*Software Similarity Detection Using Tokenizing and N-Gram Fingerprinting*

*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

# 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. METHODOLOGY

## 2.1. Requirements

This section presents a very brief summary of the software requirements specifications of this system, including important subsections of the SRS document. All sections not included can be found on the accompanying SRS document, which can be found on the repository for this project.

2.1.1.External Interfaces

According to the requirements, there are five types of input/output data present in the system.

***Projects Directory***

A directory containing all subdirectories pertaining to each project to be compared

***Multiple Separate Projects***

A group of separate directories which contain the projects to be compared

***Path Filters***

An object representing filters that the program uses to tell which projects will be compared. It could come in the form of a REGEX or a GLOB filter.

***Comparison Score***

A score between 0.0 and 1.0 representing the similarity of two compared project readers

***Projects Correlation Matrix***

A matrix-like object representing the output of comparing multiple projects from another.

## 2.1.2. Functions

***Tokenization of Readers***

The system shall tokenize the text content of readers. They will be released sequentially.

***Projects Directory Reading***

This will take in a path and a path filter and it will proceed to perform a depth-first traversal into the path. It will output a collection of input streams of the files filtered in the path

***Project reader concatenation***

This function will take in a collection of input streams and it will combine them into a single file reader to be read sequentially

***Token comparison algorithm***

This function will take in two readers and it will return a score between 0.0 and 1.0 indicating how similar they are.

2.1.3. Performance Requirements

There are a few requirements regarding the performance of this system. First, it must be able to support any number of users on the same machine and it should be able to run on any operating system. There is no limit to the possible number of projects and number of files the comparison will have to handle and read from since it will impose explicit limits to the user. Therefore, the program should be able to handle any amount of data that can be placed on memory and processed by the hardware. However, to maintain toleralability of waiting times, about 50MB of source code should be processed in less than 5 second on average. This is the bare amount of time that a user can notice a delay without being too perturbed by it.

## 2.2. Overall Structure

The entire system is divided into four different parts: the algorithm, the project organizer, the project pairing system, and the GUI. The overall relationship between the parts are shown in the class diagram above. The next few sections are dedicated to providing exposition of how each subsystem works individually and together.

## 2.3. Project Organizer

The project organizer is responsible for all tasks that involve the projects to be used by the system. This naturally includes loading (of project and streams), filtering, concatenating (loading of files into memory), and aggregating them. All the relevant code for this section can be found in the **project** package of LopezMOSS.

2.3.1. Loading and Filtering

The user of the **project** package can choose to load any arbitrary directory and convert it into a **Project** object. This is done by the project builder class, which takes a path and a path filter (which is how the project management system selects which files are relevant). Once both requirements for making a project was given, the builder can be made to build a **Project** object. This object will internally load all the files in the path into *file streams* by traversing the project folder in a depth-first traversal manner.

2.3.2. Concatenation

The program has two requirements: (1) be able to process the source code quickly using buffering and (2) project files have to be concatenated before being fed into the algorithm.

At first, it seemed like these requirements were at odds with one another. The Java **String** class and byte arrays are all immutable, so multiple copies of the same content have to be loaded in to achieve this. This meant that if we relied simply on loading them unto strings, the program would suffer in terms of both time and memory consumption. It was inferred by the author that this was impractical as it severely depletes the system of all its resources.

The solution involves the manipulation of the streams loaded in the first stage directly. The interesting aspect about Java file streams is that they do not load the files into memory beforehand. Since it was already clear from the start that the files will only be used **together**, there was no benefit to loading them into memory separately.

This combined input stream was then converted to a byte output stream and the underlying byte array was taken and stored by the class Now, wheean an outside class requests a concatenated copy of a project folder, it could be loaded directly from the byte array by loading the generated byte array into a **ByteArrayInputStream,** where it would just be referenced internally with no harm and since the algorithm required a **Reader.**

2.3.3. Aggregation

In order to include multiple projects, there needed to be a storage that stores a collection of Project folders. Furthermore, it needs to be constructible using a path that contains multiple projects or a collection of paths because these are the two ways projects can be loaded into the system according to the requirements.

To prevent ambiguity between these two forms of storage construction, two static factory methods are introduced: projectsIn and fromPathCollection. The first one traverses every folder in the projects path and loads them into Project objects and the second one takes the paths from the collection and turns them into project objects. The folder traversal was done using the iterator for the **Path** object

## 2.3. Algorithm

The algorithm used for this program is directly derived from Checksim’s algorithm (Heon & Murvihill, 2015) which uses n-gram representation of tokens to compare two pieces of source code. The algorithm itself can be divided into three phases: (1) hash-tokenizing, (2) clustering, and (3) occurrence counting or tabulation. The pseudocode below summarizes the algorithm albeit abstracting away some parts of the program (which are further elaborated in their own sections):

|  |
| --- |
| Algorithm 3.1. General Comparison Algorithm |
| Input: Reader reader1, Reader reader2   1. Let \_tab1 and \_tab2 be token cluster hash tables 2. \_tab1.tabulate(reader1) 3. \_tab2.tabulate(reader2) 4. return TokeClusterOccurrenceTable.collisionCount(\_tab1, \_tab2) / (\_tab1.total() + \_tab2.total()) |

Fig. 2. General Comparison Algorithm

2.3.1. The Interface

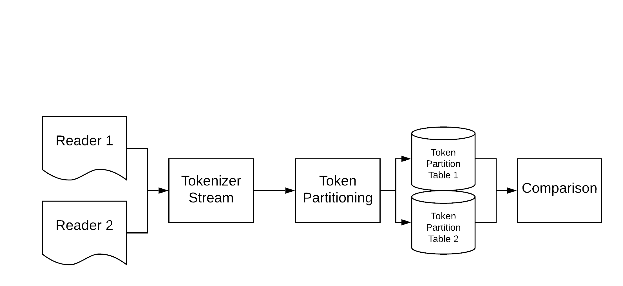
A few requirements were already known to the author at the beginning of the project: (1) the comparison must yield a score between either 0.0 to 1.0 or 0.0 to 100.0 based on the similarity of the two input programs, and (2) it had to compare two projects or two files. By creating a generic interface which reflect these two requirements, the author kept note of the exact form the algorithm class should take. The interface they came up with was as follows:

Fig. 3. General Flow of the Algorithm

The **ComparisonStrategy** class is more precisely a functional interface that encapsulates the algorithm. The use of a floating-point value as a return type makes sure that we can return a score between 0.0 to 1.0. The method also takes in two Readers, which means all classes depending on this class must convert their input to the algorithm to reader form. This is naturally the reason for some decisions made in the project package (as discussed in earlier parts)

2.3.2. Hashed tokenizer

Tokenizing is the process of extracting tokens, the fundamental grammatical units of some language. This section provides some exposition on the inner mechanisms of LopezMOSS’s internal tokenizer, which it uses to perform comparisons. It can be divided into two steps: (1) tokenizing and (2) hashing.

Character-based algorithms are not considered because it would take too long and it is too tedious and line- and word-based algorithms are incredibly vulnerable because they treat programs as plain text and do not take into account that source code actually contains a lot more information than simple words or lines.

In reality, code is divvied up into small units called tokens. They turn out to be the smallest we could break up a program without loss of information. They decided to turn to Java’s StreamTokenizer to do Reader manipulation and tokenizing for them. This class spits out strings, line breaks, white spaces, or floating-point values that correspond to the next token from a reader. This was the central piece of the HashingTokenizer class. To store the results of this tokenizing process, the author created the **Token** class.

Given the purpose of this class, maintaining the value of the tokens was not necessary. Instead, the class only needs to maintain the identity of a token so it could be compared with other tokens. A unique ID is assigned to a token to help with this. It is assigned by taking the hash of the internal value of the token. This unique ID system is done for two reasons: (1) the type of the tokens varied and converting them all to strings before storage just leads to memory bloating and (2) storing actual string tokens is costly in terms of memory and time.

The processes above are done for all tokens from the reader until the system encounters an end-of-file flag.

2.3.3.Partitioning

Tokens alone are prone to false positives in this particular use case. This is true for multiple reasons. The more problematic reason is all languages and libraries have some amount of boilerplate code.

The more localized reason is that some tokens tend to appear in most source code. This can include anything from conditionals to preprocessors.

There are many solutions to this problem, such as examining structure instead of content, or analyzing machine code behavior for similar patterns. This however requires a deeper dive into compiler theory, which the author has limited knowledge on.

The simplest way to alleviate this was using an n-gram representation for tokens. This will work better than individual token comparison in principle because it lessens the boilerplate problem to an extent. This is because there are subtle differences in how boilerplate code is implemented. It will also catch some types of obfuscation. For instance, a variable declaration can be moved around but it will always be clustered to prevent breaking the program.

*//Original:*

*int a = 5, b = 0;*

*int sum = b + a;*

//*Copy*:

*int b = 0, a = 5;*

*int sum = a + b;*

Fig.4. Example of Obfuscation

Notice that even though the code had been reordered, the grouping of the identifiers and values remain the same. This is because our potential culprit wants to preserve the behavior of the program. Furthermore, switching around the order of the operation will also be detected because the n-gram clusters will be designed to be equal even when the tokens are reordered.

N-gram clustering was achieved by using the Tokenizer earlier to gather **Token**s. Every N iterations, the collected tokens are batched together into a TokenCluster object. The hashes of each token are recomputed and combined to make a single hash value for the cluster. Re-ordering the tokens will have no effect on this value so obfuscation by reordering will not work.

These clusters are released via a stream by the **Tokenizer** class. The next module, the occurrence counter, will be responsible for tracking and tabulating each one and eventually finding the number of unique and similar clusters between two projects.

2.3.4. Cluster Occurrence Counting

With the hashof the token cluster class in place, occurrence counting now becomes a straightforward task. When a reader was put into the **TokenClusterOccurrence** class**,** token clusters were extracted from it, then they are placed into a hash map that maps each token cluster to integer pairs that indicate how many times it had appeared for both readers. From this table, the score could now be computed with the **Jaccard Similarity Coefficient** which uses the quotient of the total number of unique token clusters and similar token clusters.

|  |
| --- |
| Algorithm 2.2. tabulate(Reader reader) |
| 1. Let tok be a HashingTokenizer 2. For each TokenCluster cluster in tok.remainingTokenClusters(size))   this.addOccurred(cluster);  } |

The hashing tokenizer seen here works exactly as described in the token hashing and token clustering sections above. *addOccurred* adds the cluster to the hash table counting each token cluster occurrence.

## The Graphical User Interface

The graphical user interface (GUI) uses the JavaFX GUI library built in for Java 8. Unlike most other libraries, JavaFX chooses to make the view file (FXML file) dependent on the controller instead of the old models which does the opposite of this. This is what is known as dependency injection. (Bien, 2014).

The author modelled the GUI classes with the Model-View-Presenter model along with *Services* class to give each menu backend access.

To further build on this model, the author used the *Afterburner* framework, which allows for singleton-like dependency. The two main features of this framework that the author used are (1) the *@Inject* annotation, which injects a singleton copy of a class in a variable at compile time allowing embedding of necessary classes quickly and (2) automatic *View* and FXML binding at compile time which reduces boilerplate and speeds up the program. This can all be done by the framework so long as the classes in a GUI package follow the naming convention below:

*[package\_name]*

*|--[package\_name]View.java*

*|--[package\_name].fxml*

*|--[package\_name]Presenter.java*

*|--[package\_name]Service.java*

*|--[package\_name]Model.java*

Fig. 5. JavaFX Afterburner Package Structure Convention

The role of each one is well-defined by the MVP and MVC architectures, so the author will no longer elaborate on this.

With the architecture of the GUI well-established, each menu will be explained in the major section of this paper

# 3. RESULTS AND DISCUSSION

## 3.1. Running the Program

3.1.1.Projects Folder Menu

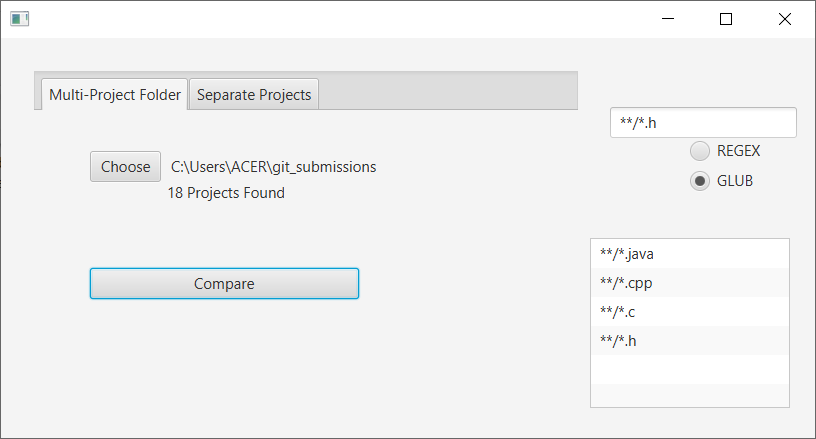


Fig. 6. Configured GUI of the Multi-Project Folder Menu

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.

3.1.2. Individual Projects Menu

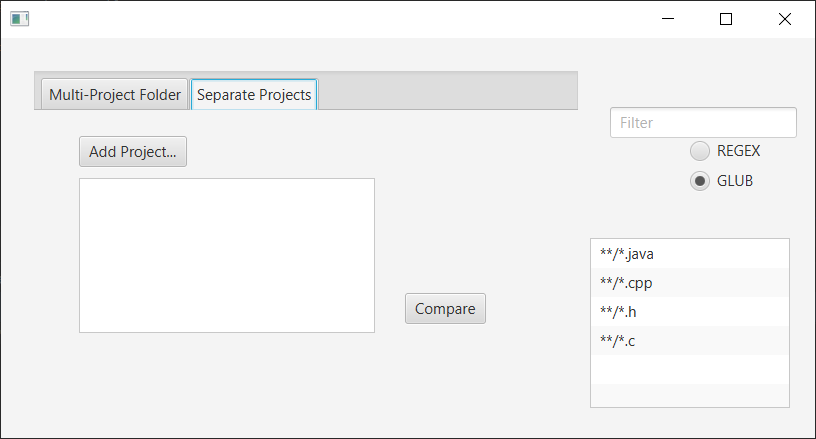


Fig. 7. Individual Projects Menu

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.

3.1.3. Correlation Matrix Menu

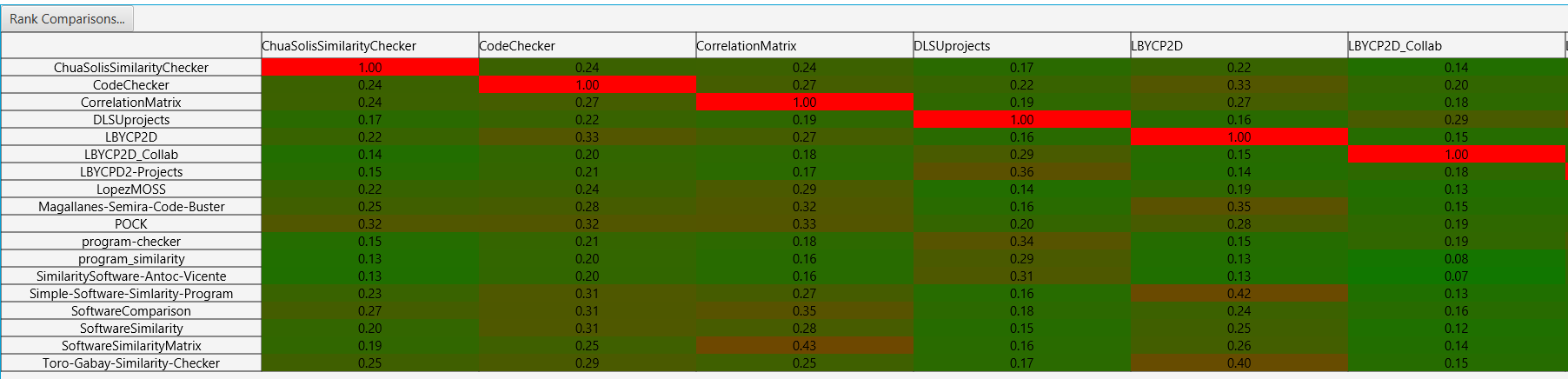


Fig. 9. Correlation Matrix (Part 1)

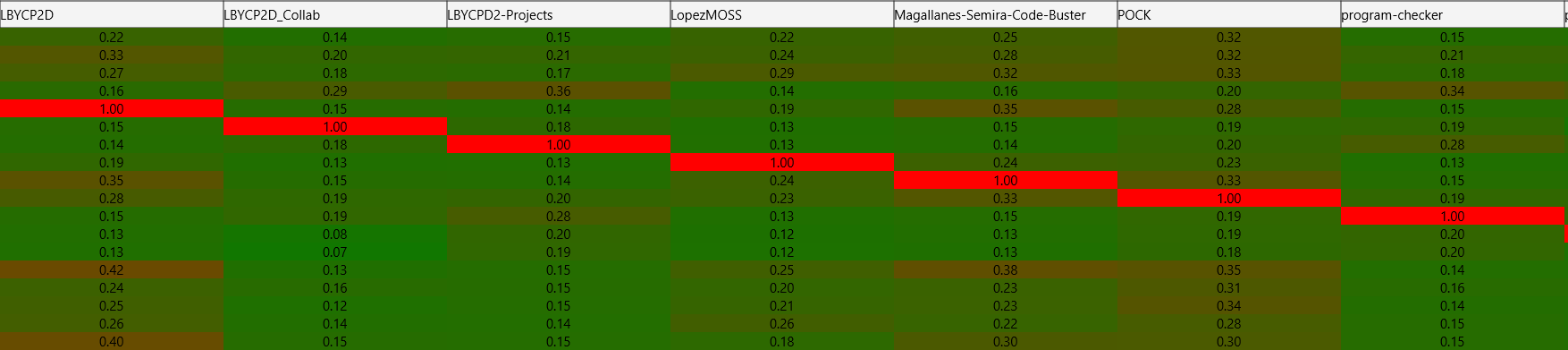


Fig. 10. Correlation Matrix (Part 2)

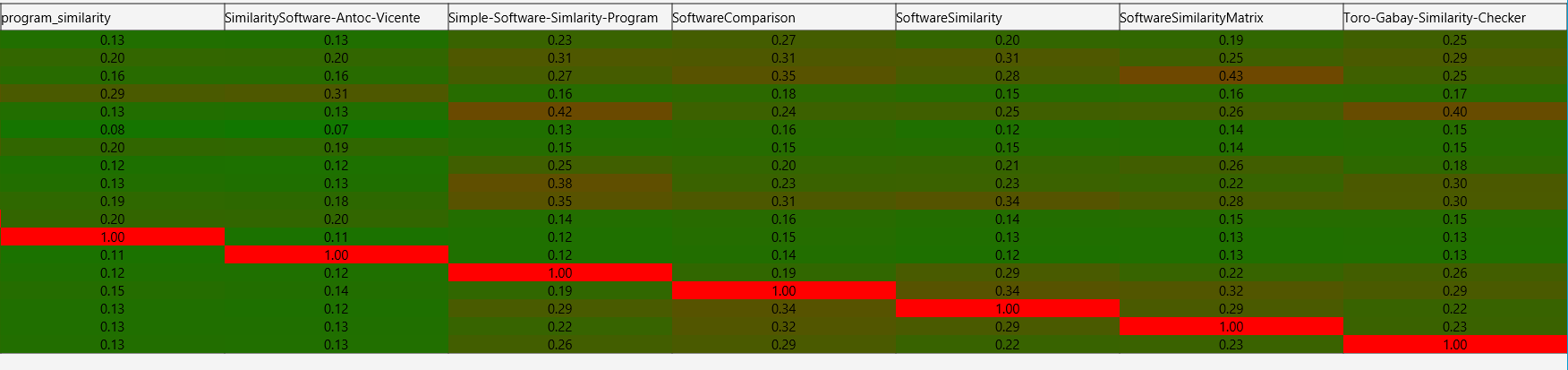


Fig. 11. Correlation Matrix (Part 3)

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***

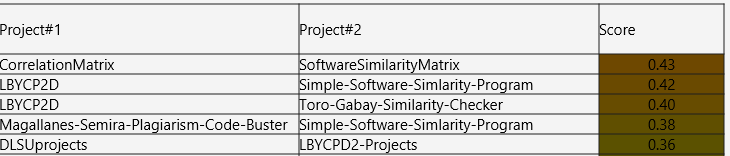


Fig. 12. Ranking Menu

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.

## 3.2. Program Correctness

3.2.1. 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.

3.3.2. 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.

## 3.4. 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:

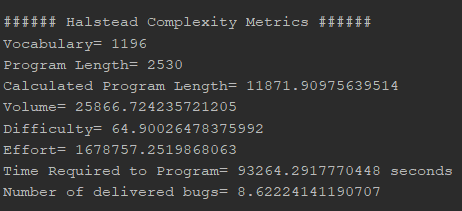


Fig. 13. Results of Halstead Complexity Metrics

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.

# 4. 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.

# 5. 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.

# 6. 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