The Automation of Grading Programming Exams in Computer Science Education

KRISTUPAS ČEPELIS, University of Twente, The Netherlands

This paper explores the automation of grading the programming exams to lighten the load for the teachers while maintaining high grading accuracy. The research involves two phases: a literature analysis and a prototyping phase. During the first phase, this research will analyze potential ways to automate the grading process, while also considering the potential drawbacks each idea could bring. The analysis will be a foundation for the next phase, which will involve creating a proof-of-concept grading system for the first-year Python programming exams at the University of Twente. The effectiveness of this tool is evaluated by comparing the efficiency and accuracy of automated grading with traditional manual grading. The developed prototype demonstrated a strong positive correlation with the manual grading results, though further refinements are needed to realize its full potential.

Additional Key Words and Phrases: Automated grading, Programming exams, Computer Science, Grading efficiency, Grading accuracy, Grading automation, Teacher workload

1 INTRODUCTION

The popularization of computer science education provided a surge of new additional students [1, 16], making handling the exams harder and more time-consuming. Some computer science courses require teachers to create exams that ask the students to write programming code. This is necessary for the teachers to test whether the students reached the required learning objectives. However, creating and grading those exams uses up lots of staff time [2]. This raises the question of whether automating the grading of those exams is feasible.

One of the current most popular grading methods is to have a team of TAs (short for teaching assistants), consisting of people who have already experienced the course in previous years. It works for some exams, but it has been shown that the grading quality of TAs is degraded due to their lack of teaching experience and misunderstanding on how to tackle grading a student's work [13]. Another popular and more consistent method is for the teachers to grade the exams themselves since they know the learning objectives they are trying to teach best. However, this can be problematic, since most programming exams have a lot of students and it will take many hours and days for the teacher to go through all of them, which introduces fatigue and makes it hard for the grader to make consistent grading decisions [9].

In this research, attempts to lower the load from the graders by automating the process of grading programming exams will be analyzed, and in the end, provide a proof-of-concept automated grading system for the University of Twente (UT) to use in one of their exams in the Technical Computer Science (TCS) and Business & Information Technology (BIT) bachelor courses.

TScIT 41, July 5, 2024, Enschede, The Netherlands

2 GOALS OF RESEARCH

This research aims to develop a prototype of an automated grading system for the programming exams in TCS and BIT studies at the University of Twente. The primary objective is to make the grading process more efficient while ensuring that the accuracy of the grading remains on par with what teachers can offer. Achieving this goal requires an analysis of existing methods and technologies to identify the most suitable approach for automating the grading of the programming exams for TCS and BIT studies. Additionally, to analyze the accuracy of the automation tool, a real-world test will be done using the data from the previous exam(s) held at the UT.

2.1 Research Question

The problem statement leads us to the research question:

How can the grading of first-year Python programming exams be automated without sacrificing the accuracy of the grades?

This can be answered with these sub-questions:

(1) What existing methods and technologies exist to automate the grading of programming questions in computer science education?
(2) Which automation tool or approach is most suitable for automating the grading of exams in the first two modules of the TCS and BIT studies at the University of Twente?
(3) To what extent does the automated grading tool produce grades consistent with those of human graders?

3 RELEVANCE AND RELATED WORKS

Automation of academic exams has existed since the early 2000s, but we have only begun to see more advanced ways to tackle this challenge. Numerous studies analyze and provide a thorough literature review on all of the different methods one can achieve the automation of grading the programming exams [4, 7, 8, 14, 17]. The existing tools can be distributed into two categories: Semi-automatic [3, 5, 19] and fully automatic [12, 15, 22].

Semi-automatic grading involves a human (usually a person who created the exam) to make specific tests that the program will run on the students' exam answers which then gives the scores respectively [3, 19]. This gives the most control to the grader since they get to choose the most important section of the code to give a proper grade. It also greatly reduces staff time and resources needed for checking the exercises [5]. This tool is similar to unit tests that any software engineer can use to test their implementations.

A fully automatic grader is a tool that does not require as much

human input as a semi-automatic tool would. They are more independent and efficient [12]. Automatic grading uses some form of machine learning to grade the answer that the student provided. This can make the grading process even more efficient and save staff time and resources [10, 15]. However, introducing machine learning brings up issues, which require more human intervention for editing the grades [10]. The rest of the research will focus on tools requiring human input instead of relying on machine learning.

4 EXISTING TOOLS

After looking into the related works in this field, a theoretical summary of a few semi-automatic grading techniques can be made to serve as a foundation and an inspiration for the prototype.

4.1 Output Testing

One of the most popular and simplest methods to test the code is by looking at the output. This is a form of dynamic program analysis since the analysis only happens after the code is executed and run. A good example is a tool created by Harvard University for an introductory computer science course called "Check50" [19]. This tool can be used to write specific tests to check the output of a program for any programming language. This can help the student know if they made a mistake by checking on many different outputs. An example of this can be seen in Figure 1, where a check is written to see if the student correctly implemented a function that calculates the Pythagorean theorem using two variables **a** and **b**. However, the tool does not go deeper than that, because it cannot read the file's source code that the student wrote. This means there is no way to check for some learning objectives that may involve using specific programming concepts like lazy evaluation, efficient loops, recursion, etc.

```
Results for . generated by check50 v3.3.11
:) pythagorean.py file exists
    checking that pythagorean.py exists...
:) Input of 3 and 4 into the pythagorean theorem results 5
    running python3 pythagorean.py...
    checking for output "5\n"...
:( Input of 3 and 5 into the pythagorean theorem results in 5.83
    expected "5.83", not "6"
    Did you round the result to 2 decimal places?
```

Fig. 1. Check50 tests for a Pythagorean theorem exercise

4.2 Source Code Analysis

Another way to tackle the grading of programming exercises is by looking at the source code of the program [21]. This is a form of static program analysis, where the code is never executed. One of the most popular tools for static program analysis is CheckStyle, which checks the style of the source code to see if the written code adheres to the programming languages' conventional standards. This differs from grading an exam, but it covers and analyzes the code in a similar technique, which could be adapted for grading an exam.

Some studies attempted to use the source code analysis to grade programming exercises [20, 21]. Another study used the source code analysis to give feedback to the students about their exercises [18]. In particular, the feedback involved checking if the student had reached the course's learning objectives. Evaluating the students' understanding of the learning objectives provided by the course fits well into the topic of automating the exams and it could be done using a form of source code analysis.

Static code analysis can be implemented by parsing the student's program and converting it to an Abstract Syntax Tree (AST). This could be used for checking if the code has any specific functions, implementations, or techniques, which could be useful in the grading process of an exam.

5 ANALYZING SAMPLE EXAMS

To analyze the design of the programming questions, a dataset with a sample exam has been acquired from the UT with anonymous data for research purposes. Exams at the UT aim to evaluate the student's knowledge by checking if they have successfully reached the course's learning objectives by using a format like Bloom's Taxonomy [11]. This can be done by ensuring that the exam uses a format where each question can be graded against an Intended Learning Outcome (ILO) [6]. ILOs are statements of what the student is expected to have learned after they finish a specific study unit. **Table 1** shows all the ILOs that were relevant for the exam of the Python introductory course. The grading rubric of the exam questions highly collides with the ILOs that the question includes.

This dataset is used to represent how the exams should be graded, thus for the remainder of this research, these obtained exams are assumed to be a reference and point of comparison to checking the grading results. The exams use the Python programming language and the questions are split into three categories that each represent knowledge required from the student to solve the problem correctly. These levels help ensure the exam can differentiate between students' understanding of basic, intermediate, and more advanced programming concepts. **Table 2** provides insight into how the exam distributes the points between different levels.

No.	Intended Learning Outcome (ILO)
1	Select and use appropriate primitive datatypes, including their preconceived behaviors.
2	Develop statements for data transformations over primitive datatypes using the appropriate operators, including typecasting of primitive types.
3	Express algorithmic solutions that use sequence and selection structures.
4	Express algorithmic solutions that use repetition structures.
5	Select and use appropriate linear data structures.
6	Appropriate use of operands order.
7	Select and use appropriate non-linear data structures.

Table 1. List of Intended Learning Outcomes (ILOs)

Skill Level	Points
Entry	10
Intermediate	20
Target	22
Total	**52**

Table 2. Exam Point distribution between skill levels

5.1 Entry Level Questions

Questions are designed to assess students' foundational knowledge and check their basic skills. They mostly focus on ensuring that students grasp the fundamentals before they delve into more advanced parts of the material. Questions at this level mostly involve straightforward problems like using conditional statements, operators, and type-checking. The entry-level questions in the sample exams were all multiple choice and none involved writing Python code, making this part out of scope for the research.

5.2 Intermediate Level Questions

This level aims to assess the student's ability to solve more complex problems and check their proficiency in using programming concepts beyond the basics. They usually involve a student working with linear structures and loops while also combining the knowledge from the entry level. **Figure 2** shows one of the typical student solutions for the intermediate-level exam question which asks to calculate the average value of the elements in the list provided in the function argument and return the number of elements that are higher than the calculated average. The code uses if-statements, for-loops, and type-checking to ensure that the correct type is passed on. Specifically, this question uses the ILOs ranging from 1 to 6 from the **Table 1**.

```
def find_tall(list_of_heights):
    error_msg = "Invalid Argument"
    if len(list_of_heights) == 0:
        print(error_msg)
        return

    for i in list_of_heights:
        if not isinstance(i, int):
            print(error_msg)
            return

    average = sum(list_of_heights)/len(list_of_heights)
    count_of_tall = 0
    for x in list_of_heights:
        if x > average:
            count_of_tall = count_of_tall+1

    return count_of_tall
```

Fig. 2. Typical Intermediate Level exercise

5.3 Target Level Questions

This level aims to test the student's ability to solve more advanced and challenging problems which require a thorough understanding of the material. Exam questions of this level involve the usage of nested loops, nested conditional statements, and more complex two or three-dimensional data structures like dictionaries, lists of tuples, etc. **Figure 3** shows a sample solution to one of the target level exercises. This exercise asks the student to process two lists of tuples into a dictionary with matching values. The concepts of such exercise include using nested for-loops, if-statements, type checking, two-dimensional data structures. Specifically, this question uses all the ILOs except No.6 from the **Table 1**. The comments shown in the figure are not required from the student.

```
def connect_key_value(key_data, value_data):
    # checking if any of the arguments is empty
    if len(key_data) == 0 or len(value_data) == 0:
        print("Invalid argument")
        return
    # checking if the arguments have the same length
    if len(key_data) != len(value_data):
        print("Invalid argument")
        return

    result = {}
    # traversing the first list (of tuples)
    for code_arg1, name_column in key_data:
        # traversing through the second list
        for code_arg2, asso_data in value_data:
            # If the arguments match, add it!
            if code_arg1 == code_arg2:
                result[name_column] = asso_data

    return result

# Example answer:
k=[(1,"Title"),(2,"Author"),(3, "Publisher")]
v=[(1,"Think Python"),(2,"Allen Downey"),
(3,"Green Tea Press")]
# The expected result of connect_key_value(k, v):
{"Title": "Think Python", "Author": "Allen Downey", "
     Publisher": "Green Tea Press"}
```

Fig. 3. Typical Target Level exercise

6 APPROACH OF AUTOMATION

The sample exams showed that the questions are simple functions that require a specific implementation. With the assumption that the exam questions will always be similar to the ones shown in **Figures 2 and 3**, the creation of a new automated grading tool can be approached.

6.1 Techniques

As seen in the provided sample data, the exams had 7 Intended Learning Outcomes from which the exercises were made (See Table 1). Certain actions and depth of knowledge can be extracted from these learning outcomes, which can later be used to assess the student's knowledge. This approach can be adapted in the automated grading tool by splitting all the exam questions into a set of ILOs. Then, multiple automatic grader agents can focus on specific ILOs and check for them in the student's answers. This can create structure and consistency that the automated grader needs to be accurate. Some already existing methods provide insight into how this can be implemented.

These ILOs can be graded using source code analysis and output testing, which were discussed in Section 4. Using the source code analysis approach, the students' source code can be converted into an Abstract Syntax Tree, allowing the grader tool to find whether the student solved the problem using any form of data structures, conditional statements, loops, operators, type instance checking, etc. All the mentioned ILOs in the **Table 1** can be checked by navigating the source code. For example, a learning outcome on which a student has to express a solution using repetition structures can be checked by determining if the code uses some looping mechanism like a 'for' or a 'while' loop. **Figure 4** shows an AST of a function with a simple `if` statement in Python for which it can easily be seen what is inside that `if` statement. If there is an algorithm that navigates the tree and visits each node, eventually it will find a 'For' node for which it would prove that the student has indeed used some repetition structure.

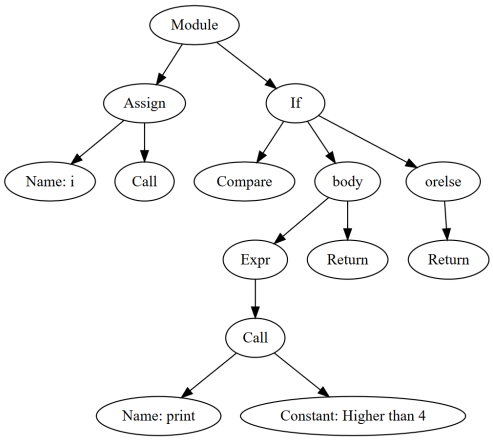

Fig. 4. AST for a simple function with an 'if' statement

Although the source code analysis can show whether the student used a specific structure, there is no certainty that the implementation is correct. To tackle this, the sample exam rubric has test cases that check whether the written answer matches the proper output for some input examples. This can be automated by using dynamic program testing to automatically check the output of the written code for a set of inputs.

6.2 Grading Process

Using dynamic and static code analysis, each exam question could be automated. The first steps will involve parsing the students' code using a tree visitor. Once all nodes are known, code checks can be made to see if the student used any of the required code implementations that are mentioned in the rubric. This can range from checking if the student has a loop to checking if the student has an if statement that checks for a specific clause. Numerous checks can be made using this, so the grader can choose how to approach this. Once the code checks are made, the grader can move to dynamic testing and check for the code outputs to see if the function returns the correct results for a given input. The inputs should be made in a way that tests if the implementation is correct and they vary for each question. These combined methods can thoroughly check each student's code and provide a result.

An example of practical usage of this approach can be explained for the code in **Figure 2** while also mentioning the ILOs from **Table 1** that each grading step checks. The checks were made to follow the ILOs made for this question, and also cover the grading criteria that exist in the rubric from the dataset. The whole exercise gives 5.5 points in total:

(1) The program checks if the code includes an `if` (or an `else`) statement which checks the length of the list. This is required because the exercise mentions the need to check if the list is not empty. This awards 0.5 points and checks ILOs 1 and 3.
(2) The program checks if the students' code includes a statement where they ensure that the elements inside the list are an instance of an integer. This can be done by using a built-in `isintance` check. This awards 0.5p and checks ILOs of 1, 2, and 3.
(3) The program checks if the code includes an `if` (or an `else`) statement which has both print and return statements inside the body. This is required because the exercise asks the student to print to the user and stop the program when an invalid input is given. This awards 0.5 points and checks ILO 3.
(4) The program uses a form of predetermined input cases to run on the code the student provided. They can range from any illegal input that could happen, to a variety of 'correct' input that checks if the logic of the code is fully implemented. If every case passes, the student is awarded 4 or 0 points and checks all ILOs 1 to 6.
(5) If the student did not get the full 4 points from the check above, new checks are run. This is done by output testing by giving values that may not break the code in case the student did not check for invalid input/empty lists. If these tests pass, they get 3.5 points. If the result is different by 1 value (this can happen if the student used >= instead of >), they get 2 points. This tests ILOs 3 to 6.

Step 5 in this list was taken from the rubric and slightly modified to fit this grading approach (more of that in Section 6.3). It needs to be stated that the output tests have to be accurate and not have an error in them. The grader is responsible for providing meaningful and thorough tests for each question.

6.3 Changes in Rubric

To make the automation approach work, slight changes must be made in the grading criteria. Sometimes the rubric asks to check whether the code has a correct implementation of a feature that can be made in multiple ways, thus making it unfeasible to automate using the selected approach. As an example, for the code in **Figure 2**, the rubric asks to check if the sum is calculated correctly (so that the average can work), however, a sum can be calculated in multiple ways (i.e. using a `sum()` function or a loop). The student can also use different variable names, so finding where the sum is

being calculated is too complex. It would make the grading tests too complicated and too specific, thus making them prone to error. The implementation can also be correct, but with a slight error (like using a >= instead of >), which shouldn't punish the student too hard. For these reasons, the rubric can be slightly adapted to be more output-focused. The grading steps provided in the Grading Process section are the same as in the rubric except for step 5. This step is slightly adapted to fit the automation approach by combining two criteria into output tests. It is also important to mention that if it is known that the exam is going to be graded automatically beforehand, the questions themselves can be created differently to avoid situations like this.

6.4 The Developed Prototype

Based on the steps described in the sections above, a prototype was developed as a proof-of-concept for this approach. An additional consideration made during the development is to allow the person writing the checks to provide feedback so that the student can know why they got that specific amount of points. After the checks are written, the analysis process can begin on the dataset. Once the tool finishes grading, it exports the results into an Excel file, where the grader can easily read and see them.

7 GRADING AN EXAM

To check whether the automated grader can grade with similar accuracy as the human grading, the prototype was used to grade the exam on the acquired dataset, discussed in the **Analyzing Sample Exams** section. The exam involved a total of 135 students. As mentioned in the previous sections, the first 4 questions did not include writing Python code, so these are discarded from this research.

7.1 Results of Manual Grading

When the exam took place, manual grading was conducted with the help of TAs. **Table 3** shows how many points were given for each exam question on average, the standard deviation (***s***), and the Median. For most exercises, the mean of the points is higher than 50% of the full points a student can get. The only two exceptions are questions 9 and 10, which show a lower mean than the other exam questions. This is because the last two questions jump from the **Intermediate** to **Target** level difficulty, so a fall in the average points is expected. The median value of 0 for Question 9 shows that 50% or more students got 0 points for that exercise, which shows that most students did not provide a valid answer. The total average point count combines into an average of 19,15 points, around 45% of the maximum amount of points a student can get (42 points). The standard deviation values show a good deviation from the mean, showing that the student's scores vary widely around the mean, which suggests that the exam grading criteria are well-balanced for this exam.

7.2 Results of Automated Grader

After writing automated tests for this exam, an automated grading was initiated. The outcome of this automation tool is shown in the **Table 4**, which shows the average points for each question, the standard deviation (*s*), and the median. For the average points for each exercise, the mean is looking to be distributed higher than 50% of the total points a student can get, with the only exception of the two harder **Target Level** questions, which have a lower mean.

Question	Mean	*s*	Median
5 (5p)	3,18	1,85	3,5
6 (5,5p)	3,62	2,12	4,5
7 (6p)	3,33	2,38	4,25
8 (3,5p)	2,10	1,34	2,5
9 (10p)	2,75	4,14	0
10 (12p)	4,17	4,64	3
Total	**19.15p**	-	-

Table 3. Results of Manual Grading

Question	Mean	*s*	Median
5 (5p)	3,52	1,53	3,5
6 (5,5p)	3,73	2,06	5
7 (6p)	3,50	2,15	4,5
8 (3,5p)	2,49	1,09	2,5
9 (10p)	2,85	4.24	0
10 (12p)	3,84	4,91	1
Total	**19.93p**	-	-

Table 4. Results of Automatic Grading

Sometimes the students' code cannot be automatically graded due to some errors which prevent the tool from running the code. For this, we need to calculate the success rate. The **Grader Success Rate** represents the percentage when the tool successfully analyzes the code by running the tests, and concludes how many points it should give. It also can succeed when the student does not provide any code, or the code is too short to grade. In that case, the point count is 0. The success rate is calculated by:

$$\text{Success Rate}(\%) = \left(\frac{\text{Successful Grades}}{\text{Total Number of Submissions}}\right) \times 100 \quad (1)$$

This formula is applied to each question to determine the grader's success rate.

Table 5 shows the success rate of running the tool on this exam. The average success rate of the grader is **76,49%**. For the remaining **23,51%** of the submissions, manual grading has to be conducted. **Table 5** also provides insight into the reasons why the tool fails the grading. The error analysis has shown that each question on average has around **27,17 Compile Errors** and very few **Runtime Errors (4,34)**. Two biggest compile errors are found in the students'

code: **Indentation** mismatch and **Syntax** errors. Another thing to note is that the **Target Level** questions also have a lower grader success rate, which can also be seen by the rise of the compile errors. This is to be expected since the exercises are longer and require more complicated code, thus it is more prone to human error.

Question	Grader Success Rate	Errors (Out of 135)	
		Compile	Runtime
5	76.86%	28	3
6	79.85%	21	6
7	85.07%	15	5
8	76.11%	26	6
9	68.65%	37	5
10	72.38%	36	1
Average	**76.49%**	**27.17**	**4.34**

Table 5. Error Analysis of Automatic Grader

7.3 Manual vs Automated

After analyzing both manual and automatic approaches separately, a comparison between the two results can be made. While looking at **Tables 3 and 4**, it can be seen that the results are similar, but not the same. One issue that was faced, is that the automatic grader did not successfully grade all exams (see **Table 5**), so to reflect that issue, the results of both automatic and manual analysis were paired, which resulted in the removal of some manual analysis results for this statistical analysis. Once the results are paired, they can be normalized and analyzed statistically.

Table 6 shows the normalized comparisons between the two grading methods. Here it can be seen that the manual analysis received an average normalized score of 0,5961, which shows that, on average, students scored 59,61% of the total possible points when graded manually. Automatic grading shows a mean of 0,5606, meaning that the students scored an average of 56,06% of the total possible points. Standard deviations for both approaches show significant variability in the scores, which shows the balance of the grading criteria for both methods and suggests that the consistency of the automatic grader remains similar to that of a manual approach. The medians show that **half** of the students scored up to 75% of the total possible points for manual grading, and up to 70% of the points for automatic grading.

Grading Method	Mean	s	Median
Manual	0,5961	0,414	0,75
Automatic	0,5606	0,401	0,7

Table 6. Comparison between grading methods **(Normalized)**

To determine whether the differences between manual and automatic grading are statistically significant, a paired sample t-test was conducted. The paired sample t-test aims to compare the scores from the manual and automatic grading methods for the same set of students. This test assesses whether the mean difference between these paired sets is significantly different. The resulting **t-value** is then used to determine the **p-value**, from which a conclusion can be made whether the mean difference between the two paired sets is significantly different or not.
The results of the paired sample t-test have shown a value of $t = 4,311$ and $p = 0,000019$. The p-value is lower than the conventional threshold of 0.05, indicating that the null hypothesis is rejected and that there is a statistically significant difference between the manual and automatic grading approaches. This implies that the differences observed in the mean scores are not due to randomness, but it indicates a genuine difference in the grading methods of this exam.

Additionally, to understand the correlation between the manual and automatic scores, a correlation analysis was performed. The Pearson correlation coefficient (r) measures the strength and direction of association between two continuous variables. It ranges from -1 to 1, showing either a positive or a negative correlation.
The correlation coefficient will measure how well the scores from manual grading align with those from automatic grading. The correlation coefficient (r) can be used to derive the **p-value**, which can reject or support the null hypothesis which states that there is no correlation between the two grading approaches.
The results showed a Pearson correlation coefficient $r = 0,878$ with an extremely low **p-value** of 4.13×10^{-194}. This high positive correlation shows a strong linear relationship between the two sets of scores. This means that students who scored high (or low) using manual grading, also tended to score high (or low) during automatic grading. The extremely low **p-value** rejects the null hypothesis and concludes that there is a statistically significant correlation between the two grading methods. The strong correlation supports the reliability of the automatic grading system, suggesting that it can consistently produce scores that align closely with manual grading.

8 CONCLUSION

In conclusion, the prototype of an automated grading system developed in this research shows promise as a reliable tool for grading first-year Python programming exams. It was developed using output testing and source code analysis and shows a positive correlation between the **accuracy** of a manual grading approach. However, the paired sample t-test suggests that the grading methods differ from manual grading, which can mean that either the tool is not fully developed to match the accuracy of the manual grading, or it could mean that it performed better than the manual grading. For this, further research is warranted.

9 DISCUSSION

This research demonstrates the potential for automating the grading of programming exams. Still, several areas warrant discussion and need future improvements and more research to enhance the

approach.

The developed tool currently only works for Python exams, which the UT does not have many of. A significant expansion could be made if the approach can be adapted to other programming languages used in different exams. I think that the approach could generally be similar to what was discussed in this research but adapted for the characteristics of that programming language. This would demonstrate the versatility and scalability of the system and show that it can be used in more exams.

The success rate of the grader (**Table 5**) could be further researched and improved. Sometimes the code cannot be compiled because of a simple mistake in the students' code, so a form of error correction research could be conducted to analyze the issue further. This could also be solved by making sure that the students submit the code that correctly compiles during the exam. This would highly improve the success rate of the automatic grading, which would lower the needed staff time for manually checking those answers.

Using an automated tool creates some ethical implications that should be addressed. This means the system must be designed to eliminate any bias and ensure transparency in its results. It also should protect the user data and have sufficient privacy measures in place to safeguard the students' sensitive information. Additionally, establishing clear accountability mechanisms is essential so that the students can easily dispute and address their grades in an exam review session if needed.

It is also important to consider the potential for human error in the manual grading dataset that was used in this research. The research assumed that the manual grading matches how the exam needs to be graded. However, the dataset may include some errors that could contribute to the discrepancies observed in the statistical analysis between the automated and manual grading. This could even introduce the possibility that the automation tool produced better grading results than the results seen in the dataset. A bigger dataset and a more thorough analysis of manual grading practices and potential errors could provide deeper insights and help refine the automated grading system.

REFERENCES

[1] Tracy Camp, W Richards Adrion, Betsy Bizot, Susan Davidson, Mary Hall, Susanne Hambrusch, Ellen Walker, and Stuart Zweben. 2017. Generation CS: the growth of computer science. *ACM Inroads* 8, 2 (2017), 44–50.

[2] Yingjun Cao, Leo Porter, Soohyun Nam Liao, and Rick Ord. 2019. Paper or online? A comparison of exam grading techniques. In *Proceedings of the 2019 ACM conference on innovation and technology in computer science education*. 99–104.

[3] José Marílio Cardoso, João Pascoal Faria, and Bruno Lima. 2017. Automatic Assessment of Programming Assignments to Enable New Educational Paradigms. https://api.semanticscholar.org/CorpusID:65001752

[4] Carl Dreher, Torsten Reiners, and Heinz Dreher. 2011. Investigating Factors Affecting the Uptake of Automated Assessment Technology. *Journal of Information Technology Education: Research* 10, 1 (January 2011), 161–181. https://www.learntechlib.org/p/111517

[5] Hans Fangohr, Neil O'Brien, Anil Prabhakar, and Arti Kashyap. 2015. Teaching Python programming with automatic assessment and feedback provision. *arXiv preprint arXiv:1509.03556* (2015).

[6] Chris Greensted. 2014. Intended learning outcomes. *EFMD Global Focus* 8, 1 (2014), 20–25.

[7] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research*. 86–93.

[8] Maria Kallia. 2017. Assessment in computer science courses: A literature review. *Royal Society* (2017), 1–60.

[9] Joseph Klein and Liatukas El. 2003. Impairment of teacher efficiency during extended sessions of test correction. *European Journal of Teacher Education* 26 (2003), 379 – 392. https://api.semanticscholar.org/CorpusID:145559783

[10] Rishabh Kothari, Burhanuddin Rangwala, and Kush Patel. 2023. Automatic Subjective Answer Grading Software Using Machine Learning. *2023 7th International Conference on Trends in Electronics and Informatics (ICOEI)* (2023), 1006–1012. https://api.semanticscholar.org/CorpusID:258869775

[11] David R Krathwohl. 2002. A revision of Bloom's taxonomy: An overview. *Theory into practice* 41, 4 (2002), 212–218.

[12] Stephan Krusche and Andreas Seitz. 2018. Artemis: An automatic assessment management system for interactive learning. In *Proceedings of the 49th ACM technical symposium on computer science education*. 284–289.

[13] Emily Marshman, Alexandru Maries, Ryan T Sayer, Charles Henderson, Edit Yerushalmi, and Chandralekha Singh. 2020. Physics postgraduate teaching assistants' grading approaches: Conflicting goals and practices. *European Journal of Physics* 41, 5 (2020), 055701.

[14] Marcus Messer, Neil CC Brown, Michael Kölling, and Miaojing Shi. 2024. Automated grading and feedback tools for programming education: A systematic review. *ACM Transactions on Computing Education* 24, 1 (2024), 1–43.

[15] Nakka Narmada and Peeta Basa Pati. 2023. Autograding of Programming Skills. *2023 IEEE 8th International Conference for Convergence in Technology (I2CT)* (2023), 1–6. https://api.semanticscholar.org/CorpusID:258870000

[16] National Academies of Sciences, Division on Engineering, Physical Sciences, Computer Science, Telecommunications Board, Policy, Global Affairs, Board on Higher Education, and Committee on the Growth of Computer Science Undergraduate Enrollments. 2018. *Assessing and responding to the growth of computer science undergraduate enrollments*. National Academies Press.

[17] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. 2022. Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education (TOCE)* 22, 3 (2022), 1–40.

[18] Arthur Rump, Ansgar Fehnker, and Angelika Mader. 2021. Automated assessment of learning objectives in programming assignments. In *International Conference on Intelligent Tutoring Systems*. Springer, 299–309.

[19] Chad Sharp, Jelle van Assema, Brian Yu, Kareem Zidane, and David J. Malan. 2020. An Open-Source, API-Based Framework for Assessing the Correctness of Code in CS50. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 487–492. https://doi.org/10.1145/3341525.3387417

[20] Zhikai Wang and Lei Xu. 2019. Grading programs based on hybrid analysis. In *Web Information Systems and Applications: 16th International Conference, WISA 2019, Qingdao, China, September 20-22, 2019, Proceedings 16*. Springer, 626–637.

[21] Susilo Veri Yulianto and Inggriani Liem. 2014. Automatic grader for programming assignment using source code analyzer. In *2014 International Conference on Data and Software Engineering (ICODSE)*. IEEE, 1–4.

[22] Francisco A Zampirolli, Joao M Borovina Josko, Mirtha LF Venero, Guiou Kobayashi, Francisco J Fraga, Denise Goya, and Heitor R Savegnago. 2021. An experience of automated assessment in a large-scale introduction programming course. *Computer Applications in Engineering Education* 29, 5 (2021), 1284–1299.

A DEVELOPED PROTOTYPE

The source code of the developed prototype can be found on GitHub using this link: https://github.com/Kristupasc/AutoGrader