A Unified Approach to Automate the Usage of Plagiarism Detection Tools in Programming Courses

A. Omar Portillo-Dominguez[†*], Vanessa Ayala-Rivera[†], Evin Murphy[†] and John Murphy[†]
[†] Lero@UCD, School of Computer Science, University College Dublin, Ireland
Email: {vanessa.ayala-rivera, evin.murphy}@ucdconnect.ie, {andres.portillodominguez, john.murphy}@ucd.ie

***Abstract*—Plagiarism in programming assignments is an extremely common problem in universities. While there are many tools that automate the detection of plagiarism in source code, users still need to inspect the results and decide whether there is plagiarism or not. Moreover, users often rely on a single tool (using it as "gold standard" for all cases), which can be ineffective and risky. Hence, it is desirable to make use of several tools to complement their results. However, various limitations exist in these tools that make their usage a very time-consuming task, such as the need of manually analyzing and correlating their multiple outputs. In this paper, we propose an automated system that addresses the common usage limitations of plagiarism detection tools. The system automatically manages the execution of different plagiarism tools and generates a consolidated comparative visualization of their results. Consequently, the user can make better-informed decisions about potential plagiarisms. Our experimental results show that the effort and expertise required to use plagiarism detection tools is significantly reduced, while the probability of detecting plagiarism is increased. Results also show that our system is lightweight (in terms of computational resources), proving it is practical for real-world usage.**

I. Introduction

Nowadays, coding has become one of the most valuable skills to learn [1]. Not only can it help people to get a more remunerable job, but it can also help to develop other abilities, such as problem-solving and computational thinking [2]. For this reason, many educational institutions have added programming courses to their study plans. To evaluate the performance of students, teachers (hereinafter referred as users) usually rely on programming assignments. However, it is common that unethical students plagiarize other people's work. For example, an Australian survey conducted in 2002 reported that 85% of a class at Monash University, and 70% of a class at Swinburne University, engaged in plagiarism during their studies [3].

Hence, detecting plagiarism in programming assignments is important. Nonetheless, this task is very difficult to carry out as it is often conducted manually, which is time-consuming, monotonous, and sensitive to human errors. To facilitate this task, a large number of automatic tools to detect source code plagiarism have been developed [4]. However, various limitations exist in these tools that prevent their efficient usage. Firstly, these tools typically compare two sets of files at a time. Thus, users need to manually define the different combinations of files and feed the tools with them. Even in a relatively modest class (e.g., 50 students), such a manual process can be very time-consuming and error-prone, due to the large number of source code combinations to execute (e.g., 1225 for our 50 students example). Secondly, the excessive amounts of outputs produced by the tools can easily overwhelm users due to the time required to correlate and analyze the results. Thirdly, the tools can differ greatly in terms of features, limitations, and implementation. Due to those differences, some tools may address better some cases than others. Moreover, no tool can assert with 100% certainty if plagiarism has occurred, requiring the user to inspect the flagged cases and then determine if plagiarism has occurred. Hence, it would be desirable to make use of several tools to complement their results. The idea is to make a more informed decision (by combining multiple perspectives) and have a higher certainty of plagiarisms detected (by mitigating each others' limitations). However, comparing the merits of each tool and choosing the best one to use is challenging.

Contributions. In this paper, we propose an automated system (LOUPE) to address the common usage limitations of source code plagiarism detection tools. LOUPE shields the users from the complexity and burden of properly configure the tools by providing a unified approach to run several different tools simultaneously for multiple pieces of work (including historical data). The system automatically gathers the tools' outputs, consolidates their relevant information (considering users' preferences), and presents a useful data visualization that includes a comparative view of the tools. We also present a practical validation of LOUPE, consisting of a prototype, and a set of experiments to assess its benefits and costs.

II. Background and Related Work

Automation is a strategy that has proven successful in multiple domains (e.g., to encourage the adoption of a technique [5]). For instance, the work on [6] presents a semi-automated approach to facilitate the grading of programming code structures, while the work on [7] presents an automated approach to protect sensitive information before sharing it with third parties. Similarly, other works have focused on automatically generating testing data [8], evaluating pedagogical e-learning content [9], or constructing generalisation hierarchies to anonymise categorical data [10]. In the particular case of plagiarism detection of source code, several tools have been developed in order to support users in this task [11]. These tools offer different features that can be used to compare them in a qualitative manner [12]. For example, considering aspects like the supported languages, extendability, quality of results, interfaces, exclusion of base code, and open-

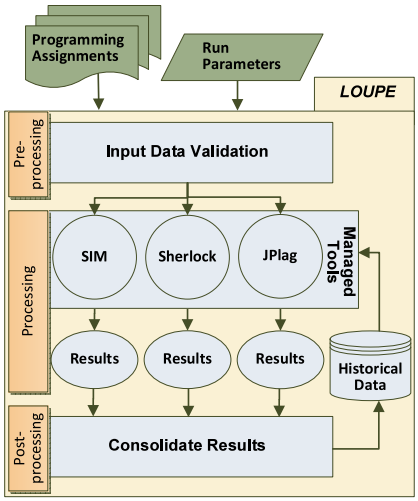

Fig. 1. LOUPE Overview

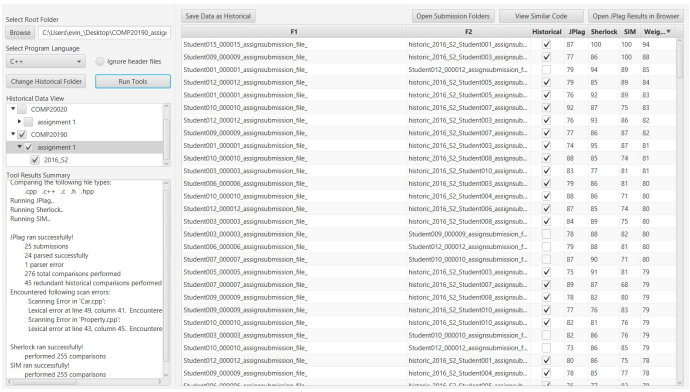

Fig. 2. Main Window with Results

sourceness, among others. Some of the tools that have been identified in the literature include JPlag, MOSS, SIM, YAP, Sherlock, CodeMatch, Big Brother, and Plaggie [4]. However, to the best of our knowledge, no other meta-software has been created to efficiently automate the execution of multiple tools and enable the user to easily compare their results for making better-informed decisions.

III. Proposed Solution

The main objective of our research was to automate the manual processes commonly involved in the usage of plagiarism detection tools for source code. The aim was to decrease the effort required to effectively use this type of tools. Next, we discuss the different stages in LOUPE and its design aspects.

A. Program Flow

Fig. 1 depicts the conceptual view of LOUPE. The interactions between the system, its inputs, and the supported tools fall into three main stages: a pre-processing stage, where the input is validated and initially processed; the processing stage, where the execution of the tools happens; and a post-processing stage, where the results for each tool are consolidated and presented on a graphic user interface (GUI).

1) Pre-Processing Stage. Here, the user provides a set of programming assignments and the parameters required to properly execute the plagiarism tools. Specifically, LOUPE requires the source directory for the assignments, the programming language used, which historical data should be used (if any), the desired tools' preference weights, and if any specific types of files (among the known extensions associated to the chosen language) should be ignored. Next, LOUPE validates such data (e.g., checking empty inputs, expected formats) to ensure the successful run of the tools. Finally, LOUPE prepares the assignments to be fed to the tools. This may involve concatenating multiple files into one, which the serial-based tools will use in their comparisons.

2) Processing Stage. Here, the execution of the supported tools occurs. Specifically, this stage encapsulates the logic referring to how the tools are run, to gather their results, and to extract the relevant information for results presentation. Internally, LOUPE constructs the command required to run each tool (as they have a different syntax and set of options available) and handles their execution. This abstraction ensures that our system can be easily extensible for other tools, as they share the same type of processing functionality (e.g., either parallel or serial tools, as detailed in Section III-B). If historical data is selected, the tools will also consider the assignments of previous semesters' course in their comparisons. Once all tools have finished their executions, their overall outputs are collected and parsed to extract the pertinent information, such as the assignments compared and their similarity score.

3) Post-Processing Stage. Here, the results from each tool are consolidated to obtain the similarity for each pair of assignments per tool. Then, a weighted average of the tools' similarity values is calculated for each comparison. This weighted score facilitates the decision-making as it aggregates the "plagiarism certainty" of all the tools. As a result, the user can be more confident about the occurrence of alleged plagiarism cases and concentrate on those. Moreover, the weights are configurable to suit the user's preferences. For example, they can be set based on the reputation of the tools or their novelty. The combined results are then presented in the GUI (as depicted in Fig. 2), along with any extra information produced by the tools. For instance, with SIM is possible to visualize the sections where two assignments had similar code. Furthermore, the user has the option to save the newly processed submissions as historical data (so they can be used in future runs), or to export the results in comma-separated value (CSV) format.

B. Architecture

LOUPE follows a Model-View-Controller (MVC) [13] object-oriented architecture where every component is represented as an object and interacts with each other in order to execute the system. Considering that one of our key goals was to reduce the amount of expertise needed to run a plagiarism detection tool, the architecture has been designed to be easily understandable, highly extensible, lowly maintainable, and simple to use. Fig. 3 shows a simplified view of the system.

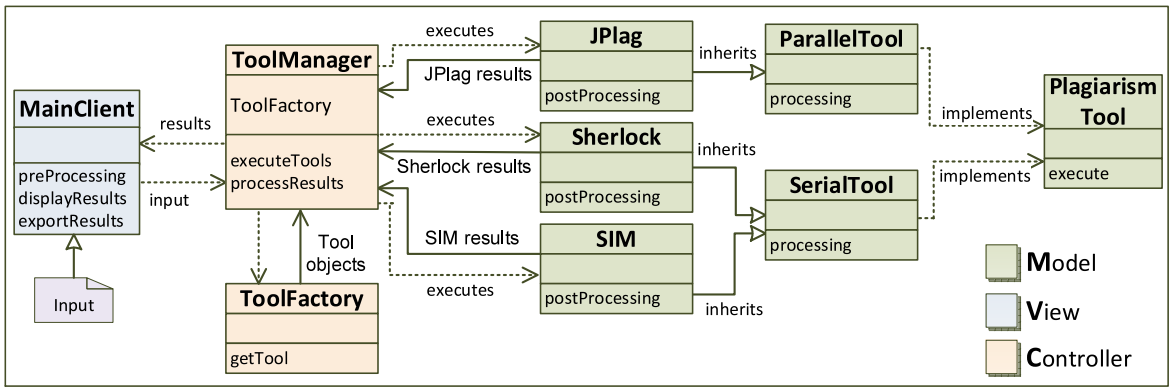

Fig. 3. LOUPE's Architecture

It can be seen how the *MainClient* handles any tasks that are associated with the pre-processing stage, as well as the display of the results. This client also interacts with the *ToolManager* which implements a *facade* design pattern [13] to control the execution of the supported tools and the gathering of their results. Besides, a *ToolFactory* class [13] is responsible for the instantiation of the different tool implementation objects, which are later used by the *ToolManager*. In this part, it was important to find coincidences among the tools in order to model them as a reusable hierarchy of classes and reduce the code redundancy. In this manner, the system can be easily extended to support further tools.

As of now, LOUPE supports two types of tools: (1) Folder-based tools, which only need to be executed once (per pair of submissions) as they can natively iterate the inner structure of the compared directories, and return consolidated results for all file comparisons performed. (2) File-based tools, which need to be run multiple times as they do not have built-in support for performing such cross comparisons, instead, they perform pairwise file comparison. Thus, LOUPE must manage their execution to produce a total similarity between two folders. Considering this classification of tools, we created two abstract classes that encapsulate the common logic required to execute each type of tool (as shown in Fig. 3): a *ParallelTool* class for folder-based tools, and a *SerialTool* class for file-based tools. As the tools produce their results in different formats, each concrete tool class has its own method to process their output and extract the relevant information (e.g., similarity scores per folder/file combinations). The data from each tool is then sent back to the *ToolManager* who consolidates it and sends the final results to the *MainClient* to be presented in the GUI.

Finally, it is worth mentioning that LOUPE leverages multi-threading to perform the comparisons concurrently (within the limits of the computer where LOUPE runs). This design decision was taken with the aim of reducing the time complexity of the tools. To illustrate the need for this optimization, consider that the processing of a *ParallelTool* has a quadratic time complexity, relative to the number of submissions in the directory. Thus, the total number of comparisons for a directory containing n submissions is $\frac{n \cdot (n-1)}{2}$, where every submission is compared with every other (excluding itself), and no repeated comparisons. Regarding a *SerialTool*, its complexity is even higher (as it is quadratic relative to the total number of files in all submissions). Likewise, the complexity increases in the presence of historical data, as the number of submissions to compare against increases (roughly linear) with respect to the years of history.

IV. Experimental Evaluation

In this section, we describe the experimental configuration used in our evaluation and discuss the results obtained.

A. Experimental Setup

We conducted a series of experiments that pursued three objectives: (i) To assess the benefits of using LOUPE (i.e., its effectiveness to enhance the detection of plagiarism in source code); (ii) to assess the costs of using LOUPE (in terms of computational resources); and (iii) to compare LOUPE's benefits and costs against those of manually identifying plagiarism. For this purpose, we conducted two types of test runs: without historical data and with historical data. In the first type, we fed the plagiarism tools (implemented in LOUPE) with all the evaluation data that is composed exclusively of submissions belonging to the same year and assignment type. Namely, the only sources of potential plagiarisms are the students of each class in the same year. In the second type, we used all the evaluation data belonging to the same assignment type (but not necessarily the same year).

As evaluation data, we used three years of historical information (i.e., 2014-2016) belonging to a C++ undergraduate programming module taught in the university. Each yearly data is composed of nine different assignment types (i.e., numbered from 1 to 9). The total dataset comprised 224 students and a total of 1420 different submissions. Regarding the implemented plagiarism detection tools, our current prototype supports JPlag (JP) [14], Sherlock (SH) [15], and SIM [16]. These tools were selected because they are well-known and widely used in the literature. Also, as extensibility was a key design consideration for the prototype (i.e., easily adaptable to integrate further tools), we initially opted for tools that can be used locally (i.e., without the need of external dependencies, such as web services). In order to diversify the tested scenarios,

two sets of weights for the tools were used: (1) Uniform (uni), a conservative weight combination in which the outputs of each tool have the same importance (i.e., 33% for each tool). This weight was considered under the assumption that, if all the tools coincide, it is more likely that the suspected plagiarism is actually true. (2) Non-uniform (non-uni), an alternative weight combination to exemplify the scenario in which the user considers the merits of each tool. For example, based on a qualitative analysis of the tools, we decided to assign to JP and SIM a weight of 45% each, and a 10% to Sherlock. The reason is that JP and SIM have knowledge of specific programming knowledge keywords, allowing them to ignore lexical changes such as comments, whitespaces, and identifiers (hence potentially achieving a higher accuracy).

As evaluation criteria, we assessed the benefits of LOUPE by measuring users' productivity (i.e., the number of suspected cases of plagiarism that can be identified per hour) and the time invested in the plagiarism detection process. In terms of costs, our main metrics were the execution time, CPU (in %), and memory (in MB) usage. The execution time involved the lapse between starting the plagiarism detection process until getting the consolidated results' report. Garbage collection (GC) was also monitored as it is a critical performance concern in Java [17] (which is the language used to develop LOUPE's prototype). Finally, all experiments were done in an isolated test environment using a machine running Windows 8.1 64-bit, with 8 GB of RAM and a CPU at 2.20Ghz. The Java environment was Hotspot JVM 1.8 with a heap of 1 GB.

B. Experimental Results

Due to space constraints, we only present the most relevant results (as the performed experiments involved the evaluation of 96K different combinations of undergraduate submissions).

Benefits. Our first analysis focused on assessing the advantages of integrating different plagiarism tools (i.e., diverse perspectives) in order to have more certainty that a suspected plagiarism case is worth investigating. It is important to remark that our intention was not to exhaustively study the accuracy of the individual tools, but to document how their complementarity (exploited by LOUPE) is useful to offer a broader view of the potential plagiarisms cases. This would strengthen plagiarism checking and increase the confidence in the outputs obtained as sufficient evidence would be provided. The notion of plagiarism is not a polar question, instead, it is represented as a score indicating the amount of similarity between two different programs (e.g., from 0% to 100% for the implemented tools). Therefore, a percentage threshold is usually set to determine which cases should be reviewed. In our experiments, we set it to 90% in order to offer a conservative perspective of the results obtained.

Fig. 4 depicts the total cases of alleged plagiarisms (across the nine assignments) found per year with each evaluated perspective. It can be seen how the results obtained for each tool significantly differ among them. Consider, for example, LOUPE(uni), which shows the cases where the three tools concurred. In the results of the year 2014, it can be noticed

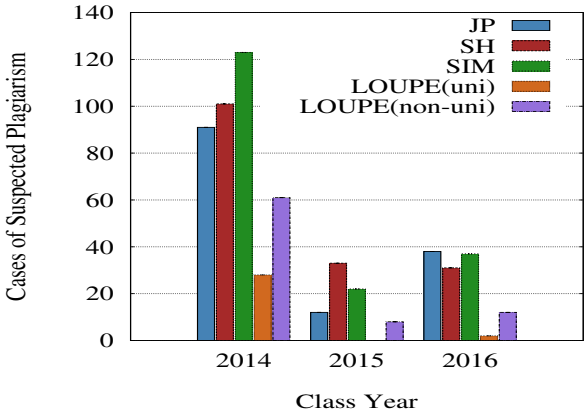

Fig. 4. Cases detected per year for each perspective

that the tools only concurred in 23 cases (even though each tool identified more than 80 cases). Moreover, in the years of 2015 and 2016, the level of agreement of the tools was minimal. This can be explained by the fact that the number of suspected plagiarism cases found by the individual tools was small. Thus, there was a lower probability for the tools to coincide in the identified cases (as opposed to 2014). For those scenarios, conducting plagiarism check with alternative weight combinations can be useful, as shown by LOUPE(non-uni). In that situation, the user can still focus on a considerable small subset of cases.

To further exemplify the risks of only consider a single plagiarism perspective (i.e., tool), we show the breakdown per assignment for our most recent dataset (i.e.,2016) in Fig. 5. It can be seen how the number of alleged plagiarisms is considerably higher in the last assignments (i.e., 7 to 9). These results suggest that students are more tempted to plagiarize at the end of the semester. This sounds reasonable as students are typically busier at that period, when they have to work on more tasks in parallel (e.g., study for exams, work on other projects). Thus, they do not have enough time to do the assignments or they feel more pressured to obtain a high grade. Additionally, the fact that the complexity of the assignments gradually increases (i.e., 1 is often the easiest, whereas 8 and 9 are the hardest), can also partially explain the trends. Under these circumstances, a weighted score can be useful. For instance, in assignment 9, the tools did not agree in their suspicious cases, hence, making difficult to determine which cases to pursue. However, by using an alternative weight combination, LOUPE(non-uni) in this case, some suspicious plagiarisms were identified. A similar situation occurs in assignment 8, where any of the two weighted scores would be useful.

To complement this analysis, we present another set of results that show the benefits of considering historical data to carry out the plagiarism check (as LOUPE does). Even though the syllabus of a module may evolve through time (to include recent trends in its field), these changes typically occur gradually. This means that, in the short term, it is fair to assume that the assignments might be similar in consecutive years (or at least of similar complexity and/or

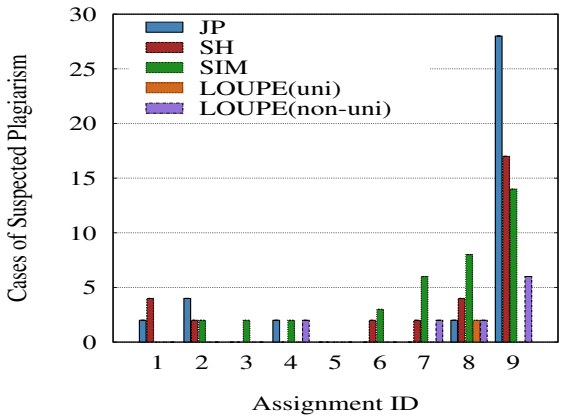

Fig. 5. Breakdown per assignment for 2016

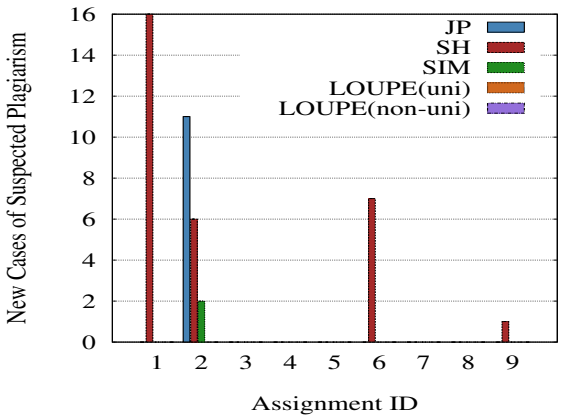

Fig. 6. Additional cases detected with historical data

structure). Considering this, it would be valuable to assess if any plagiarism has occurred across academic years (i.e., a student had access to assignments from previous years). Fig. 6 depicts the results of the historical data test run. It can be seen how new cases of suspected plagiarism were identified as a result of considering all the historical data (in comparison with only using the data from 2016). Although the individual tools do show some differences in the number of additional cases identified, for our particular evaluation data, LOUPE could not identify any strong cross-year plagiarism cases (for any of the two weighted scores). Moreover, the highest number of newly-identified cases were in assignment 1 which is probably misleading. This is possibly a false positive caused by the simplicity of the code involved, hence the high similarity detected by SH. In general, not having found a coincidence between the tools was a useful insight as it exemplified the benefits of using historical data. Otherwise, the user would not know if this type of plagiarism was occurring for the module.

Comparison of Plagiarism Detection Approaches. This analysis centered on assessing the time-savings that LOUPE can bring to the plagiarism checking process. For this purpose, we conducted a survey among a group of teaching assistants (TAs) that were originally involved in the grading of our evaluation data. According to their responses, TAs used two different approaches to detect plagiarism: Some TAs were solely interested in finding very notorious plagiarisms (e.g., two students practically submitting the same code), hence, they mainly performed a visual inspection of the code as a piggybacking of their grading; other TAs utilized a more formal method by using the Sherlock tool (manually configured and ran), however, they only checked a subset of the submissions as it was unfeasible (i.e., too time-consuming) to evaluate all possible combinations. For instance, even in a relatively manageable cohort of students (e.g., around 75), the number of combinations (per assignment) is not trivial (more than 2K). Hence, the effort associated to not only configure, but also process, analyze, and consolidate the results, is significant. This is even more challenging if historical data is taken into account. For example, in our experiments, considering

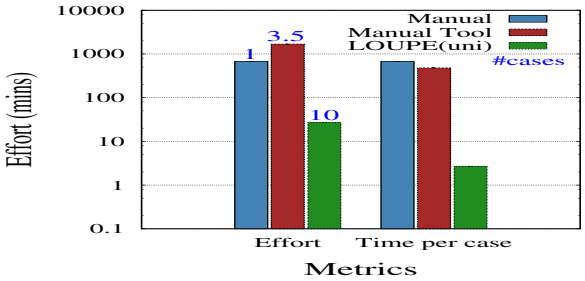

Fig. 7. Plagiarism Detection Effort

two years of history increased the number of combinations to a little bit more than 5K. This analysis allowed us to establish an initial baseline to which LOUPE could be compared against. More specifically, three plagiarism detection approaches were compared: manual inspection, software-assisted using a single tool (manual tool), and LOUPE (automated system).

Our results showed that significant time-savings can be obtained by using LOUPE, without sacrificing accuracy (as it is actually higher than the manual approaches). This can be seen in Fig. 7 which presents a comparison of the evaluated approaches in terms of the time spent in plagiarism detection and the average number of suspected cases identified. This figure also shows the effort invested in identifying a single case. It can be seen how manual was relatively less time-consuming than the manual tool approach. This is because, in manual, the user inspects a smaller number of submissions, as plagiarism detection is performed as a piggy-back of the grading (investing approximately one minute per student submission). Not surprisingly, the manual approach was the most ineffective (as only one case was identified) and the most expensive in terms of time spent per case (with a productivity of only 0.09). Manual tool approach proved to be the most effort-intensive as it required above 1000 mins during the semester (investing approximately 2-3 mins per student submission). However, this additional effort was compensated by the fact that a higher number of cases were identified (i.e., 7 cases during two years, resulting in an average of 3.5 per module/year). In contrast to the manual approaches, LOUPE required the least amount of effort. This was the result

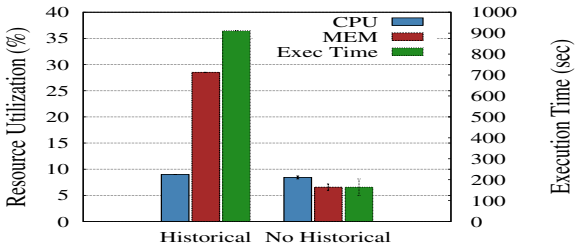

Fig. 8. LOUPE's Costs

of only requiring the user to select the folder with the submissions, and then perform a quick review of the generated report to see which comparisons exceeded the defined plagiarism threshold. LOUPE was also the most effective in terms of the number of cases identified (i.e., 10 cases) and the time spent per case (an average of 3 mins per case). As a result, LOUPE achieved the highest productivity (i.e., 22.22 cases per hour). It is also worth mentioning that all the cases detected by manual means were subsets of the ones detected by LOUPE.

Finally, it is important to remark that the reported effort did not consider the final review performed by the user to determine if a suspected case was, in fact, a plagiarism. This is currently out-of-scope for the following reasons: (1) The review is something that is required independently of the approach used. Nonetheless, this task would be facilitated using LOUPE as it leverages the outputs of the tools to present in a GUI the sections of code that are similar. (2) The final decision to determine over an alleged plagiarism case typically involves an investigative meeting with the teaching staff and the implicated students. At the end, the resolution might be influenced by more subjective elements. For instance, the judgment of the staff and the reasons of why the code was similar, like the students working together on the assignment.

Costs. This analysis concentrated in studying the costs of using LOUPE. Fig. 8 depicts the execution time, average CPU, and memory utilization for both types of test runs performed. LOUPE proved to be lightweight in terms of CPU and memory. The average CPU usage did not exceed 10%, likewise, no major difference was observed between the two types of test runs. The average memory consumption was also low, as it never exceeded 30% (i.e., 300MB) of the assigned memory. This peak was reached by the historical test run, which is reasonable, as it required to maintain more data in memory to perform the different comparisons (approximately 8MB per assignment, and 70KB per submission). Both resources' usages were considered tolerable because the computer was far from exhausting its resources. Moreover, no significant GC overhead was observed, as the total experienced GC was less than 1% of the total execution time (meaning that the memory settings were appropriate). LOUPE also proved to be efficient in terms of execution time. For instance, the historical test runs (which are more complex as they involved around 5K combinations of submissions per assignment) took an average of 15.19 mins, while the non-historical runs took considerably less (an average of 2.73 mins per assignment).

V. Conclusions and Future Work

In this paper, we presented LOUPE, an automated system that addresses the common usage limitations of plagiarism detection tools for source code. Our solution shields the users from the complexity and burden of properly configuring the parameters of each tool used, gathering their outputs, extracting the relevant information from their outputs, and presenting the results in a consolidated and useful manner. As a result, users can make better-informed decisions about the process of assessing assignments for plagiarism. Our results show that not only the effort and expertise required to use the tools is reduced, but the probability of detecting plagiarism is increased. As future work, we plan to develop a layer of expert knowledge in LOUPE to provide extended information about the patterns of plagiarism detected with the aim of improving the effectiveness of the system. Also, we plan to enhance our prototype to release it as a publicly-available tool.

Acknowledgments

This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094.

References

[1] Burning Glass Technologies, "Beyond point and click: The expanding demand for coding skills," 2016.
[2] J. M. Wing, "Computational thinking." in *VL/HCC*, 2011, p. 3.
[3] S. Burrows, S. M. Tahaghoghi, and J. Zobel, "Efficient plagiarism detection for large code repositories," *Software-Practice and Experience*, vol. 37, no. 2, pp. 151–176, 2007.
[4] V. T. Martins, D. Fonte, P. R. Henriques, and D. da Cruz, "Plagiarism detection: A tool survey and comparison," in *OASIcs-OpenAccess Series in Informatics*, vol. 38, 2014.
[5] A. O. Portillo-Dominguez, M. Wang, J. Murphy, and D. Magoni, "Automated WAIT for cloud-based application testing," *ICSTW*, 2014.
[6] S. Buyrukoglu, F. Batmaz, and R. Lock, "Increasing the similarity of programming code structures to accelerate the marking process in a new semi-automated assessment approach," in *ICCSE*, 2016, pp. 371–376.
[7] V. Ayala-Rivera, D. Nowak, and P. McDonagh, "Protecting organizational data confidentiality in the cloud using a high-performance aonymization engine," *ITT*, 2013.
[8] V. Ayala-Rivera, A. O. Portillo-Dominguez, L. Murphy, and C. Thorpe, "COCOA: A synthetic data generator for testing anonymization techniques," *PSD*, 2016.
[9] S. de S Sirisuriya, L. Ranathunga, S. Karunanayaka, and N. Abdullah, "Can pedagogical evaluation process of web based e-learning be automated?" in *ICCSE*, 2013, pp. 650–655.
[10] V. Ayala-Rivera, L. Murphy, and C. Thorpe, "Automatic construction of generalization hierarchies for publishing anonymized data," in *KSEM*. Springer, 2016, pp. 262–274.
[11] M. Agrawal and D. K. Sharma, "A state of art on source code plagiarism detection," in *Int. Conf. on Next Generation Computing Technologies (NGCT)*, 2016, pp. 236–241.
[12] J. Hage, P. Rademaker, and N. van Vugt, "A comparison of plagiarism detection tools," *Utrecht University. The Netherlands*, p. 28, 2010.
[13] M. Fowler, *Patterns of enterprise application architecture*, 2002.
[14] L. Prechelt, G. Malpohl, and M. Philippsen, "Finding plagiarisms among a set of programs with jplag," *J. UCS*, vol. 8, no. 11, p. 1016, 2002.
[15] M. Mozgovoy, K. Fredriksson, D. White, M. Joy, and E. Sutinen, "Fast plagiarism detection system," in *International Symposium on String Processing and Information Retrieval*, 2005, pp. 267–270.
[16] D. Grune and M. Huntjens, "Detecting copied submissions in computer science workshops," *Informatica Faculteit Wiskunde & Informatica, Vrije Universiteit*, vol. 9, 1989.
[17] A. O. Portillo-Dominguez, M. Wang, J. Murphy, and D. Magoni, "Adaptive GC-aware load balancing strategy for high-assurance java distributed systems," in *HASE*, 2015.