7th International Conference on Higher Education Advances (HEAd'21)
Universitat Politècnica de València, València, 2021
DOI: http://dx.doi.org/10.4995/HEAd21.2021.13045

Considerations when using an Automatic Grading System within Computer Science Modules

Amy Thompson, Aidan Mooney, Mark Noone, Emlyn Hegarty-Kelly
Department of Computer Science, Maynooth University, Ireland.

Abstract

This paper aims to investigate the effectiveness of automatic grading systems, with a focus on their uses within Computer Science. Automatic grading systems have seen a rise in popularity in recent years with publications concerning automatic grading systems usually linked to a specific system. This paper will discuss the factors that need to be considered when using automatic grading, regardless of which system is being used, and will make recommendations for each factor. This discussion is based on the authors' experience of using an automatic grading system in a CS1 environment. From the research conducted, many elements should be considered when using these systems. These include how the code will be tested, the need for plagiarism checks and how marks are awarded. The findings of this study suggest there is a lack of defined standards when using these systems. This analysis of the considerations provides valuable insight into how these systems should be used and what the standards should be built on.

Keywords: *Automatic grading system; CS1; programming; student feedback.*

1. Introduction

An "Automatic Grading System" (AGS) can be defined as any grading tool which provides instant feedback to students or teachers about assignments. However, every AGS handles the amount of feedback given, the amount of data collected, the way it handles errors and other considerations, differently. From a programming perspective, there are more considerations to keep in mind, for example, differing ways to solve a problem, syntactic and semantic errors. In this paper, we discuss the efficacy of AGSs and their usage in Computer Science (CS), with a particular focus on considerations when setting up, and using, an AGS. Discussion and recommendations will be borne from the authors' history of using AGS, and from a deep examination of other similar tools. Before examining any specific use-cases, the overall placement of AGS in the pedagogy of CS must be examined.

Caiza & Del Alamo (2013) performed a literature review looking at AGSs and their implementation. One of the key outputs from the work was determining, for a given set of tools, what programming languages they supported, their architectural design, technologies used by the tool, and how the tool handles code evaluation to grade a student. They found that many of the existing AGSs perform very similarly. The main differences lay in language support and grading metrics. The authors suggest that grading metrics should be normalised to have some form of industry standard. Some of the grading metrics that were discussed include test cases, compilation tests, correctness, typography tests, syntactic analysis, and others. However, we should ask "Are all of these metrics needed in the "perfect" AGS?"

Keuning, Jeuring & Heeren (2016; 2018) examined some of these automatic feedback metrics and techniques and discussed their efficacy by way of a systematic literature review. We will examine this paper in detail in Section 3, but overall, the authors conclude that many AGS are not very diverse. Most simply identify mistakes without providing much context on how to proceed with an incorrect solution. This appears to be the biggest issue with AGS from the literature. They are either too basic to allow for ease of creating questions and solutions, but with little available to "help" students, or they are too specific and complicated meaning the average educator won't be able to immediately begin using it. A recent literature review on Introductory Programming states that "carefully managed automation can increase student performance while saving teaching resources" (Luxton-Reilly, et al., 2018). The perfect balance of ease-of-use and extensive feedback, therefore, is desirable. This is what we aim to discuss in this paper, among other considerations when using an AGS.

CS1 is traditionally the first module that students take at third level when studying CS. This module introduces students to computer programming and threshold concepts. Typically, students will take CS2 in the second semester, which introduces more advanced concepts. We use Java as our teaching language, but research shows the choice of language is not as important as how the language is taught (Noone & Mooney, 2018). Our CS1 and CS2

modules comprise of three hours of lectures per week and a further three-hour lab, where students work on assignments. With approximately 400 students, several initiatives have been used to try to deal with associated issues of teaching large classes, including the opening of a dedicated support centre and the use of PBL, Lego Mindstorms, and response systems.

All of these initiatives were beneficial, but none were addressing perhaps the biggest issue within the class; the amount of time spent in labs grading student assignments. Traditionally, students would attempt their assignments and if they were struggling would seek assistance from demonstrators. Each week, demonstrators would spend a large amount of time grading a random selection of assignments meaning valuable support time was lost. Another issue was the unconscious bias amongst demonstrators grading student's work.

In 2015, we trialled an AGS, the Virtual Programming Lab (VPL) (Rodríguez-del-Pino, et al., 2012). The feedback from all involved was extremely positive and we decided to incorporate the tool in our teaching. This system ensured that a standard grading rubric was applied to all students while also ensuring students were graded for all completed work.

2. Related Work

The field of automatic grading has become increasingly popular in recent years. A survey published on the presentations at SIGCSE looked at the breakdown of papers between 1984–1993 and between 1994–2003. Valentine (2004) showed that the percentage of papers relating to such tools had grown from 18.5% between 1984–1993 to 24.6% between 1994–2003. Software to support AGS was included in the tools papers.

Keuning, Jeuring & Heeren (2016; 2018) examined feedback generated by an AGS, the techniques for generating it, how to create effective questions for a tool and if the feedback was effective for students. They examined 101 different tools with 96% of these tools giving user feedback about their mistakes (to different levels), but only 44.6% of the tools gave ideas on how to proceed. In terms of feedback techniques, automated testing was present in 58.4% of systems with other techniques such as static analysis, tracing, and constraint modelling in a lower percentage of systems. In terms of effective question creation, the authors posit that a good AGS should provide an easy manner of adding new questions with 50.5% of the tools examined making use of model solutions, while 47.5% used some form of test data or test cases. These are the primary option for a functional grading system, with both proving effective depending on the question type. Importantly, however, the authors found that evaluating the effectiveness of these tools is a very detailed task, and most tools do not provide detailed technical analyses for their efficacy.

Examples of the comparison of the students' output to an expected output can be found in Repl.it (Repl.it Classroom, 2021), CodeRunner (Lobb & Harlow, 2016) and Stepik (Stepik,

2021). Code is evaluated against a set of predetermined test cases, made up of given inputs and expected outputs. A grade is assigned based on the number of test cases passed. With Web-CAT (Edwards & Pérez-Quiñones, 2008), students must supply tests with the code to ensure the correctness and validity of their programs. In addition to checking the output of the programs, some AGS have checks for plagiarism. Autolab (Autolab, 2021) uses MOSS (Measure Of Software Similarity) to check for plagiarism between students and past submissions. HackerRank for Schools (HackerRank, 2021) flags submissions that have a similarity of over 70% to other submissions.

OK (OK, 2021), along with checking for plagiarism and expected outputs, provides realtime assignment statistics. These statistics include what questions students are working on, how many students have completed a question, and which students have not started. OK also has integrated automatic feedback and targeted conceptual hints (DeNero et al., 2017). In addition to testing outputs and providing feedback, it may be important to consider the constructs students have used to complete the programming assignment. MULE (Culligan & Casey, 2018) uses pattern matching to look at the constructs within the students' code, as well as looking at test cases. Pattern matching allows for feedback based on what they have written to be provided to the student (Hegarty-Kelly & Mooney, 2021).

3. Considerations and Recommendations when using an AGS

Cheang et al. (2003) notes three components involved in grading programming assignments based on their priority: Correctness, Efficiency, and Maintainability. The maintainability of code is vital in large business settings; it is not viewed as highly within a CS1 environment. These three components are the basis for how AGS should award marks to students. In the following section, we discuss some considerations when using an AGS.

3.1. Complexity of Questions

Students need a clear understanding of what is being asked of them. When wording questions, it can be of benefit not to give them a list of instructions, but rather an open-ended question where they need to determine what constructs are needed in their code. It's also crucial to consider what idea the question is aiming to test. Students need to have a strong understanding of a concept before the complexity is increased (Wang & Wong, 2008).

3.2. Flexibility in Answers

When comparing the expected output to the students' output, only key information should be matched. For example, if the student is asked for the sum of an array, the value they print as the sum is what the system should check for. When a question's output is a Boolean value, additional information is required to ensure the student is not gaming the answer. For example, if a student is asked if a number is prime, and the expected output is True or False,

a student could simply print both True and False to all test cases. The system will find the correct answer each time and award marks. Asking students to print information, such as the number's factors if it is not prime, ensures that they have coded the solution.

If a system is checking for a particular construct, it is essential to note that there are many ways of answering a question. Giving students feedback of how to proceed needs to allow for alternative solution strategies (Keuning et al., 2016). When pattern matching, only constructs that are fundamental to the question should be searched for.

3.3. Test Cases

Test cases are small unit tests that ensure good test coverage. For example, if a question asks students to find the row with the largest sum in a 2D array, the row number should not be the same in each test case. The students' code might need to respond to an invalid input, such as what to do if a number is entered as words rather than numerically. These test cases are important to show students how to respond to incorrect inputs. The required response, such as an error message, should be specified in the question. The number of test cases used should also be considered (Cheang et al., 2003). If there are many test cases all testing the same aspect, some are redundant. However, if there are very few test cases, students may get marks easily even if their code is not properly implemented. The test cases must cover a range of values while not overloading the student with outputs to check.

3.4. Awarding Marks

Students may be awarded marks solely for matching the expected outputs or for also having correct code constructs. Regular expressions can be used to search for patterns within the students' code. For example, if students were asked for a *for* loop in Java then the following regular expression could be used to detect the presence of a valid *for* loop:

```
for\s*\([^;]*\s*;[^;]*\s*;[^)]*\s*\)
```

Figure 1. Expression for the detection of a valid for *loop*

Using regular expressions ensures that the testing coverage is extensive across code. The benefit of pattern matching is that students can easily build up their marks even if their code is not complete or correct thus improving their confidence and motivation levels. Marks can also be awarded if the program compiles successfully. The unwarranted awarding of marks, to students who are hardcoding, can be avoided too.

Some systems store the highest mark achieved, meaning students can continue to submit attempts retaining their highest score. Some systems will take one submission and the mark achieved is the mark awarded. Other systems may have penalties for each incorrect submission. The decision on which approach to take can depend on the human grader's

preference and whether it is a weekly assignment or an exam or for formative assessment purposes.

3.5. Inputs and Provided Code

If students have learnt how to take input into their code, test cases can become simpler to design. If this has not been covered yet, another solution is required. By providing a variable declaration for the test case input, the testing is eased as the system finds the variable provided, changes its value, and the code recompiled to run with the test value. This can be useful for questions where the student needs to take in multiple inputs. Providing the definition of a class or method can also ease issues in the same manner, depending on the language. However, providing this code should be avoided in the early stages of learning. Students need to be competent in declaring a class, method, or variable.

Should students be shown all inputs, all expected outputs, and their incorrect outputs? If shown the expected outputs, it may be easier to hardcode answers. However, it may be easier to solve an issue if they can understand the differences between their output and what is expected. In lab assignments, it is useful for students to see the inputs and expected outputs. However, in exam situations, it may be better to not show expected outputs.

3.6. Feedback

An AGS can be daunting if adequate feedback is not given to the students at all stages of writing code. Compilation errors can be difficult for students to interpret the first time they are seen, as shown in student feedback from Hegarty-Kelly and Mooney (2021). It is useful to give the compiler errors to students in a manner that is easy to interpret. When pattern matching is used to look for constructs, it can also be used to provide feedback to students. If, for example, the question asks the students to use a *for* loop and one is not found by pattern matching, the student can be told that there is not a syntactically correct *for* loop in their code.

3.7. Timing Out

While we may not aim for novice programmers to achieve the best possible run time, the program must run promptly. If there is a loop that will run indefinitely, it is necessary to terminate the evaluation of test cases and give appropriate feedback indicating the issue. When a submitted program will not run due to the time out, there is ambiguity about the grade the student should be awarded. If there was a human grader, the student would get marks for the concepts and constructs they used correctly. This is where pattern matching can be capitalised on, allowing the student to be awarded the marks.

3.8. Plagiarism

Cheang et al. (2003) noted a large increase in the number of plagiarism cases detected after using an AGS, compared to the years prior. Plagiarism is a serious offence and AGS can help solve this, especially in large classes where checking for plagiarism by hand is near impossible. While most programs will contain many of the same basics, programs which are found to be clear substrings of each other can be flagged as suspicious.

3.9. Aesthetics

Comments and indentations tend to have a high importance with human graders as it makes code much easier to read. With an AGS, it can be hard to detect an appropriate comment and indentation (Caiza & Del Alamo, 2013). Pattern matching can find the comment structure, and award marks for using these. Finding indentations with pattern matching is not as straightforward. Students learning languages where indentation is not vital may fixate on it. This will decrease the amount of time spent focusing on the correctness and efficiency of the code. For most, the understanding of the indentation comes with a better understanding of the language. The need for an AGS to find correct indentation is not a high priority.

3.10. Peer Review

Once the questions have been created on the AGS, having a colleague test them can be beneficial. By allowing a colleague to answer the questions, one can ensure the pattern matching is running correctly. It can also uncover other solutions that might use different constructs or exhibit the need for extra pattern matches.

4. Conclusion

We have examined the history of AGS within the CS, how they function and what kind of feedback they provide. We discuss considerations and recommendations for those who are using or plan to use AGSs. The field of AGS is missing a defined standard for what should be included in a tool, and how it should perform. We believe an AGS should do much more than simply give a student a grade. When utilised correctly, an AGS can be of huge benefit to the development of a student's knowledge, providing additional scaffolding. Through effective feedback, effective use of test cases and all the other considerations, a student will have access to what is essentially a "digital tutor". We hope that these recommendations go some way towards defining a standard and helping educators make effective use of AGSs.

References

Autolab. (2021). Retrieved January 26, 2021, from https://autolabproject.com/

Caiza, J. C., & Del Alamo, J. M. (2013). Programming assignments automatic grading: review of tools and implementations. *7th international technology, education and development conference (INTED2013).*

Cheang, B., Kurnia, A., Lim, A., & Oon, W.-C. (2003). On automated grading of programming assignments in an academic institution. *Computers & Education, 41*(2), 121-131. doi:10.1016/S0360-1315(03)00030-7

Culligan, N., & Casey, K. (2018). Building an Authentic Novice Programming Lab Environment. *International Conference on Enguaging Pedagogy*.

DeNero, J., Sridhara, S., Pérez-Quiñones, M., Nayak, A., & Leong, B. (2017). Beyond Autograding: Advances in Student Feedback Platforms. *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, 651-652. doi:10.1145/3017680.3017686

Edwards, S. H., & Pérez-Quiñones, M. A. (2008). Web-CAT: Automatically Grading Programming. *SIGCSE Bull, 40*(3), 328. doi:10.1145/1384271.1384371

HackerRank. (2021). Retrieved January 26, 2021, from https://www.hackerrank.com

Hegarty-Kelly, E., & Mooney, A. (2021). Analysis of an automatic grading system within first year Computer Science programming modules. *Computing Education Practice 2021 (CEP '21)*, 17–20. doi:10.1145/3437914.3437973

Keuning, H., Jeuring, J., & Heeren, B. (2016). Towards a systematic review of automated feedback generation for programming exercises. *ACM Conference on Innovation and Technology in Computer Science Education*, (pp. 41-46). Arequipa, Peru.

Keuning, H., Jeuring, J., & Heeren, B. (2018). A systematic literature review of automated feedback generation for programming exercises. *ACM Transactions on Computing Education (TOCE), 1*(19), 1-43.

Lobb, R., & Harlow, J. (2016). Coderunner: a tool for assessing computer programming skills. *ACM Inroads, 7*(1), 47–51. doi:10.1145/2810041

Luxton-Reilly, A., Albluwi, I., Becker, B. A., Giannakos, M., Kumar, A., Ott, L., Paterson, J.H., Scott, M.J., Sheard, J., Szabo, C. (2018). Introductory programming: a systematic literature review. *Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education*, (pp. 55-106).

Noone, M., & Mooney, A. (2018). Visual and textual programming languages: a systematic review of the literature. *Journal of Computers in Education, 2*(5), 149-174. doi:10.1007/s40692-018-0101-5

OK. (2021). Retrieved January 26, 2021, from https://okpy.org/

Repl.it Classroom. (2021). Retrieved January 26, 2021, from Repl.it: https://repl.it/site/classrooms

Rodríguez-del-Pino, J. C., Rubio-Royo, E., & Hernández Figueroa, Z. (2012). A Virtual Programming Lab for Moodle with automatic assessment and anti-plagiarism features. *International Conference on e-Learning, e-Business, Enterprise Information Systems, and e-Government.*

Stepik. (2021). Retrieved January 26, 2021, from https://stepik.org/catalog

Valentine, D. W. (2004). CS Educational Research: A Meta-Analysis of SIGCSE Technical Symposium Proceedings. *ACM SIGCSE Bulletin, 36*(1), 255-259. doi:10.1145/1028174.971391

Wang, F. L., & Wong, T.-L. (2008). Designing programming exercises with computer assisted instruction. *International Conference on Hybrid Learning and Education* (pp. 282-293). Berlin, Heidelberg: Springer.