*LopezMOSS: A Software Similarity Scoring System*

*Emil John Lopez1*

*1 De La Salle University*

*\*Corresponding Author: emil\_lopez@dlsu.edu.ph*

**Abstract:** Plagiarism of source code is common in academic institutions and so it threatens the integrity of the educational process. There have been many attempts to detect such activity since it was first popularized in 1994 by a program called MOSS. This paper presents yet another solution to this problem: LopezMOSS. This program reads in either a folder which contains multiple projects to be compared or separate folders. It uses tokenizing and clustering using n-gram fingerprinting, and then it proceeds to tabulate the occurrence of each token. To quantify the similarity of two projects, it yields a score from 0.0 to 1.0, showing the severity of the plagiarism. Various tests were performed to check if the different program operations were being done correctly by the system, all of which passed. Additionally, different software metrics were done to quantify the quality of the source code. The author found that the program could be useful for software similarity testing.

**Key Words:** similarity, plagiarism, token

1. INTRODUCTION

Plagiarism is an age-old problem in intellectual property law. However, the speed of data transfer and the rise of the Web has made it easier to do the act and has made detection much harder. Furthermore, adjuging someone of commiting the act turns out to be very complicated and can lead the overseer and lawyers in a legal grey area, where it is unsure whether the act violates the law. It is important to examine its definition first to attempt to solve the problem. According to a paper, actions that fall under plagiarism include: (1) turning in someone else’s work, (2) copying someone’s idea without giving credit, (3) not putting quotation marks, and (4) changing words only without changing the structure of the sentence (Maurer, Kappe, & Zaka, 2006).

In educational institutions, software assignments are usually subject to this form of malpractice and so it threatens the integrity of the educational process. In particular, it is subject to the first and second definitions stated above. However, given the number of students each professor must handle, and the large amount of effort required to perform ad hoc comparisons between them, there is a need for a tool that reliably detects plagiarism and can look past obsfucation, reordering, refactoring and other methods of deception (Bowyer & Hall).

The first significant solution to this problem was introduced in 1994 by an associated professor in UC Berkeley. It used winnowing, a local document fingerprinting algorithm that grouped grammatical tokens in groups of some number of tokens, hashed them to minimize storage space, and counted the frequency of each group using their hash value. This is directly derived from other techniques such as Karp-Rabin String Matching (Scheilmer, Wilkerson, & Aiken, 2003)

Fingerprinting algorithms calculate numbers (which we call fingerprints) to help identify a document. Conflicts within these numbers usually indicate that some part of a document (or segment of code in this case) is similar to another document. Better fingerprinting algorithms have since been derived from this method, but it remains that the foundation of most of them is either Karp-Rabin String Matching or n-gram fingerprinting (Heon & Murvihill, 2015)

This paper will present an implementation of a variation of these key fingerprinting algorithms. In particular, it showcases a version of n-gram fingerprinting. The implementation will also have a graphical user interface (GUI) that will allow its user to simply pick directories which correspond to projects.

# 2. RESULTS AND DISCUSSION

## 2.1. Running the Program

2.1.1.Projects Folder Menu

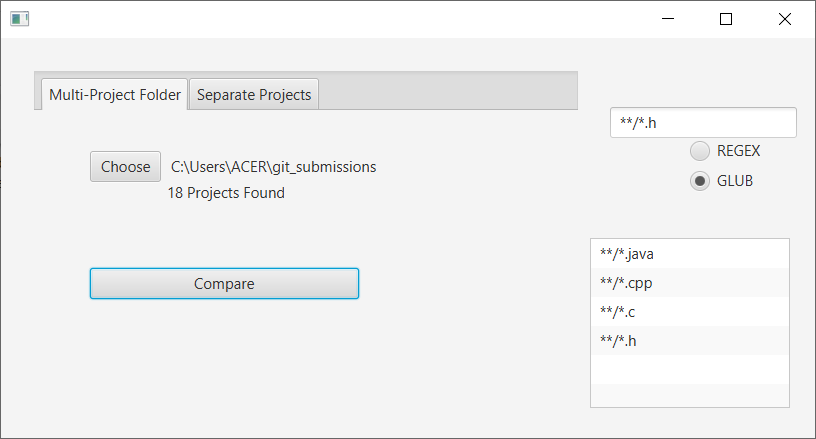


Fig. 7. Configured GUI of the Multi-Project Folder Menu

The multi-project menu allows the user to choose a folder which contains all the projects to be compared. Once the user presses *Choose*…, the standard folder selection UI (which varies between operating systems) will appear to allow the user to choose the folder more easily.

Furthermore, the user will be able to choose whether he wants to use a GLOB-filter or a REGEX filter to filter specific types of files he wants to compare. This was discussed in detail in the Algorithm section of the Methodology.

2.1.2. Individual Projects Menu

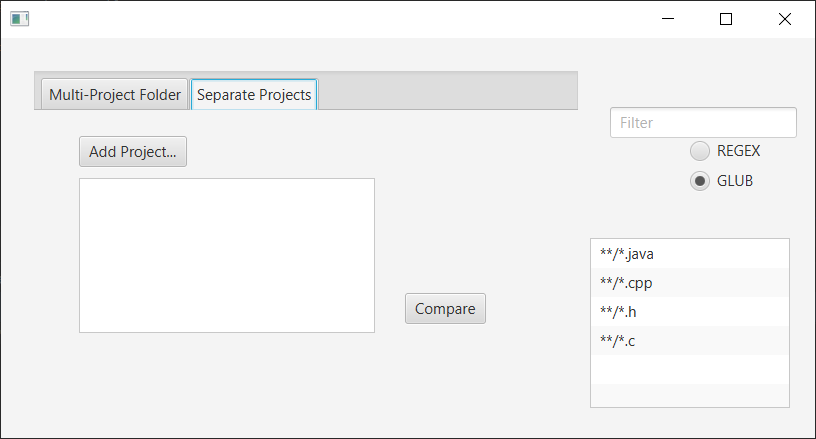


Fig. 8. Individual Projects Menu

The individuals project menu allows the user to pick out the projects separately (like in the previously discussed menu). Like the previous menu, it uses the folder selection UI of the host OS.

2.1.3. Correlation Matrix Menu

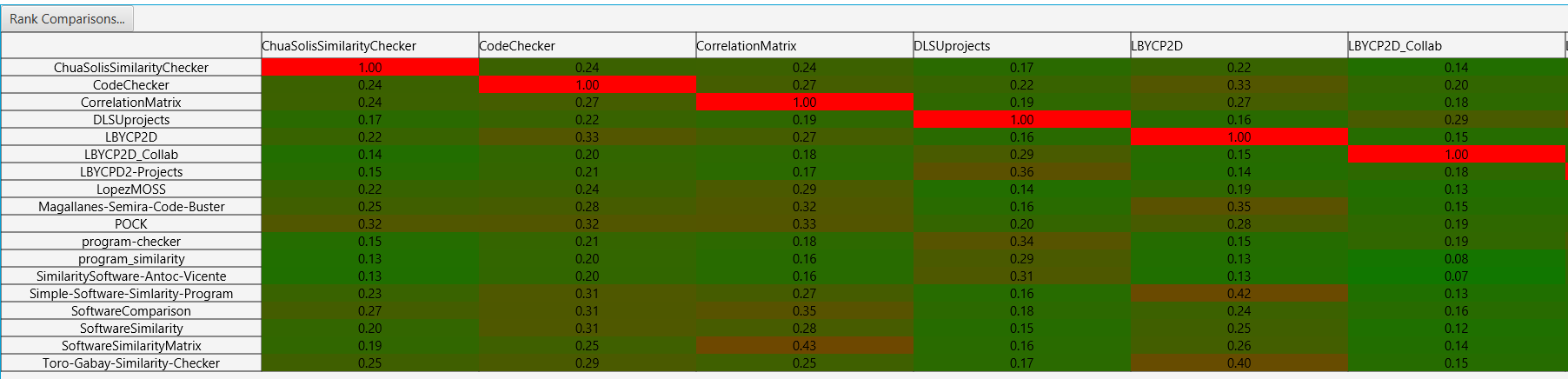


Fig. 9. Correlation Matrix (Part 1)

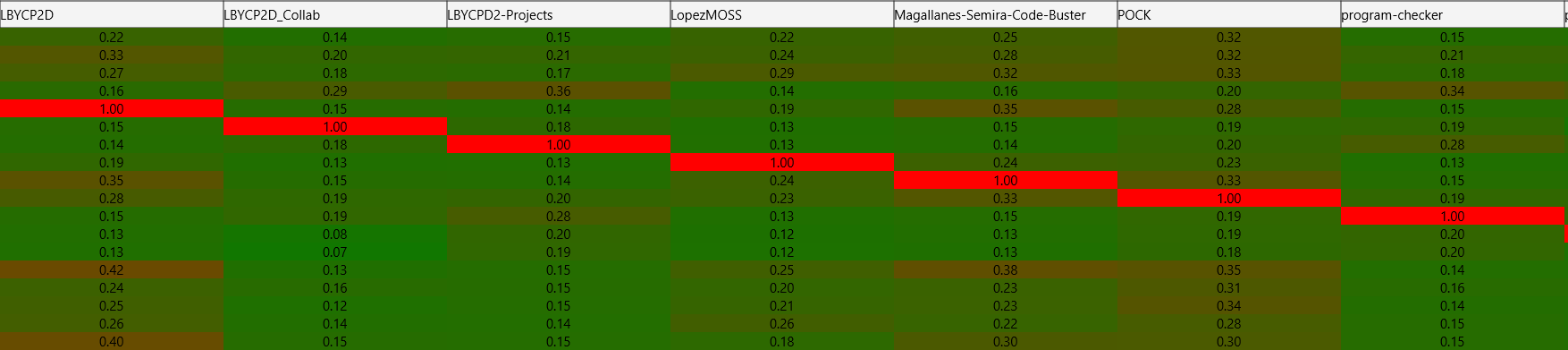


Fig. 10. Correlation Matrix (Part 2)

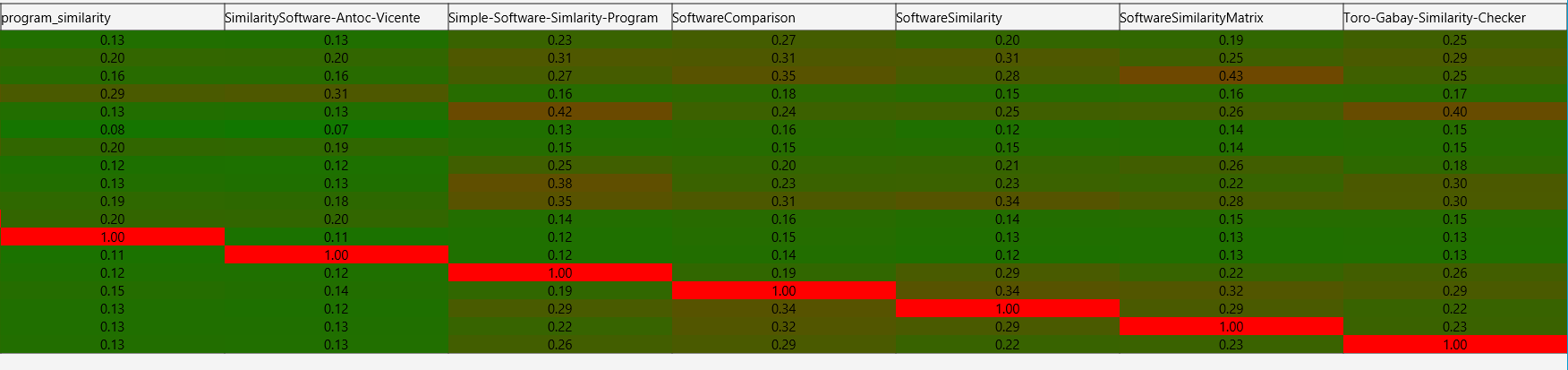


Fig. 11. Correlation Matrix (Part 3)

Once the projects have chosen, the correlation matrix pops up. As can be seen above, each result pair has been assigned a color between green and red to denote how similar the two projects are. This uses a simple linear interpolation algorithm found in Java’s standard library.

***Ranking Menu***

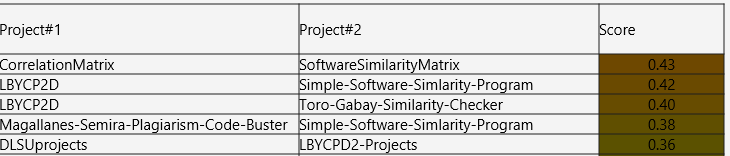


Fig. 12. Ranking Menu

The figure above shows the overall rankings of all comparisons done from highest similarity to lowest similarity. The first column shows the first project compared and the second column shows the second column compared. This was done by flattening the correlation matrix object then sorting it in reverse order.

## 2.2. Program Correctness

2.2.1. Unit Testing

Many unit tests were done on some of the major modules to make sure their behavior remains the same throughout the development of the system. This section describes how each of them were tested and the results of these tests.

***Project Flat Reading Test and Multi-Stream Reader Generation Test***

This is the part of the program that concatenates input streams together into a single reader without extra copies. The test project with multiple files was used to test this and as expected, it yielded a reader that contains a properly combined version of all the test files in the project.

***Projects Test***

This tests all project-based operations (mainly comparison). In particular, it takes in two **Project** objects and compares them. With a lack of ways to check correctness, the only way to make sure the program yields proper results was to check if the output score was between 0.0 and 1.0, the actual range of possible scores.

***Tokenizing Test***

This test checks if the tokenizer properly tokenizes some basic string patterns. To check if this was correct, the program had to yield a **Token** object that had the same underlying value as the tokens expected to be found on the string.

***Token Equality Test***

This test checks if two similar strings (same content, different string) yields *equal* token objects and hash values and different token strings yielded different token objects.

2.3.2. Performance Testing

The performance of the program will be based on how fast some of its key operations will take. However, there is no uniformity here as each computer runs the program will inevitably have different operation times. Nonetheless, the author will try their best to specify the exact conditions for testing. First, the program was executed on an *ACER Aspire E 15* laptop, running on an Intel Core i7-8550U at 1.8 GHz. It has a 12GB DDR3 L RAM (three separate RAM modules, each with 4GB of memory) and a 2000GB hard disk drive. The program was run on Windows 10. The exact condition of the background processes, memory, and the OS cannot be fully documented and will not be included here. The following are the time results given these conditions:

Fig. 13. Operation Time Table

|  |  |  |
| --- | --- | --- |
| Trial | File Loading | Matrix Creation |
| 1 | 548 ms | 1171 ms |
| 2 | 565 ms | 1173 ms |
| 3 | 548 ms | 1139 ms |

Eighteen projects were used for testing and their combined size at the moment of testing was 50.1 MB. Averaging the three results, we find that loading the files takes an average of 553.67 ms and creating the matrix from these projects takes an average of 1161 ms.

## 2.4. Software Metrics

There is a need to quantify some key characteristics of the code of this program. As such, the author turned to the Halstead complexity measures as a means of quantifying aspects of the implementation and expression of the algorithms in this program (Virtual Machinery, 2017). Here, the author used an open-source library by Ahmed Metwally to calculate these values (Metwally, n.d.). However, they modified a lot of the code to make it into an externally usable API and they made a small program which made use of all the library’s internals. The results from this program shows the following:

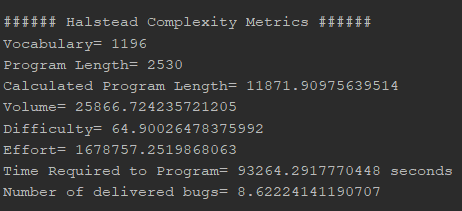


Fig. 13. Results of Halstead Complexity Metrics

The results show that the program had a total vocabulary of 1,196 tokens overall and a length of 2,530 tokens. It found that the volume is large, roughly 25,866.72, which means that the reader must absorb a lot of information from the code before they can understand it. Most interestingly for the author, the metrics estimate that it requires roughly 93,264 seconds or 25.9 hours to make this program.

# 3. CONCLUSIONS

After conducting tests for program correctness, it was found that it does all its operations correctly. The author found that LopezMOSS could potentially be used to do software similarity tests amongst multiple projects in a short amount of time with a small memory footprint.

# 4. ACKNOWLEDGMENTS

The author would like to thank his family, his professor Melvin Cabatuan, his friends and of course, God Almighty for guiding him throughout the development of this system.

# 5. REFERENCES

# Agrawal, M., & Sharma, D. (2016). A State of Art on Source Code Plagiarism Detection. *International Conference on Next Generation Computing Technologies.* Dehradun.

Bien, A. (2014, October). *Structuring Complex JavaFX 8 Applications for Productivity.* Retrieved from https://www.oracle.com/technical-resources/articles/java/javafx-productivity.html

Bowyer, K., & Hall, L. (n.d.). Experience Using "MOSS" to Detect Cheating On Programming Assignments. *Deparment of Computer Science and Engineering*.

Ducarik, M., Krsak, E., & Hrkut, P. (2017). Current trends in source code analysis, plagiarism detection and issues of analysis big datasets. *International scientific conference on sustainable, modern, and safe transport.* Zilina.

Fu, D., Xu, Y., Yu, H., & Yang, B. (2017). WASTK: A Weighted Abstract Syntax Tree Kernel Method for Source Plagiarism Detection. *Scientific Programming, 2017*.

Heon, M., & Murvihill, D. (2015). Program Similarity Detection with Checksims. *Worchester Polytechnic Institute*.

Maurer, H., Kappe, F., & Zaka, B. (2006). Plagiarism-A Survey. *Journal of Universal Computer Science, 12*(8).

Metwally, A. (n.d.). *Halstead Complexity Measures*. Retrieved from https://github.com/aametwally/Halstead-Complexity-Measures

Prechelt, L., Maphohl, G., & Philippsen, M. (2003). Finding Plagiarisms among a Set of Programs with Jplag. *Journal of Universal Computer Science, 8*(11), 1016-1038.

Scheilmer, S., Wilkerson, D., & Aiken, A. (2003). Winnowing: Local Algorithms for Document Fingerprinting. *SIGMOD*.

Sharma, S., Sharma, C., & Veena, T. (2015). Plagiarism Detection Tool "Parikshak". *International Conference on Communication, Information & Computing Technology.* Mumbai.

Tian, Z., Zheng, Q., Liu, T., Fan, M., Zhuang, E., & Yang, Z. (2015). Software Plagiarism Detection with Birthmarks Based on Dynamic Key Instruction Sequences. *IEEE Transactions on Software Engineering*, 1217-1235.

Virtual Machinery. (2017). *The Halstead Metrics.* Retrieved from http://www.virtualmachinery.com/sidebar2.htm