CEUR-WS.org/Vol-2218/paper29.pdf

Towards Automated Student Programming Homework Plagiarism Detection

Aleksejs Grocevs[1] and Natālija Prokofjeva[1]

[1] Riga Technical University, Riga, Latvia

Abstract. In the emerging world of information technologies, a growing number of students is choosing this specialization for their education. Therefore, the number of homework and laboratory research assignments that should be tested is also growing. The majority of these tasks is based on the necessity to implement some algorithm as a small program. This article discusses the possible solutions to the problem of automated testing of programming laboratory research assignments.

Keywords: plagiarism detection, automation.

1 Research Motivation

The course "Algorithmization and Programming of Solutions" is offered to all the first-year students of The Faculty of Computer Science and Information Technology (~500 students) in Riga Technical University and it provides the students the basics of the algorithmization of computing processes and the technology of program design using Java programming language (the given course and the University will be considered as an example of the implementation of the automated testing). During the course eight laboratory research assignments are planned, where the student has to develop an algorithm, create a program and submit it to the education portal of the University. The VBA test program was designed as one of the solutions, the requirements for each laboratory assignment were determined and the special tests have been created. At some point, however, the VBA offered options were no longer able to meet the requirements, therefore the activities on identifying the requirements for the automation of the whole cycle of programming work reception, testing and evaluation have begun [Gro16].

At the moment all of the assessment checking, test executing and running as well as plagiarism check/percentage evaluation is done manually. It is obvious that an ordinary human cannot keep the source code of five hundred similar programs in mind, nor is he able to measure their similarity. That is why similar researches exist that also suggest automated testing implementation [Poze15], [Hasen13].

In order to aid the manual testing the academic department of the University has developed a VBA script that automatically executes and checks the assessments for a compiled program; the test results are recorded in an Excel file. This improvement

has allowed to speed up the evaluation process making it possible for the professor to focus more on the source code, which is a more important than an actual test run.

Some authors [Cheng11], [Thieb15] are offering to use the automated verification of test runs and a plagiarism check as a separate Moodle module, which implements many previously defined criteria at the time the assessment is uploaded to Moodle. However, none of these researches provide the complete solution to this problem.

2 Choosing the Metrics and Evaluating Implementation Possibilities

In order to get a grade, the student has to implement the required algorithm as a program, submit the program in a binary (compiled) form as well as providing its source code. The professor has to test both the program using a subset of pre-calculated input/output data pairs, and its source code equivalence to the binary representation, including the compilation ability and the absence of errors while executing it. It is imperative to evaluate the factors affecting the whole process and to choose the metrics for intercomparison.

2.1 Identified Metrics

Algorithm implementation validation speed is criteria that reflects how fast can the professor obtain the program and its source code, execute and test it using the predefined test patterns. In the University the student has to upload the result of his work to the "Homework" section of the Moodle education portal, where the professor should run and evaluate it after downloading.

Evaluation quality – even in case of simple tasks, such as "implementing a list sorting algorithm" there might be many border cases which can lead to incorrect execution results, but sometimes may not be checked due to time limitation. When using the automated testing solutions, it is possible to pre-create the large number of different tests that check all the aspects of the performance of the given assessment.

Report preparation – it is important to put the data of successful/unsuccessful test runs together and to inform the student of the result. When using on-line solutions there is a possibility to send a test-passed-successfully notification as soon as an assessment has been uploaded and tested. It is also important to record the summary of all students' assessments in form of a table that can later be used to be uploaded to the University education portal.

Plagiarism check – even if the tasks are relatively similar it is crucial to make sure the source code was written by the student himself and not plagiarized from a colleague. This check should be made once by a professor in order to avoid the code consecutive altering so it can successfully pass the check.

Safety check – the binary representation of the program and its source code are usually tested separately in order to save time, and this is most commonly done in that particular order. Although, it is not always possible to quickly detect the malicious code, which is meant to erase or alter the test results or even the testing system itself,

when the solution consists of several files or modules. Therefore, while testing the compiled programs they should be treated as potential malware or viruses that may damage the test environment. Ideally they should be executed in the sandbox environment which ensures the isolation of the potential threat.

Bug fix tracking – if the code has been partially altered (either on the professor's request or during the debugging process) the modified parts of the file are indistinguishable from the previous code in the file. Therefore, the professor has to manually compare two versions of the file in order to detect these changes, or even look through the entire source code file. This problem can be solved by using version control system (VCS). The Moodle system itself does not provide neither an option nor plugin for that, so this possibility can only be considered in the context of implementing it in on-site solution.

3 Automated Detection Solution Development

While creating an on-site solution using the third-party developer tools, it is possible to meet all the abovementioned needs. We propose to use Gogs as a Git-repository – the storage for the source code of all programming assessments; Jenkins as a Continuous Integration server; Docker as a sandbox-environment and Apache Solr or other full-text search system as a plagiarism checker. Many [Cosma12], [Poze15] have approached the problem of detecting the plagiarism in the source code, and some authors [Kiku15] are proposing the solutions that are consistent with the University requirements and can be integrated into the suggested infrastructure. Our suggested infrastructure involves the following workflow:

1. The students codes the task in the program code and uploads/updates it using Git;
2. Jenkins checks all the students' repositories once per minute, sends a plagiarism check request to Sorl whenever a new commit appears, creates a separate sandbox-environment, compiles a program within it and runs the tests. If the tests are passed – Moodle API is used to mark the student's assessment as successfully completed and to upload the source code his behalf. If the tests are not passed or the harmful code was found – the student is notified by an e-mail and has to go back to step one. The Docker sandbox container is being automatically removed either when the tests are completed or by timeout.
3. After the assessment submission deadline, the professor sets all Git-repositories to read-only mode, runs a Jenkins-plugin, which sends a plagiarism check request for each of the assessments to Solr and records the additional data on plagiarism score for every task's source code to Moodle.
4. The only action left for the professor is to check the source codes or to compare the latest source code version with the previous using the built-in Gogs tools.

Such an approach provides the level of checking speed and quality as well as a report preparation level similar to that of the Moodle plugin.

It is possible to use outer systems for plagiarism checks since Jenkins API allows to connect the various external services.

Safety check is at a high level due to the isolated container (sandbox) use, so the risk of system infestation by the malicious code and other destructive actions is minimized.

Tracking of error correction is a standard feature of the Gogs-repository which facilitates the comparison of file versions and change-tracking.

3.1 Automated vs. Manual Verification

The speed and the quality of manual check of each assessment will be definitely lower compared with the automated check. The numerical representation of absolute comparison results is impossible in that case due to average human concentration and performance abilities of each individual.

Report preparation is done manually, therefore both the error rate and the time consumed will be considerably higher when compared to an automated check.

Plagiarism check in general will be less accurate with a large amount of assessments, however human perception makes detecting such cases reflexively or by using additional environment information possible (the plagiarism rate is higher if the students are friends). Furthermore, biased perception and evaluation of the work are also possible due to the human factor.

Safety check depends on the set of rules enforced by the University. These rules define how to verify the programming assessments and how to run the executable files. It is most likely that a professor will primarily check the compiled program in order to return it for correction in case of a program failing the tests. This saves him the time for compiling the program from its source code.

4 Plagiarism Detection Interference

What kind of refactoring, code change or cheating manipulations should the ideal tool detect, how many differences in the code may be considered accidental similarity and when will the code be suspiciously identical? It is difficult to draw a line under this question, multiple studies [Mann06], [Ji07] suggest that ~25%–70% similarity should be considered normal variation (regardless of the code change types involved) and plagiarism itself is source code reproduction with a small number of routine transformations, i.e., without core logic changes and without deep knowledge or understanding of execution flow. The most modern IDEs provide a wide variety of refactoring approaches to change the code structure without affecting the "business logic" – external behavior of the code. However, opposite to a general intention – to reduce code complexity, students can use refactoring to obscure and hide code similarities.

It is obvious that a person, who is not familiar with program development or programming language at all, but knows how to use refactoring tools and interpret IDE warnings/errors, can modify a source code that way that it will be undetected most of the time by most of the available tools. For example, IntelliJ IDEA, one of the most powerful Integrated Development Environments available on the market, provide wide variety of the refactoring tools for multiple supported languages, e.g. Java, PHP,

Python, JavaScript, Kotlin and others. To aid development for programmers (and, unfortunately to make plagiarism masking easier) IDEA refactoring possibilities can cheat almost all source code plagiarism detection tools.

Fig. 1. IntelliJ IDEA first-level refactoring menu

5 Conclusion and Future Work

Hereafter we are planning to investigate the integration with anti-plagiarism systems and the available implementations in this area in details, as well as to create an on-site solution for integrating it into the educative process.

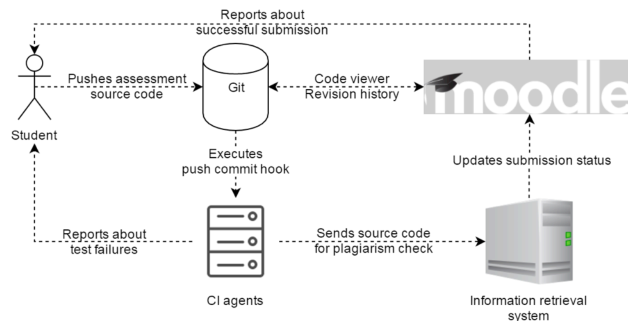

Fig. 2. Proposed plagiarism detection flow

Additionally, to address this issue, our team will continue the research on this problem – further research will be directed towards the next program execution phase, bytecode and instruction execution flow analysis [Gro17]. We believe that this approach will eliminate the need of source code comparison and filter out all kind of refactoring in a natural way, using built-in code inlining and other compiler-specific optimizations. In this case, we can compare the final execution flow using abstract syntax trees, bytecode meta-model generation and other low-level techniques.

References

1. [Gro16] Grocevs, Aleksejs, and Natālija Prokofjeva. "The capabilities of automated functional testing of programming assignments." Procedia-Social and Behavioral Sciences 228 (2016): 457-461.
2. [Poze15] Pozenel, M., Furst, L., & Mahnicc, V. (2015, May). Introduction of the automated assessment of homework assignments in a university-level programming course. In Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on (pp. 761-766). IEEE.
3. [Hasen13] Hansen, D. M. (2013). Paperless subjective programming assignment assessment: a first step. Journal of Computing Sciences in Colleges, 29(1), 116-122.
4. [Cheng11] Cheng, Z., Monahan, R., & Mooney, A. (2011). nExaminer: A semi-automated computer programming assignment assessment framework for Moodle.
5. [Thieb15] Thiébaut, D. (2015). Automatic evaluation of computer programs using Moodle's virtual programming lab (VPL) plug-in. Journal of Computing Sciences in Colleges, 30(6), 145-151.
6. [Cosma12] Cosma, G., & Joy, M. (2012). An approach to source-code plagiarism detection and investigation using latent semantic analysis. Computers, IEEE Transactions on, 61(3), 379-394.
7. [Kiku15] Kikuchi, H., Goto, T., Wakatsuki, M., & Nishino, T. (2015). A Source Code Plagiarism Detecting Method Using Sequence Alignment with Abstract Syntax Tree Elements. International Journal of Software Innovation (IJSI), 3(3), 41-56.
8. [Mann06] Mann S, Frew Z. Similarity and originality in code: plagiarism and normal variation in student assignments. InProceedings of the 8th Australasian Conference on Computing Education-Volume 52 2006 Jan 1 (pp. 143-150). Australian Computer Society, Inc.

9. [Ji07] Ji J, Park S, Woo G, Cho H. Understanding the evolution process of program source for investigating software authorship and plagiarism. InDigital Information Management, 2007. ICDIM'07. 2nd International Conference on 2007 Oct 28 (Vol. 1, pp. 98-103). IEEE.
10. [Gro17] Grocevs, Aleksejs, and Natālija Prokofjeva. "Modern programming assignment verification, testing and plagiarism detection approaches." (2017).