { delete(key); return; }
    root = put(root, key, val);
}

private Node put(Node x, Key key, Value val)
{
    if (x == null) return new Node(key, val, 1);
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) x.left  = put(x.left,  key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else              x.val   = val;
    x.N = 1 + size(x.left) + size(x.right);
    return x;
```

Listing 4.6: **User 136 put function**

```
public void put(Key key, Value val) {
    if (val == null) { delete(key); return; }
    root = put(root, key, val);
}
```

```
private Node put(Node x, Key key, Value val) {
    if (x == null) return new Node(key, val, 1);
    int cmp = key.compareTo(x.key);
    if      (cmp < 0) x.left  = put(x.left,  key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else              x.val   = val;
    x.N = 1 + size(x.left) + size(x.right);
    return x;
}
```

Both users have the same put method, using the same variable names and their of conditional statements is the same also. From this we can observe that the runtime score could potentially indicate a signal that there is similarities in the code but in order to be certain, it is important to check two user's that whose runtime score difference was large. The pair of user's with second closest runtime score are user 174 and user 27. For this pair of submission, user 174 has slightly different formatting then user 27 but that is only through how the code is formatted it. The conditional statements still operate in the same way, this could potentially fool a plagiarism tool but the runtime score is not impacted by moving statements. See appendices .2 and .3 for their code.

The submissions from the pair user 64 and user 145 share the third closest runtime scores. The interesting part about these users is that they both have a single submission, whereas the other two closest pairs have more than submission of the binary search tree code. The pair user 64 and user 145, the put, get and contains function are the exact same and considering they both have the same code for those functions it's understandable why their runtime scores are similar.

4.1.3 Comparison between runtime scores and Compare50 similarity scores

The next part of the evaluation was to compare the 25 closest pairs of submissions based on their runtime score to compare50's 25 closest pairs based on their similarity scores. From observing both, there was no direct correlation between compare50's closest pairs and the 25 closest pairs of runtime scores. Compare50's closest pairs were user 60 and user 61's submission, the runtime score obtained from the experiments shows the pair had a difference of 0.00234793988 seconds. Compare50 scored this with a similarity score of ten out of ten. When looking at the lowest similarity scores according to which is 0.9 out of 10, these were users 174 and 188, users 12 and 174, and finally users 171 and 147. Looking at the runtime score difference between these pairs for user 174 and user 188 it is

0.00784664077 seconds. For user 174 and user 12 it is 0.00829447651 seconds and for user 171 and 147 it is 0.02784342206 seconds. From this, it is clear that the runtime scores differ slightly for pair programs that had a lower comparison score.

4.2 Discussion

The comparison of runtime scores and traditional similarity scores generated by tools like Compare50 provides valuable insights into the efficacy and limitations of different plagiarism detection methodologies. This section discusses the implications of the findings, explores potential explanations for observed discrepancies, and suggests avenues for future research in the field of code similarity detection.

Discrepancies between Runtime Scores and Compare50 Similarity Scores

The lack of direct correlation between runtime scores and Compare50 similarity scores raises important questions about the underlying factors influencing code similarity detection. While Compare50 relies on static analysis techniques to identify syntactic similarities between code fragments, runtime-based approaches offer a more dynamic perspective, capturing nuances in program behavior that may not be evident from static code analysis alone. The observed discrepancies suggest that traditional static analysis methods may overlook certain types of code similarities that become apparent when analyzing program execution.

Factors Influencing Runtime Scores

Several factors may contribute to variations in runtime scores across different code submissions. Firstly, the efficiency and optimization of code implementation can significantly impact runtime performance. Submissions with well-optimized algorithms and data structures are likely to exhibit lower runtime scores compared to less optimized counterparts. Additionally, the choice of programming language, compiler optimizations, and hardware architecture may influence runtime behavior, leading to variations in observed runtime scores.

Potential Limitations of Compare50

The findings suggest that traditional static analysis tools like Compare50 may have limitations in accurately detecting certain types of code similarities. Static analysis techniques typically focus on syntactic features such as variable names, function calls, and control structures, overlooking deeper structural similarities and algorithmic approaches. As a

result, Compare50 may fail to identify similarities present at the algorithmic or semantic level, leading to discrepancies between runtime scores and similarity scores.

Implications for Plagiarism Detection

The discrepancies between runtime scores and Compare50 similarity scores underscore the importance of adopting a multi-faceted approach to plagiarism detection. While static analysis tools like Compare50 play a valuable role in identifying surface-level similarities, runtime-based approaches offer complementary insights into program behavior and execution dynamics. Integrating both static and dynamic analysis techniques into plagiarism detection frameworks can enhance the accuracy and robustness of code similarity detection, enabling educators to identify instances of plagiarism more effectively.

Findings

The comparison between runtime scores and Compare50 similarity scores in this study yields significant insights into the effectiveness of various plagiarism detection methodologies. While conventional static analysis tools like Compare50 serve as valuable initial screening tools for identifying surface-level code similarities, the incorporation of runtime-based approaches provides a complementary perspective by delving into program behavior and execution dynamics. This research underscores the importance of integrating both static and dynamic analysis techniques to enhance the robustness and effectiveness of plagiarism detection frameworks, particularly in academic settings. The findings suggest that a combined approach offers a more comprehensive understanding of code similarity and improves the accuracy of detecting instances of plagiarism.

Chapter 5

Limitations and Future Works

5.1 Limitations

While this study sheds light on whether or not runtime information can signal similarities between programs, it's essential to acknowledge its limitations. The research focused primarily on a specific set of programming tasks and may not capture the full spectrum of code similarity scenarios encountered in real-world academic settings. Additionally, the evaluation was mostly based on runtime-based scores and Compare50 similarity scores, neglecting the exploration of other plagiarism detection methodologies such as JPlag which makes use of a different algorithm. This narrow focus may limit the generalizability of the findings and overlook alternative approaches that could offer complementary insights. Furthermore, the study's sample size and diversity of programming tasks may not be sufficient to draw definitive conclusions about the efficacy of different detection methods across various contexts. Lastly, the evaluation may not fully account for the complexities of detecting plagiarism in programming assignments, such as the potential for obfuscation techniques as discussed in Devore-McDonald and Berger (2020). These limitations highlight the need for further research to explore a broader range of detection methods, consider diverse programming tasks, and address the complexities inherent in plagiarism detection in coding assignments.

5.2 Future Works

Future work endeavors on this research question should continue to explore the central research question of whether runtime information can effectively signal similarities between programs. Building upon the findings of this study, there are several avenues for further investigation that can advance our understanding of this topic.

1. One promising direction for future research is to conduct more extensive empirical studies to validate and refine the findings of this research. This could involve expanding the dataset to include a wider range of programming tasks, languages, and contexts, thereby enhancing the robustness of the conclusions drawn. By analyzing a larger and more diverse set of programming assignments, researchers can gain deeper insights into the relationship between runtime behavior and code similarities, identifying patterns and heuristics that are applicable across different scenarios.

2. Exploring novel methodologies and techniques for leveraging runtime information to detect similarities between programs. This could include developing advanced algorithms and machine learning models that can analyze runtime data more effectively, identifying subtle patterns and correlations that may not be apparent through traditional static analysis techniques alone. By making use of the power of machine learning and data analytics, researchers can develop more sophisticated and accurate plagiarism detection tools that leverage runtime information to identify suspicious code similarities.

3. Furthermore, there is a need to investigate the practical implications of incorporating runtime-based approaches into existing plagiarism detection frameworks. This could involve conducting usability studies and user evaluations to assess the feasibility and effectiveness of these approaches in secondary schools and universities. By partnering with instructors and students to deploy and evaluate runtime-based detection tools in programming courses, researchers can gather valuable feedback on the usability, acceptance, and impact of these tools on academic integrity practices.

4. Additionally, future research should explore the potential limitations and challenges associated with using runtime information for plagiarism detection. This includes investigating factors such as code variability, optimization techniques, and the impact of external factors on runtime behavior. By conducting thorough sensitivity analyses and benchmarking experiments, researchers can identify the boundaries and constraints of runtime-based detection methods, helping to guide the development of more robust and reliable detection strategies.

5. Moreover, in the future there may be a need to address the ethical implications and privacy concerns associated with the use of runtime-based plagiarism detection tools. This includes ensuring that these tools adhere to principles of data privacy, transparency, and fairness, and that they do not infringe upon students' rights or compromise their academic integrity. By incorporating ethical considerations into the

design and implementation of runtime-based detection systems, researchers can promote responsible and equitable use of these technologies in educational settings.

6. Furthermore, future research should explore the potential synergies between runtime-based detection methods and other approaches to plagiarism detection, such as static analysis, manual inspection, and peer review. By integrating multiple detection techniques into a unified framework, researchers can develop more comprehensive and reliable plagiarism detection systems that leverage the strengths of each approach while mitigating their individual limitations.

In conclusion, future research in the field of code similarity detection should continue to explore the potential of runtime information as a signal for identifying similarities between programs. By advancing our understanding of the relationship between runtime behavior and code similarities, researchers can develop more effective and reliable plagiarism detection tools that enhance academic integrity and promote fairness in educational evaluation practices.

Chapter 6

Conclusions

This paper has explored the potential of leveraging runtime information to detect similarities between programs, aiming to redefine the applications of existing similarity detection tools. Through a comprehensive investigation, we have gained valuable insights into the efficacy and limitations of different plagiarism detection methodologies.

The comparison between runtime scores and traditional similarity scores obtained from tools like Compare50 revealed intriguing discrepancies, suggesting that traditional static analysis tools may overlook certain types of code similarities. While static analysis techniques focus on syntactic features, runtime-based approaches offer a more dynamic perspective, capturing nuances in program behavior that may not be evident from static code analysis alone.

Factors such as code optimization, programming language choice, and hardware architecture influence runtime behavior, leading to variations in observed runtime scores across different code submissions. Despite these variations, by integrating runtime-based approaches into plagiarism detection frameworks there could be an enhancement of the accuracy and robustness of code similarity detection, complementing the capabilities of traditional static analysis tools.

Our findings underscore the importance of adopting a multi-faceted approach to plagiarism detection in programming assignments. By integrating both static and dynamic analysis techniques, educators and researchers can develop more effective and reliable plagiarism detection tools, thereby upholding academic integrity and promoting fairness in educational evaluation practices.

Looking ahead, future research endeavors should focus on validating and refining the findings of this study through more extensive empirical studies. Exploring novel methodologies for leveraging runtime information, conducting usability studies, addressing ethical implications, and exploring synergies between different detection techniques are all

promising avenues for further investigation.

The aim of this research was to uncover whether the runtime information could signal similarities in programs, it is clear from this research that this is in fact a possibility. With the appearance of generative language models, the ability to create and modify code has become simpler and with that a goal has appeared: To shift from traditional approaches in detecting plagiarism and uncover new techniques to create a more effective plagiarism tool.

In conclusion, this dissertation contributes to the ongoing discourse on plagiarism detection in programming assignments by highlighting the potential of runtime-based approaches. By advancing our understanding of the relationship between runtime behavior and code similarities, we pave the way for the development of more effective and reliable plagiarism detection tools, thereby enhancing academic integrity and promoting fairness in educational evaluation practices.

Bibliography

Ahadi, A. and Mathieson, L. (2019). A comparison of three popular source code similarity tools for detecting student plagiarism. In *Proceedings of the Twenty-First Australasian Computing Education Conference*, ACE '19, page 112–117, New York, NY, USA. Association for Computing Machinery.

Devore-McDonald, B. and Berger, E. D. (2020). Mossad: defeating software plagiarism detection. *Proc. ACM Program. Lang.*, 4(OOPSLA).

Doi, M. (2023). Regression verification via fuzz testing. Technical report, Trinity College Dublin.

Gipp, B. and Meuschke, N. (2011). Citation pattern matching algorithms for citation-based plagiarism detection: greedy citation tiling, citation chunking and longest common citation sequence. In *Proceedings of the 11th ACM Symposium on Document Engineering*, DocEng '11, page 249–258, New York, NY, USA. Association for Computing Machinery.

Hasan, S., King, Z., Hafiz, M., Sayagh, M., Adams, B., and Hindle, A. (2016). Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 225–236, New York, NY, USA. Association for Computing Machinery.

Jhi, Y.-C., Jia, X., Wang, X., Zhu, S., Liu, P., and Wu, D. (2015). Program characterization using runtime values and its application to software plagiarism detection. *IEEE Transactions on Software Engineering*, 41:1–1.

Kakarontzas, G. and Pardalidou, C. (2018). Improving component coupling information with dynamic profiling. In *Proceedings of the 22nd Pan-Hellenic Conference on Informatics*, PCI '18, page 156–161, New York, NY, USA. Association for Computing Machinery.

Malan, D. J., Sharp, C., van Assema, J., Yu, B., and Zidane, K. (2021). Cs50's github-based tools for teaching and learning. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*, SIGCSE '21, page 1354, New York, NY, USA. Association for Computing Machinery.

Novak, M., Joy, M., and Kermek, D. (2019). Source-code similarity detection and detection tools used in academia: A systematic review. *ACM Trans. Comput. Educ.*, 19(3).

Omar, K., Esmaeel, N., and Ebrahim, Z. (2024). Review on plagiarism detection systems, algorithms, weakness points. *International Journal of Intelligent Systems and Applications in Engineering*, 12(18s):693–699.

Padhye, R., Lemieux, C., Sen, K., Papadakis, M., and Le Traon, Y. (2019). Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, page 329–340, New York, NY, USA. Association for Computing Machinery.

Prechelt, L. and Malpohl, G. (2003). Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, 8.

Schleimer, S., Wilkerson, D. S., and Aiken, A. (2003). Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, page 76–85, New York, NY, USA. Association for Computing Machinery.

Tuli, R. (2016). Ruchi tuli, "plagiarism: A threat to intellectual property rights" international journal of recent research aspects issn: 2349-7688, vol. 3, issue 1, march 2016,pp. 27-32. *International Journal of Recent Research Aspects*, 3:27–32.

Yin, F., Dong, D., Li, S., Guo, J., and Chow, K. (2018). Java performance troubleshooting and optimization at alibaba. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, page 11–12, New York, NY, USA. Association for Computing Machinery.

Appendix

.1 Test Class

Listing 1: **Test Class**

```
@RunWith(JQF.class)
public class TestBST {

    @Fuzz
    public void BstTest(@From(ArrayGenerator.class) Integer[] numbers)
    {
        // Specify the path to your file
        String filePath = ".../current_file.txt";
        String line = "";
        try (BufferedReader br =
        new BufferedReader(new FileReader(filePath))) {
            line = br.readLine();
        } catch (IOException e) {
            e.printStackTrace(); // Handle any IO exceptions
        }
        BST<Integer, Integer> bst = new BST<>();
        HashMap<Integer, Double> runTimes = new HashMap<>();
        long startTime = 0;
        long endTime = 0;
        double elapsedTime = 0;
        for (int i = 0; i + 3 < numbers.length; i = i + 3) {
            switch (numbers[i] % 3) {
                case 0:
                    startTime = System.nanoTime();
                    // Add a try/catch here
```

```
            try {
                boolean containsOutput
                = bst.contains(numbers[i]);
            } catch (Exception e) {
            }
            endTime = System.nanoTime();
            elapsedTime += ((endTime - startTime) /
            1_000_000_000.0);
            runTimes.put(0, elapsedTime);
            break;
        case 1:
            startTime = System.nanoTime();
            // Add a try/catch here
            try {
                Integer getOutput = bst.get(numbers[i]);
            } catch (Exception e) {
            }
            endTime = System.nanoTime();
            elapsedTime += ((endTime - startTime) /
            1_000_000_000.0);
            runTimes.put(1, elapsedTime);
            break;
        case 2:
            startTime = System.nanoTime();
            // Add a try/catch here
            try {
                bst.put(numbers[i], numbers[i]);
            } catch (Exception e) {
            }
            endTime = System.nanoTime();
            elapsedTime += ((endTime - startTime) /
            1_000_000_000.0);
            runTimes.put(2, elapsedTime);
            break;
        default:
            break;
    }
```

```
        }
```

.2 User 174 BST class

Listing 2: **User 174's Binary Search Tree class**

```
public boolean contains(Key key)
    {
        if (get(key) != null)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public Value get(Key key)
    {
        return get(root, key);
    }

    private Value get(Node x, Key key)
    {
        if (x == null)
        {
            return null;
        }

        int cmp = key.compareTo(x.key);

        if (cmp < 0)
        {
            return get(x.left, key);
        }
        else if (cmp > 0)
```

```
        {
            return get(x.right, key);
        }
        else
        {
            return x.val;
        }
    }

    /**
     *  Insert key-value pair into BST.
     *  If key already exists, update with new value.
     *
     *  @param key the key to insert
     *  @param val the value associated with key
     */
    public void put(Key key, Value val)
    {
        if (val == null)
        {
            delete(key);
        }
        else
        {
            root = put(root, key, val);
        }
    }

    private Node put(Node x, Key key, Value val)
    {
        if (x == null)
        {
            return new Node(key, val, 1);
        }

        int cmp = key.compareTo(x.key);
```

```
        if (cmp < 0)
        {
            x.left = put(x.left,  key, val);
        }
        else if (cmp > 0)
        {
            x.right = put(x.right, key, val);
        }
        else
        {
            x.val = val;
        }

        x.N = 1 + size(x.left) + size(x.right);

        return x;
    }
```

.3 User 27 BST class

Listing 3: **User 27's Binary Search Tree class**

```
public boolean contains(Key key) {
        return get(key) != null;
    }
    public Value get(Key key) { return get(root, key); }

    private Value get(Node x, Key key) {
        if (x == null) return null;
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) return get(x.left, key);
        else if (cmp > 0) return get(x.right, key);
        else              return x.val;
    }

    /**
     *  Insert key-value pair into BST.
```

```
     *  If key already exists, update with new value.
     *
     *  @param key the key to insert
     *  @param val the value associated with key
     */
    public void put(Key key, Value val) {
        if (val == null) { delete(key); return; }
        root = put(root, key, val);
    }

    private Node put(Node x, Key key, Value val) {
        if (x == null) return new Node(key, val, 1);
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) x.left  = put(x.left,  key, val);
        else if (cmp > 0) x.right = put(x.right, key, val);
        else              x.val   = val;
        x.N = 1 + size(x.left) + size(x.right);
        return x;
    }
```

.4 Pom.xml file

Listing 4: **POM file for Maven**

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ie.tcd.makanjui</groupId>
  <artifactId>fuzz</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>fuzz</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
```

```
      <version>4.13.1</version>
    </dependency>
    <dependency>
      <groupId>com.pholser</groupId>
      <artifactId>junit-quickcheck-core</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>com.pholser</groupId>
      <artifactId>junit-quickcheck-generators</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>edu.berkeley.cs.jqf</groupId>
      <artifactId>jqf-fuzz</artifactId>
      <version>2.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.commons</groupId>
      <artifactId>commons-csv</artifactId>
      <version>1.10.0</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>edu.berkeley.cs.jqf</groupId>
        <artifactId>jqf-maven-plugin</artifactId>
        <version>2.0</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.0.0</version>
      </plugin>
    </plugins>
```

```
    </build>

</project>
```