Detecting Academic Dishonesty in Computer Science Courses

A Major Qualifying Project
Submitted to the faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
Degree in Bachelor of Science
in
Computer Science

By:
Matthew Olson, Matthew Hurlbut-Coke, and Marcelino Puente-Perez

Date: 4/28/2022
Project advisor:

_____

Professor Joseph Beck

Abstract

Plagiarism Detector (PD) is a software that was created in response to an alarming incident of academic dishonesty in an entry-level computer science course at Worcester Polytechnic Institute (WPI). The software developed in this project is meant to detect similarities in code and present the results to instructors in a meaningful way. PD was built in Java using JavaFX with the intention of checking assignments in the Racket programming language. Racket is a general purpose programming language that was selected for the introductory CS classes at WPI due to its intuitive function templates and high-level features. PD used a syntax-tree based similarity detector that was custom built for Racket's unique syntax structure. We found that our algorithm worked respectably well at finding cases of possible cheating.

Acknowledgments

Special thanks to Professor Joseph Beck for recruiting us to this project and supporting us along the way.

Special thanks to the WPI Computer Science Department.

Special thanks to Reilly Norum and Jake Casey for helping edit and revise this paper.

Special thanks to Professor Hugh Lauer and his team of students for paving the way for this type of software at WPI with Checksims.

Table of Contents

Abstract **1**

Acknowledgments **2**

Table of Contents **3**

1. Introduction **4**

2. Literature Review **5**

2.1 What is Academic Dishonesty 5

2.2 Previous Solutions 6

2.2.1 Moss 6

2.2.2 Checksims 7

2.3 Racket Specific Approaches 8

2.3.1 Syntax Trees 8

2.3.2 Racket Syntax 9

2.3.3 Tree Comparison Algorithms 9

3. Requirements **10**

4 Solution **11**

4.1 Application 11

4.2 Racket Tree 12

4.3 Data Visualization 14

5. Results **16**

6. Future Work and Conclusions **18**

6.1 Future Work 18

6.2 Conclusions 19

References **20**

Additional Materials **22**

1. Introduction

The COVID-19 pandemic has brought many stressors to students and professors alike. Universities all over the world have since reported a significant increase in academic dishonesty as classes have transitioned to an online learning format (Herdian et al. 2021). However, online learning is not a new concept. A 2008 study taken at a Northeastern University of 12,000 students found that between 50-70% of undergraduates had performed some form of academic dishonesty (Schmelkin et al. 2008). Additionally, it appears that stress and fear of loss due to the COVID-19 pandemic are tightly connected to the increase in academic dishonesty (Grolleau 2016).

Professor Beck, the advising professor for this project, experienced an extreme case of cheating in his Spring 2021 Computer Science classes with 19 student admissions of dishonesty and 2 cases referred to the Campus Hearing Board. This incident was prompted by a Student Assistant noticing a clear case of copying on an assignment. That prompted Professor Beck to manually evaluate all 63 student submissions for evidence of cheating. This process was laborious and not one he wanted to repeat. In recent years, record numbers of CS majors have been earning degrees; from 2017 to 2018 there was a 15% increase (Partovi 2020). As WPI CS grows similarly, it has become increasingly more difficult for teaching assistants and professors to detect cheating in the multiple assignments per class. Our team has worked hard to create a program that is accurate and efficient at detecting academic dishonesty in these Racket-based computer science courses. With our easy to use program, professors can review possible cases of cheating in a much faster period of time.

2. Literature Review

2.1 What is Academic Dishonesty

Academic dishonesty is commonly defined as the act of misrepresenting the work of a student, such as through cheating on tests or plagiarizing the work of others. The definitions slightly vary from school to school but WPI separates its definition into four categories: Cheating, Fabrication, Facilitation, and Plagiarism. Cheating can be described as using or attempting to use materials, information, or study aid that are unauthorized for use in an academic exercise. For example, copying a peer's work or communicating with a peer during an exam would be considered cheating. Fabrication is the falsification or misrepresentation of information in an academic exercise such as altering grades or changing exam solutions after the allotted time. Facilitation involves helping someone commit academic dishonesty by sharing answers or doing work for a peer. Lastly, Plagiarism can be defined as using words, ideas, data, or code without properly attributing it to the original author. This can be seen as paraphrasing without citation or taking credit for someone else's work (*What is Academic Dishonesty* 2022). The academic dishonesty policy for WPI can be accessed through the link below:

https://www.wpi.edu/about/policies/academic-integrity/dishonesty

As mentioned earlier, some professors have seen a surge in the rates of academically dishonest behavior during the COVID-19 pandemic (Beck, personal communication), such as blatantly copied assignment submissions with only superficial changes made to them. Our project is designed to help combat dishonest behavior by making it easier for professors to compare submitted code, and draw attention to potential plagiarism/copying.

2.2 Previous Solutions

The problem of academic dishonesty is not a new one and has had many attempts trying to detect it. This problem is especially difficult to deal with in introductory computer science courses, such as Introduction to Program Design (CS1101) and Accelerated Introduction to Program Design (CS1102). These courses in particular use Racket, a language that is not commonly used, and as such, there are not many dedicated tools for detecting plagiarized work. Most schools keep their methods secret to avoid students circumventing the tool, but the benchmark for detecting plagiarized code is the widely used tool known as Moss (Measure of Software Similarity).

2.2.1 Moss

Moss is a free to use software and currently maintained and provided by Stanford University. While the ideas behind its method are publicly available, the source code is closed source. Originally developed in 1994 (Aiken, n.d), Moss can be used for a number of languages, such as Java, Matlab, and most relevant to our project: Scheme, which Racket is based on. Moss is a web-based application where registered users can upload a set of files to a server, which will return a report displaying the similarity between each pair of files. It also has the ability to ignore predetermined sections of code, such as libraries or starter code, so that it will not erroneously flag code that was provided by the instructor.

As a copy-detection algorithm, Moss has three key features that help it detect similarity across files: whitespace insensitivity, noise suppression, and position independence (Yang, D, 2019). Being whitespace insensitive means that the algorithm does not take whitespace, such as spaces or new lines, into account when calculating similarity. Noise suppression refers to the ability of the algorithm to ignore small, meaningless matches, such as ones consisting of only a

few words. The last feature is position independence, which means that even if copied code is rearranged and restructured, the program will still flag it as being similar.

Despite Moss's complexity and power, however, it is not without its faults. One of its biggest weaknesses is its inability to detect syntactic changes, such as replacing if-else statements with switch-case statements, or for loops with while loops. Another drawback is that while Moss can detect similarity, there is no way for it to detect causation. It can indicate which two files were likely copied, but it cannot conclusively determine that plagiarism took place; it can find evidence, but not definitive proof. Ultimately, whether or not academic dishonesty has taken place comes down to one's own judgment.

2.2.2 Checksims

Academic dishonesty is not a new problem at WPI. In 2015, Professor Hugh Lauer created an MQP for this same problem. This MQP would end up creating the program Checksims to find academic dishonesty. Checksims was designed to be a multi-use application that was language agnostic and able to compare students' submissions on a line by line basis. The software is a local app that a professor could download and run on their submissions, outputting a similarity value from 0 to 1 to determine how much code from one file could be found in another.

Checksims calculates its comparisons by using one of two algorithms, either a simple line comparison algorithm or the more complex, Smith-Waterman algorithm. The line comparison is designed to be a fast technique that uses n-gram fingerprinting on a line by line basis to get the number of similar lines. While the algorithm is very quick and scales linearly with file size, a small change per line is enough to mess with the outputs. The other algorithm used in Checksims

is Smith-Waterman. While this algorithm is very tamper resistant, it has a time complexity of $O(mn)$ on two files with m and n fingerprints (Heon and Marvill, 2015).

While Checksims is designed to be language agnostic, it is only designed to work on text files. This means that it does not work on all Racket files as some of them are stored in Racket's proprietary file format.

2.3 Racket Specific Approaches

Previous approaches of academic dishonesty software are language agnostic. While these are very useful for large computer science departments, they sacrifice potential improvements that can be made using the specifics of certain languages. They are also typically based on reading plaintext files, which causes an issue when working with Racket. Racket supports a proprietary file format called WXME that is closed source with no documentation (*Editors*, n.d). This makes working with Racket specifically complicated as it cannot be plugged into existing solutions.

2.3.1 Syntax Trees

Languages are defined by their syntax and how that syntax is translated to actual machine code. Most languages process the given code using what is called a syntax tree. This is a way of translating human readable source code into a machine usable data structure. Syntax trees are useful for plagiarism detection because many simple syntactic changes (e.g. variable names, whitespace, comments, etc.) are not expressed in the final tree. This allows for relatively simple methods of disguising academic dishonesty to be ignored.

2.3.2 Racket Syntax

Syntax trees are helpful specifically for Racket because of its simple syntax structure. Racket's syntax uses s-expressions to represent the underlying syntax tree of the document (*reader*, n.d). This makes the translation to a syntax tree extremely easy as parsing the s-expressions will lead to a simple tree data structure.

2.3.3 Tree Comparison Algorithms

One of the most convenient ways to compare two files after they have been parsed into syntax trees are tree comparison algorithms. These algorithms are designed to, when given two trees, produce a similarity value between them. The most common type is tree edit distance (TED). TED is a family of algorithms that describe the difference between two trees as the weighted sum of the number of edits needed to make them the same. The edits are commonly described as either the addition, removal, or renaming of a tree node and are each given a weight (Paaßen, 2018). The algorithms output the lowest sum of changes needed to transform the trees. While a lot of work has been done in optimizing these algorithms, the current best known asymptotic bound is that of $O\left(n^3\right)$. This runtime is not acceptable for the real-time program we were asked to develop as our trees are on the order of 1000 leaves. While we can not use TED in our program it could be a possible avenue in a less time dependant scenario or with a lot of optimization as real-world cases can get the experimental evaluations down to $O(n\log(n))$ in specific scenarios (Schwarz, Pawlik, & Augsten, 2017).

3. Requirements

The project's requirements were laid out by Professor Beck at the beginning of the MQP. The requirements call for the creation of a program that can fulfill them in a functional and easy to use manner. The project requirements are as followed:

1. The program must be easy enough to be used by any CS professor or staff.
2. The program must be run locally in order to assure student confidentiality and program accuracy.
3. The output of the program must give a straightforward explanation of what cases may or may not be academic dishonesty to later be verified by the course staff.
4. The program must be easily configurable through a set of variables and switches.
5. The program must output a side by side comparison of code to allow for quick analysis.
6. The program must implement Checksims[1] as an alternative method of similarity detection.

Due to the unique syntax of Racket and the time constraints of an MQP, our team decided to go with a plaintext-based program. We wanted a solution that could be run locally on any client with the program downloaded. With the resources and documentation our project will provide, we hope that our project may be easily modified and improved in the future. Our requirements call for a security by obscurity approach as the program must only be used by those authorized to view student grades and information. However, the similarity detection algorithm can easily be migrated to a host-based online format. This is a format that may be explored by future projects but it would require a different set of security layers to remain secure.

[1] Please refer to section 2.2.2 for more information on Checksims

4 Solution

4.1 Application

An important feature of our project was a finished standalone app that users could submit assignments into. We built this app using JavaFX with some Racket to help support translating Racket files. A key focus of this app was simplicity; we wanted users to be able to open up the app and know exactly what to do. This was accomplished by having the main function of the app be the center focus, with other options being in a hamburger menu to the side (Figure 1). In the submission section of the app, we would let the user upload a directory of files that the algorithm would run through, producing a value of every pair of submissions in the project. The application would then provide the user with a list of pairs, ordered by the similarity value (higher value means high similarity).

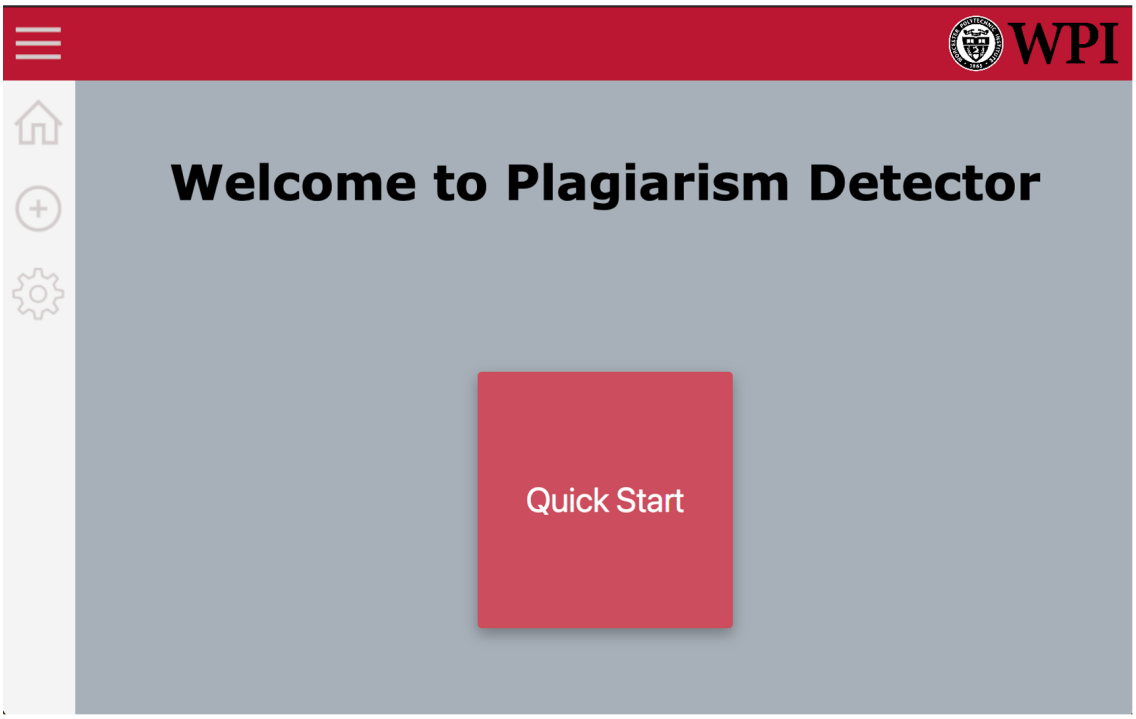

Figure 1: The opening screen of the application

Another goal of the app was to keep the usage fully contained. We wanted the user to be able to make judgments without having to open the code in another app. To do this, we supplied the user with a side by side view of any pair of submissions so they can manually compare the two files and make their own informed decision (Figure 2).

While a major goal was to make the app simple, we also wanted the app to be flexible in case the default results were not desired. To do this we implemented a settings page with many

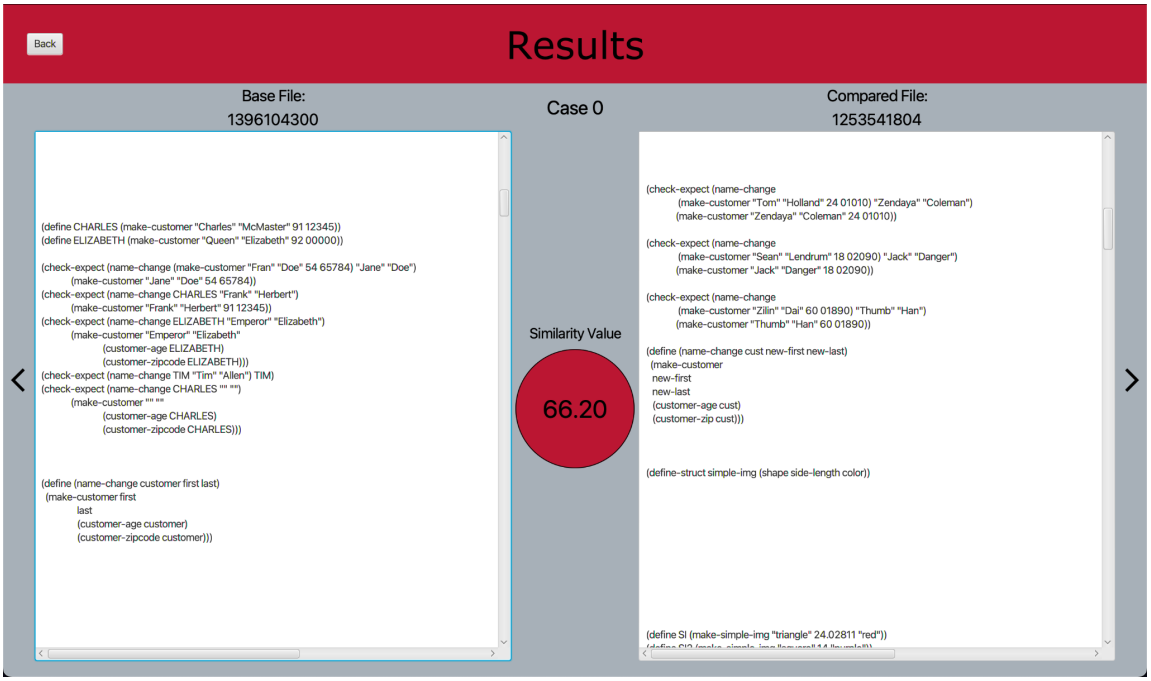

Figure 2: Comparing two files in the application

controls that can be accessed while the user uploads their datasets. These options include things like number of threads, algorithm controls, and also a granularity control for our proprietary algorithm.

4.2 Racket Tree

A goal for our project was to take advantage of our unique scenario to try and produce an algorithm with better results on Racket files over existing solutions. The solution we came up

with for this was to use a syntax tree of Racket and a tree comparison algorithm. While existing tree comparison algorithms, like that of TED, were considered, we found that the large size of the tree and our specific use case made these algorithms too slow. Instead, we tried to come up with a unique solution based around the number of similar leaves in two syntax trees of Racket files.

Algorithm 1 Find the similarity value between two leaf nodes, x and y

```
function SimilarityValue(Two leaves x, y)
    if x ≠ y then
        return 0
    end if
    xparent ← Parent(x)
    yparent ← Parent(y)
    total ← 0
    while xparent ≠ null and yparent ≠ null do
        count ← SizeOfMatchingChildren(xparent, yparent, x, y)
        if count = 0 then
            break
        end if
        total ← total + count
        x ← xparent
        y ← yparent
        xparent ← Parent(x)
        yparent ← Parent(y)
    end while
    return total
end function
function SizeOfMatchingChildren(Parentx, Parenty, x, y)
    sum ← 0
    for child ∈ Parentx ∩ Parenty do
        if child ≠ x and child ≠ y then
            sum ← sum + Size(child)
        end if
    end for
    return sum
end function
```

Figure 3: Pseudocode for the similarity value between two leaves

This algorithm would start by hashing each tree's leaves into a hashmap. Then, between two trees, for each identical leaf in both trees it would traverse up the tree counting the number of identical trees rooted at each parent node. The algorithm will continue until it finds a root

node with no matching children, other than the path already being traversed, where it would terminate. Pseudocode for this algorithm can be found in Figure 3. The algorithm then averages out the values for each leaf and sums the total values, before normalizing by the number of leaves in the tree.

4.3 Data Visualization

One goal we wanted to incorporate into our project was an easy and concise way of seeing a visual representation of the data from the comparison. Inspired by a visualization charting character interactions in *Les Misérables,* made by Mike Bostock, we originally created a visualization in HTML/Javascript (Bostock 2012). This visualization takes a set of comparisons and generates a heatmap showing levels of similarity between files (Figure 4).

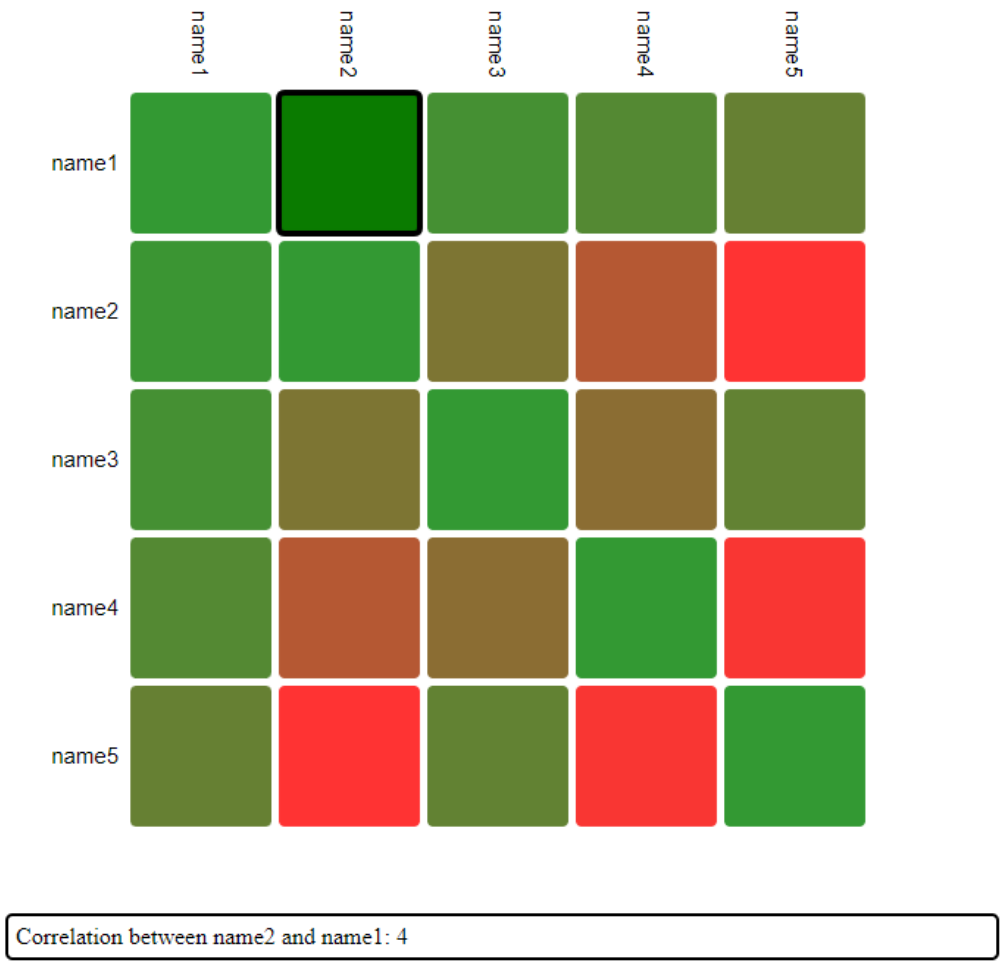

Figure 4: Visualization with sample data

This heatmap visualization went through a few iterations, with tweaks to increase its efficacy, such as having the colors scale off of the minimum and maximum values from the comparisons, or the ability to filter out results within a certain range. Ultimately, incorporating an HTML file into our program proved difficult due to limitations of the JavaFX webviewer, and so we attempted to recreate it in JavaFX, using a custom charts library (HanSolo 2022). Technical difficulties with IntelliJ, however, prevented the visualization page from being finished.

5. Results

To be able to evaluate our application, we were given access to an anonymized data set containing previous years assignments. These assignments were stripped of identifying information such as filename and comments. We tested these files against Checksims and our own algorithms to find any evidence of academic dishonesty. While we found that the amount of academic dishonesty was rather light in this test suite[2], we were able to find a couple examples most specifically in Assignment 3 and Assignment 4.

If we plot the outputs comparing both of these algorithms on box and whisker plots we can see that for both algorithms academically dishonest work is rated vastly above average, and is usually a massive outlier compared to the rest of the work. Both of these algorithms agreed on what was academically dishonest in Assignment 4, while both struggled for certain submissions in Assignment 3. We think this is because Assignment 3 had a larger body of starter code that affected both data sets. Because our algorithm does not have a filter for starter code, it affected our results. Despite Checksims having built in starter code handling, we were unable to provide the starter code files for this assignment dataset.

Along with our sample data set we were also able to get some anecdotal evidence by providing the application to our advisor, Professor Beck. Our advisor was able to run it on a real world data set with confirmed academic dishonesty. In this data set, our algorithm outperformed Checksims by correctly identifying six cases within its proposed top ten, compared to Checksims' three.

[2] The most likely explanation for this is our use of data from CS 1102, which is the more advanced of the two courses. We think this causes a self selecting behavior where academically dishonest students would not take the more advanced class. This is supported by Professor Beck's experience on course feedback forms where none of the students in 1102 claimed to be aware of any academic dishonesty.

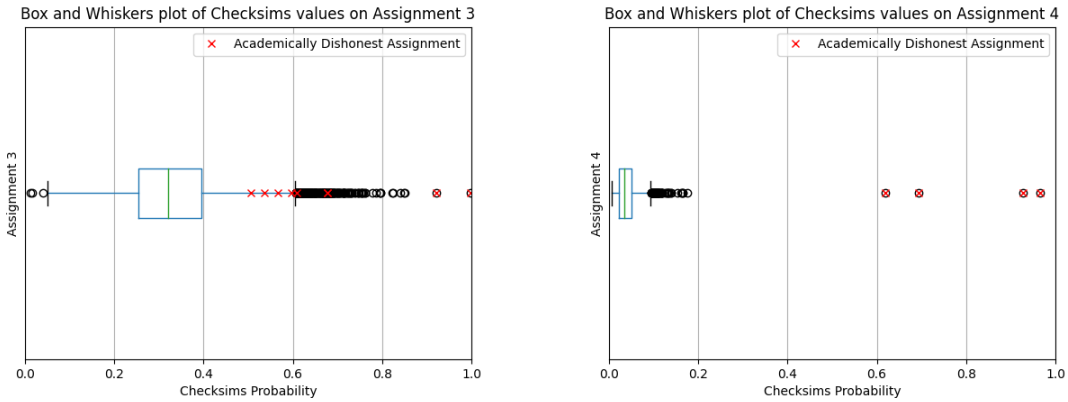

Figure 5: Checksims output for Assignments 3 and 4

We also tried to compare our algorithm against Moss. While Moss is the gold standard of the field, it being run on a remote server where the user has little control can hamper its useability. For example, we wanted to apply Moss on our Assignment 3 and 4 data sets but we were not able to get a usable result, and since the only output from the program was a broken HTML results page, we were not able to troubleshoot the problem and gather usable results to compare.

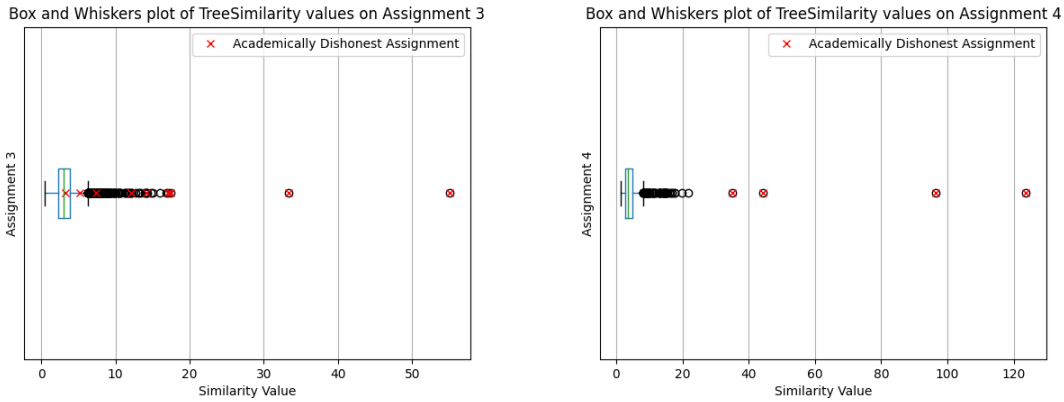

Figure 6: Tree Similarity output for Assignments 3 and 4

6. Future Work and Conclusions

6.1 Future Work

Plagiarism Detector was developed keeping future expandability in mind. We provide a simple JavaFX platform that can be easily extended with a simple code structure. Following are some improvements and pivot points we believe would greatly benefit the future development of a program like ours at WPI.

First, an expansion of the similarity scoring system would maximize the user's ability to recognize cheating. For example, implementing the ability to remove provided template code would further refine similarity value results. With strictly student-generated code, we would be able to better characterize cheating methods and identify them much faster.

Another method of improving the similarity scoring system is to implement a machine learning (ML) model. The model would be capable of analyzing a large series of results and provide an alternate plagiarism metric based on additional characteristics that our plain-text detector can not take into consideration. In order for this feature to be meaningful, a considerably large dataset would be required to train a good ML model. One way to develop our own model would be by giving the professor the option to characterize "cheating" or "not cheating" from within the program itself. Adding these characterizations can greatly benefit the model. As time goes on and the dataset of plagiarism grows, the ML model is able to further identify cheating.

A powerful feature addition to this program would be the ability to fuse two or more similarity detection algorithms into one output. In its current state, our platform only allows for the use of Checksims *or* Tree Similarity. The fusion of algorithms could result in a more accurate analysis of cheating, allowing us to provide more confident similarity values.

Finally, a general interface overhaul would greatly benefit Plagiarism Detector. While the purpose of our program was to be quick and easy to use, appearance was not our primary consideration. Some recommendations would be: improved and standardized settings page, an interactive visualization of the checked assignments, better transitions, and scaling of user interface elements for different window sizes and resolutions.

6.2 Conclusions

Our work on the Plagiarism Detector has helped us gather a better understanding of the solution space around detecting academic dishonesty. Working first–hand with the creation of our Racket Syntax-Tree algorithm has shed light on the limitations and problems that arise when detecting cheating in code. Limited code databases, renaming of functions and variables, and alternate valid code structures are just some of the issues all detectors struggle to combat.

Despite this, the Plagiarism Detector was able to detect academic dishonesty to a respectable degree. Most notably, in our anonymized Assignment 3 Dataset, PD rated possible cases of cheating vastly above the average similarity value, making them very easy to identify. Additionally, evidence was collected when Professor Beck used PD on a real world dataset that contained confirmed cases of cheating. Of the ten highest instances reported by each program, we found that PD detected 6 cases while Checksims detected 3.

Plagiarism Detector is our own contribution to the effort against academic dishonesty. We strongly suggest that further development is made so that methods, algorithms, and databases like our own can be shared across academia. Future research must focus on gathering larger datasets and defending against alternate methods of obfuscation of cheating. The continuous accumulation of these solutions is essential to keeping up with the ever changing world of programming languages and code structures. While a fully language-agnostic solution, similar to

MOSS, may one day be achievable, we believe language-specific solutions like our own are incredibly valuable to the cause.

References

Aiken, A. (n.d.). *Plagiarism Detection*. Moss. Retrieved April 28, 2022, from https://theory.stanford.edu/~aiken/moss

Bostock, M. (2012, April 10). Les Misérables Co-occurrence. Retrieved April 28, 2022, from https://bost.ocks.org/mike/miserables/

Editors. Racket Lang. (n.d.). Retrieved April 26, 2022, from https://docs.racket-lang.org/gui/editor-overview.html#%28part._editorfileformat%29

genchang1234. (n.d.). *How to cheat in computer science 101*. GitHub. Retrieved from https://github.com/genchang1234/How-to-cheat-in-computer-science-101

Grolleau, G., & Czibor, E. (2016, January 27). Cheating and loss aversion: Do people cheat more to avoid a loss? Management Science. Retrieved April 28, 2022, from https://pubsonline.informs.org/doi/10.1287/mnsc.2015.2313

HanSolo. (n.d.). *A javafx library that contains different kind of charts*. JavaRepos. Retrieved from https://javarepos.com/lib/HanSolo-charts

Heon, M., & Murvihill, D. (2015). (working paper). *Program Similarity Detection with Checksims*. Worcester Polytechnic Institute. Retrieved April 26, 2022, from https://digital.wpi.edu/pdfviewer/cj82k8447.

Krou, M.R., Fong, C.J. & Hoff, M.A. Achievement Motivation and Academic Dishonesty: A Meta-Analytic Investigation. *Educ Psychol Rev* 33, 427–458 (2021). https://doi.org/10.1007/s10648-020-09557-7

Liora Pedhazur Schmelkin, Kim Gilbert, Karin J. Spencer, Holly S. Pincus & Rebecca Silva (2008) A Multidimensional Scaling of College Students' Perceptions of Academic Dishonesty, The Journal of Higher Education, 79:5, 587-607, DOI: 10.1080/00221546.2008.11772118

Paaßen, B. (2018). Revisiting the tree edit distance and its backtracing: A tutorial. *arXiv preprint arXiv:1805.06869*.

Racket Lang. (n.d.). *Reader*. Racket Lang. Retrieved April 26, 2022, from https://docs.racket-lang.org/reference/reader.html

Schleimer, S., Wilkerson, D. S., & Aiken, A. (2003). Winnowing: Local Algorithms for Document Fingerprinting. https://doi.org/10.1145/872757.872770

Schwarz, S., Pawlik, M., & Augsten, N. (2017). A new perspective on the tree edit distance. *Similarity Search and Applications*, 156–170. https://doi.org/10.1007/978-3-319-68474-1_11

What is Academic Dishonesty. WPI. (n.d.). Retrieved April 25, 2022, from https://www.wpi.edu/about/policies/academic-integrity/dishonesty

Yang, D. (2019, May 3). How MOSS Works. Retrieved April 28, 2022, from https://yangdanny97.github.io/blog/2019/05/03/MOSS

Additional Materials

Project GitHub repository:
https://github.com/MQP-Academic-Dishonesty-AB2021/PlagiarismDetector