Loughborough University

This item was submitted to Loughborough's Research Repository by the author.
Items in Figshare are protected by copyright, with all rights reserved, unless otherwise indicated.

Improving marking efficiency for longer programming solutions based on a semi-automated assessment approach

PLEASE CITE THE PUBLISHED VERSION

https://doi.org/10.1002/cae.22094

PUBLISHER

© Wiley

VERSION

AM (Accepted Manuscript)

PUBLISHER STATEMENT

This is the peer reviewed version of the following article: BUYRUKOGLU, S., BATMAZ, F. and LOCK, R., 2019. Improving marking efficiency for longer programming solutions based on a semi-automated assessment approach. Computer Applications in Engineering Education, 27(3), pp. 733-743, which has been published in final form at https://doi.org/10.1002/cae.22094. This article may be used for non-commercial purposes in accordance with Wiley Terms and Conditions for Use of Self-Archived Versions.

LICENCE

CC BY-NC-ND 4.0

REPOSITORY RECORD

Buyrukoglu, Selim, Firat Batmaz, and Russell Lock. 2019. “Improving Marking Efficiency for Longer Programming Solutions Based on a Semi-automated Assessment Approach”. figshare. https://hdl.handle.net/2134/35734.

Improving marking efficiency for longer programming solutions based on a semi-automated assessment approach

Selim Buyrukoglu, Firat Batmaz, Russell Lock

Computer Science, Loughborough University, Loughborough, UK

Abstract

In recent years, many students in higher education have begun to learn programming languages. In doing so they will complete a variety of programming tasks of varying degrees of complexity. The students need to get consistent and personalised feedback to develop their programming skills. Human markers can provide personalised feedback using traditional manual approaches to assessment, but they may provide inconsistent feedback (especially for long programming solutions) since marking the programming solutions of multiple students can represent a significant workload for them. While full-automated assessment systems are the best to provide consistent feedback, they may not provide sufficiently personalised feedback for novice programmers. This study develops a novel semi-automated assessment approach in order to improve efficiency of human marker in the marking process and increase consistency of feedback (for both short and long programming solutions). It advocates the reuse of human marker's comments for similar code snippets, defined as segmented marking in this study. New full and partial marking models are developed based on segmented marking and they are tested by expert markers. The findings show that the two models are similar in efficiency, but that a partial marking approach potentially offers an improved efficiency for longer programming solutions. Such a finding has significant potential to reduce time spent on marking throughout the sector, which would have significant impact on both resourcing and timeliness of feedback.

Keywords

novice programmers, segmented marking, full and partial marking, marker workload

1. INTRODUCTION

In recent years, an increasing number of students have started to learn to program, with programming a core part of many higher education courses [27]. At this stage of learning, these students are considered as novice programmers, and as such, they need to receive consistent and personalised feedback to hone their programming skills effectively [3]. As a pre-cursor to this study, students' (novice programmers) lab practice programming solutions were observed in terms of their structures. These were first-year students undertaking the 'Introduction to Programming' module at Loughborough University. The observations made on students' (novice programmers) programming solutions, included that student code at this level mostly consisted of similar code pieces (e.g. loops, control statements, etc.). In addition, a review of prominent computer-based assessment marking techniques in terms of their marking approaches, presented in Section 2, which reveal that existing approaches focus on whole programming solutions in order to provide feedback, rather than on individual code pieces. In this study, a whole programming solution refers to a code script, while a code piece refers to a code segment. Table 1 shows a code script that contains two code segments.

TABLE 1 A code script

Code Segment – I	`import math #Imports the math module`
	`print "Height/Radius Surface Area Volume"`
	`print "=================================="`

Code Segment – II	`for i in range(2, 11):`
	`  surfArea=(pi*i)*(i+(math.sqrt(i**2+i**2)))`
	`  volume=((pi*i**2*i)/3)`
	`  print "%8d %20.2f%15.2f"%(i,surfArea,volume)`

This study focuses on marking each code segment in a code script, a novel approach we call segmented marking. In such a manner, the human marker focuses on each code segment,

and a more detailed feedback can be generated. The proposed approach also advocates the re-use of human markers' comments for semantically similar code segments among code scripts based on segmented marking to reduce human markers' workload. In this sense, the role of the human marker can be used more efficiently in the marking process based on static assessment (during a static analysis, the code is examined and evaluated without running the program), since they may dedicate more focus to each code segment in the code scripts, and students may thus obtain personalised feedback. In contrast, no human markers are used in dynamic assessment (during a dynamic analysis, each student's program code is executed and then the result is checked to ascertain the correctness of the program), which may prevent students from obtaining sufficiently personalised feedback.

Segmented marking can also be used to reduce the marking workload and provide more consistent feedback, which is a significant gain for computer-based assessment systems. More detailed information on segmented marking is presented in Section 3. A feasibility study was carried out on segmented marking, and the participants' thoughts were captured using a questionnaire [23]. More detailed information can be obtained from [23]. Normally, the marker sees the whole code script on the screen without scrolling down if code script is short. In this study, this case is referred to as "full marking" (the traditional way of marking). However, in the feasibility study conducted at that start of this study on segmented marking, we realised that in cases where students provide long code scripts, markers see part of the code script, rather than the entire code script on screen. Note that, in this research, a code script is considered "long" if each code segment in the code script cannot be displayed on a single screen. Bearing this in mind, if participants only see (on the screen) a code segment instead of a code script in the marking process, they may be encouraged to provide more detailed feedback. This case is referred to as "partial marking" in this study. This can thus be more beneficial for human

markers, especially in marking long code scripts, since the proposed marking approach facilitates improved marking process efficiency and reduced marking workload.

The feasibility study also contributes towards identifying the two user interface design requirements for the marking tool. Requirement 1 concerns the provision of feedback based on segmented marking, and Requirement 2 involves the re-use of human markers' comment(s) for semantically similar code segments. As such, full and partial marking models can be developed to provide feedback for both short and long code scripts, and increase the efficiency of human markers based on these requirements. Upon their development, the study compares their efficiency in terms of marking workload and feedback consistency. Thus, this study intends to answer the following research questions:

- Which marking process (full or partial) model enables a more efficient marking process?
- Which marking process (full or partial) model enables a more significant reduction in markers' workload?

The rest of this paper is structured as follows. A literature review of prominent studies is provided, and this is followed by discussions on model development of full and partial marking. Subsequently, the results from the study are presented and discussed in the evaluation section. Specifically, the evaluation answers the research questions highlighted above. Finally, the conclusion is outlined.

2. LITERATURE REVIEW

This section discusses why semi-automated assessment approach should be used to provide feedback for novice programmers in Section 2.1. In addition, existing semi-automated assessment systems are discussed with their specific advantages and disadvantages in Section 2.2. In Section 2.3, common drawbacks of the existing systems are interpreted.

2.1. Why semi-automated assessment?

Programming solutions can be assessed using three types of marking; manual, automated and semi-automated marking [15]. In addition, assessment systems use a static and/or dynamic assessment approach [14]. Manual marking (traditional method of marking) is the most effective way to provide personalised feedback. In manual marking although computer tools may be used to display code the tool does not assist in the marking process, relying on the skills of the marker alone. The human marker may potentially provide inconsistent feedback, especially when programming solutions are longer [10], and this is a disadvantage of manual marking. In addition, manual marking represents a significant workload for human markers, especially for large student numbers. On the other hand, automated marking is considered to be the best approach to provide consistent feedback, and constitutes less workload compared to other assessment approaches [1]. However automated marking systems generally provide less personalised and detailed feedback. Semi-automated marking systems use dynamic and static assessment, providing the human marker with technological assistance in analyzing the code scripts. As such, students may get personalised and more detailed feedback. Thus, this study aims to develop a semi-automated assessment system for novice programmers. In current semi-automated marking systems, dynamic assessment is initially used, and a human marker then provides feedback based on static assessment [5]. The use of a human marker is important in ensuring the provision of personalised and detailed feedback. However, static assessment constitutes a substantial workload for human markers, and feedback consistency may potentially reduce if the number of programming solutions to be assessed is large. In light of this, this study intends to develop two novel semi-automated marking approaches to improve the efficiency of the marking process and reduce human markers' workload, while providing personalised and consistent feedback.

2.2. Existing semi-automated systems

Many computer based assessment marking systems have been developed to provide feedback for code scripts supporting different ways of marking. As this research focuses on semi-automated assessment systems, this section presents a discussion of prominent semi-automated marking systems in terms of their efficiency in the marking process and marking workload for human markers.

Most semi-automated assessment marking systems share a common approach, which is that they initially intend to provide feedback based on dynamic assessment (automated assessment), while reducing the marking workload for human markers and improving the feedback consistency [21, 8, 30]. However, a human marker then extends the feedback based on static assessment (manual assessment) to provide detailed and personalised feedback, at the expense of increasing the workload of human markers (especially for long code scripts). Such systems provide feedback on the correctness of (both short and long) code scripts based on dynamic assessment [29, 17, 13, 4, 24, 16, 6, 25, 5, 18]. In this sense, marking systems compare the output of students' code scripts and the model answers to provide feedback on the correctness of the students' code scripts. In addition, a human marker extends the feedback and provides information about code structures etc. [17, 6]. Furthermore, static assessment allows human markers to provide feedback on the style of a code script [17, 6, 24, 29]. Therefore, feedback consists of information about indentation, modularity and meaningful variables. Moreover, [22, 4] enable the human marker to fix syntax errors in order to provide feedback if a student's code script fails to compile and returns errors. In such a case, the systems provide feedback based on dynamic assessment. In all other cases, the system provides feedback for code scripts based on dynamic assessment. The rest of this section provides detailed information on representative semi-automated approaches.

Work in this area has evolved over many years. In early work on [12] (TRY), several testing approaches were evaluated. Initially, the student's and model answer's output are character-

by-character matched, which is a process that is part of the dynamic assessment. If they match, the student's answer is correctly accepted by the system. In the second approach, the student's answer is normalised before matching with the model answer, where the redundant parts of student's answer are removed. The latter approach is one in which students must print the output based on the marker's definition in the question. Additionally, students' code scripts must pass all tests to be considered successful. The TRY system then provides feedback on which scripts pass or fail. Finally, the marker can provide comments on the each student's answer based on the static assessment. In this system, the second approach can be especially helpful for markers since the marker does need to provide many model answers. Furthermore, the number of similar code script can increase after the applying the second approach which may increase feedback consistency. However, the TRY system has some significant drawbacks including.

- The output of students' programming code and the model answer are character- by-character matched in this approach. In this case, although the student answer may be correct, it may be different in terms of syntax/format to the model answers. In this case, the student answer is deemed incorrect. Thus, in such cases, the student could receive incorrect feedback.
- The addition of further comments by the marker to extend the feedback provided by the system could be time consuming. Furthermore, the marker may also provide inconsistent comments.

The system proposed by [7] (Sakai) can compile, test, execute and score a student's program without human intervention. It is developed based on dynamic and static assessments. If the output is incorrect, feedback is then given to the student regarding his/her produced output and the expected one by the human marker. If the output is correct, the automatic marker

is used to check the student's code script for equality. Equality is defined as the number of lines between the student's and the model answer that are equal. Additionally, the student code script is normalised before checking the equality between students' answers and the model answer. Normalisation involves the removal of redundant parts from code scripts. Hence, equality is increased between code segments. Marks are then specified for the students' code scripts by the system if the student's and model answers are equal. If they are not equal, the student answer is marked by the human marker. Furthermore, both the system and the human marker provide comments on specific parts of code scripts such as the basic structure, input, computation and output. Manual marking also has an advantage in Sakai over automatic marking in that human markers could provide richer feedback. In addition, normalisation of code scripts can be very helpful in reducing the markers' workload. This is because after normalisation, the number of semantically equal code scripts may increase and the system can then provide feedback based on dynamic assessment using the model answers. Also, the similarity between code scripts may increase and the marker can subsequently provide more consistent feedback based on the static assessment. However, Sakai's approach has a few drawbacks such as:

- Equality between students' programming scripts, even on shorter solutions is problematic . In this case, the marker must provide all possible model answers to automatically mark students' programming codes. This could involve unnecessary repetition for the marker, since students' solutions can vary slightly from one another, and as such, model answers need to cover each student's solution.
- Markers might provide inconsistent feedback for similar parts of different programming codes (especially for code scripts, which do not match the model answers).

- Students do not get detailed feedback if the output is incorrect. In such cases, Sakai only provides information about the expected output as a feedback. In this sense, the feedback cannot be helpful for the students.

[28] (ALOHA) divides a given assessment into smaller parts. The marker can concentrate on a single aspect of the work instead of giving a general grade for the final work. ALOHA provides an online rubric that each marker must fill in. An assessment rubric is a guide listing specific criteria for grading smaller parts in the code script. The marker can choose a ready-made comment to be added for the student's feedback. ALOHA provides only text-based feedback and utilises formative and static assessments. In this sense, the use of a rubric can help the marker if it is created in great detail. That is, if the marker makes comments on smaller parts using the detailed rubric, they can provide strong and comprehensive feedback for novice programmers. However, ALOHA has some drawbacks including:

- The marker should provide feedback according to a rubric. The marker cannot make comments outside of the rubric. If a rubric is created superficially, the marker may not be able to provide helpful comments.
- The marker must mark all programming code respectively. It is a workload for the marker and could lead to inconsistent feedback.

2.3. Discussion on disadvantages of existing systems

In literature, none of the existing systems (including representative approaches [12, 7, 28]) focuses on automation of static assessment which is the gap in this area. In addition, the systems of [12] and [7] also apply a normalisation approach to increase the similarity between students' solutions and model answers as described in previous section. However, in their current form the normalisation approaches applied does not completely solve the problems faced Furthermore, human marker's comments could be re-used (automatically) to provide feedback

for semantically similar code segments based on static assessment. Thus, the marking system reduces the workload and accelerates the marking process. In addition, the efficiency of the human markers in the marking process can potentially increase, since they save marking time in comparison to existing marking systems. In this sense, the proposed assessment approach can be considered quite different from existing approaches.

Human markers aim to provide the same comments for each solution repeatedly (over dynamic assessment) which is monotonous and time-consuming for them. Even if they provide similar feedback, consistency of feedback can be reduced, if only through fatigue. In addition, human markers may minimize the feedback to increase marking speed, preventing students from getting personalised and sufficient feedback if the number of solution is high and the solutions are long. Therefore, this study advocates a novel way of marking which increased automation in the marking process based on static assessment, in contrast to all semi-automated assessment marking systems.

The following section describes the proposed segmented marking approach, developed based on semi-automated assessment, which aims to provide solutions for the disadvantages described above. It then describes the development of the full and partial marking models.

3. DEVELOPMENT OF MARKING MODELS

In this study, the marking process utilises the segmented marking technique based on semi-automation. The marking process of the semi-automated approach (advocating segmented marking) is briefly described in this section before describing the development of the both full and partial marking process models. This is done because although these models support different methods of marking, both advocate the use of segmented marking. In addition, the partial marking model is introduced to improve the efficiency of the marking process, especially for long code scripts.

3.1. Overview of the semi-automated marking approach

This section presents the semi-automated marking approach. Figure 1 illustrates an overview of the marking approach which contains four processes namely; segmentation, codifying, grouping and marking process.

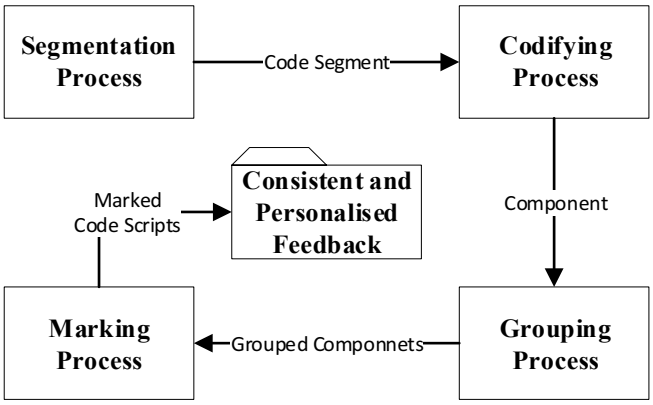

FIGURE 1 Overview of the marking approach

In the **segmentation process**, code scripts are parsed to generate code segments. Each code segment then undergoes a **codifying process,** where it is normalised by applying generic rules to increase the similarity between code segments. As can be seen from Figure 1, each code segment refers to a component after the codifying process. A similarity measurement technique is then applied to group components in the **grouping process**. If the similarity between two components are more than a certain threshold, they are put into the same group. A string matching technique was used in this research in order to make line-based comparisons to other lines in components. Variable names, blank lines, comments explaining code lines and print messages are not considered in the similarity measurement process after applying the generic rules in codifying process. This allows highly similar components to be placed into the same group and consistent feedback to be provided to all within that group. In addition, a threshold value needs to be specified when creating groups according to the similarity measurement results. Without a threshold only exact matches (which are unlikely in longer assignments) would be placed in groups. This would result in increased marker's workload due to the increase in the number of groups. There is an important relationship between accuracy and threshold value. If a low threshold value is applied the groups will contain less semantically

equivalent members, lowering the accuracy of potential feedback for that group, but in doing so reducing the number of groups (decreasing the workload of the marker). An appropriate threshold therefore needs to be established, however this is context specific. For this study the threshold value was established as 90%, (the methodology for testing values is outside the scope of this paper due to length limitations). Note that the marker cannot mark a component directly, as these have been normalised beyond the point a marker could easily interpret them. The human marker's comment is then re-used to automatically mark the semantically similar code segments within the same group. Therefore, the marking technique reduces the human marker's workload in terms of marking time, while providing personalised and consistent feedback.

One of the limitations of the proposed approach is that students can use different variable names. String match is used to group components in the similarity measurement technique. For this research, it is worth mentioning that students generally use the same variable names and print messages and their solutions provide mostly same results since the asked question highlighted the print messages and variable names that should be used in the solutions. Due to these reasons, the way the question is written effects the outcome. However, students typically may use different variable names and print messages. Although the variable names and print messages used are equivalent semantically, they do not match and cannot be placed into the same group, which results in increasing the number of groups. These limitations could be solved using a code writing editor enabling drag and drop code parts like Scratch programming [20]. In this sense, each student can use the same variable names and print messages which may reduce the number of groups. In future research, a code writing editor (similar to Scratch) will be developed and used to capture novice programmers' solutions. Then, the solutions will be assessed based on the segmented marking.

3.2. Overview of the full marking process model

This section presents the development of the full-marking process model. It is developed based on the user-interface requirements for the marking tool, which were obtained from the feasibility study of segmented marking, as highlighted in Section 1. Figure 2 illustrates the full-marking process model. In full marking, the code script is displayed to the marker allowing them to see how the code segments are linked. As such, the human marker should select a code script and then chooses a code segment to provide comment for it. In addition, the human marker should mark each code segment from the selected code script (requirement 1 in Section 1). Then, human marker's comment is re-used for semantically similar code segments within same group (requirement 2 in Section 1), which refers to automated marking. From the markers perspective the more code scripts they mark, the greater the number of code segments within the remaining scripts will already have been commented on, thanks to the comment reuse.

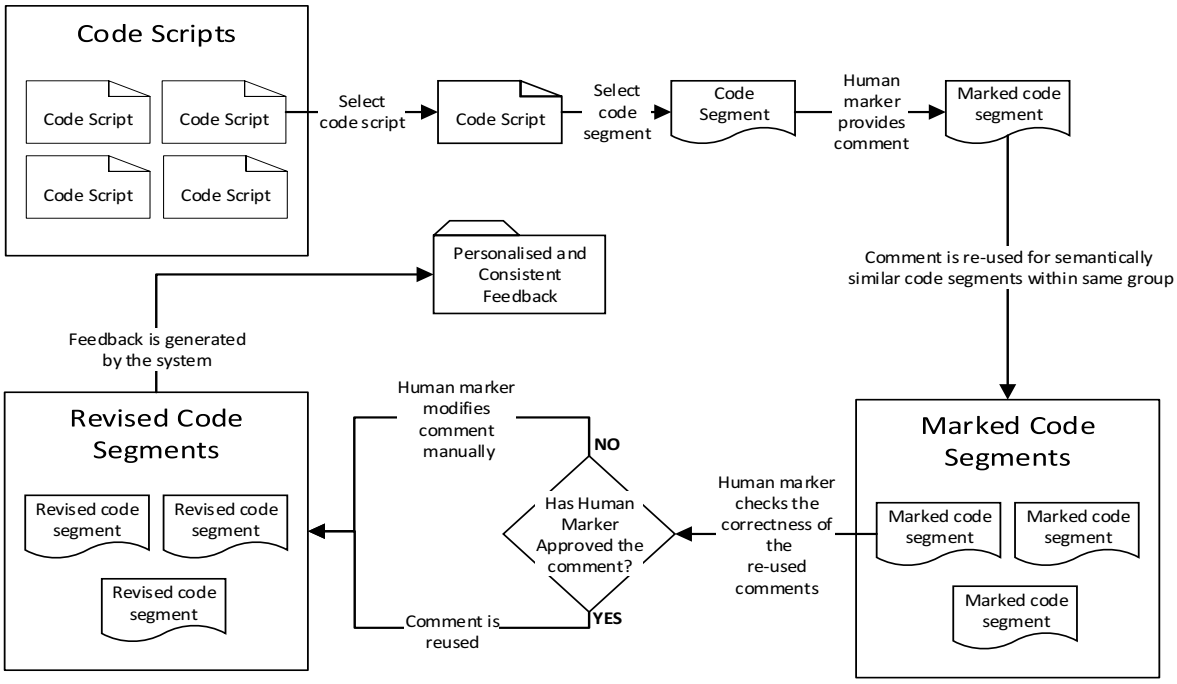

FIGURE 2 Overview of full marking model

The automatically marked code segments need to be reviewed by the human marker, since they do not automatically see the marked code segments. As such, in some cases, the re-used comments could prove unsuitable for certain code segments. In other words, this review

process guarantees that the human marker checks the automatically marked code segments in order to ensure the correctness of the comments, making adjustments as necessary. In addition, the modified comments can be re-used for semantically similar code segments within the same group to accelerate the review process if the human marker needs to re-use them more widely. By doing this, checking/modifying of applied comments can be performed faster than just writing them manually one by one. After this process, the code segment is considered as reviewed according to the full marking model. Thus, each code segment is considered marked after the human marker reviews the automatically marked code segments. This process ensures that the human marker's workload is reduced, and that personalised and consistent feedback is generated based on segmented marking. However, the efficiency of the marking process may reduce if a code script is long, since such code script cannot be fully displayed on the screen, and the human marker cannot link easily between code segments. As such, the full marking approach is considered as a more appropriate option for providing feedback on short code scripts.

3.3. Overview of the partial marking process model

This section introduces the partial marking model (Figure 3 illustrates the partial-marking model) and outlines the development of the full marking model based on the user interface design requirements.

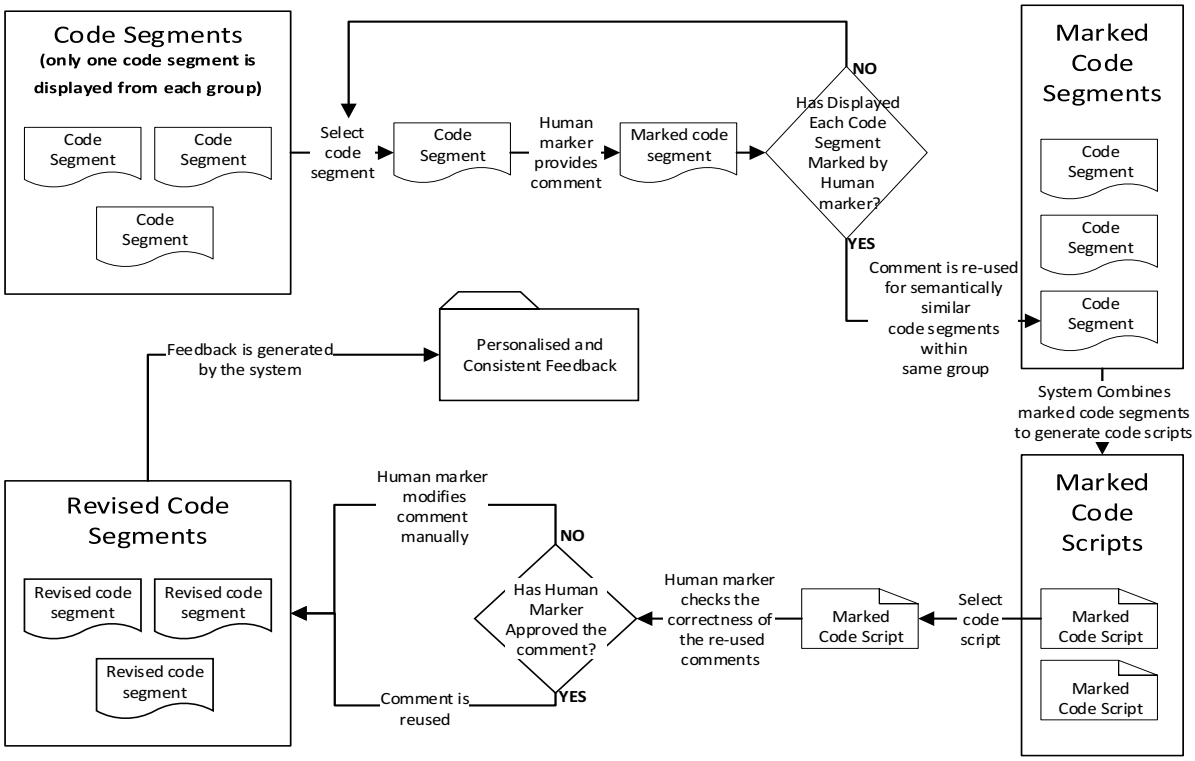

FIGURE 3 Overview of the partial marking model

The partial-marking process utilises a new method of marking in which a code segment is displayed instead of the whole code script, since code scripts can be long, in which case they cannot be fully displayed on a screen. In such cases, the human marker may not be able to focus on code script to provide detailed feedback, and the efficiency of the marking process may also reduce. In contrast, markers can dedicate more focus on individual code segments in the partial marking process than in the traditional way of marking (full-marking model).

In the new marking approach, the human marker needs to only see a code segment from each group. The groups are created after the grouping process, which is described in Section 3. Thus, the human marker initially provides comments for each displayed code segment (which refers to manual marking and requirement 1 highlighted in Section 1). This manual marking process is repeated until each displayed code segment is commented on. Subsequently, the human marker's comments are re-used for semantically similar code segments within the same group (which refers to automated marking and requirement 2). The marking tool then combines

the comments to generate feedback, and subsequently displays them to the human marker. The human marker then reviews the generated comments, since the automatically commented code segment can be inapplicable. Finally, personalised and consistent feedback is generated for each student's code script, and the efficiency of the marking process improves since the human marker focuses only on certain code segments rather than all code segments in a code script. In this sense, the partial marking approach can be an appropriate option to provide consistent feedback for long code scripts, where it can reduce the marking workload while improving the efficiency of the marking process.

4. METHODOLOGY

Participants with programming solution marking experience were used in this experiment. In order to evaluate the two techniques (full and partial marking) developed, an experiment was conducted. The participants did not have any information on the approaches or how the marking tools worked. Therefore, a marking guide, questions and model answers were presented to each participant before the experiment. Moreover, a brief demonstration of the tools was provided before the experiment, so that the participants could use the marking tools effectively. Each of the participants then provided feedback using the full-marking and partial-marking tools. The participants started the experiment by using the full marking tool. This is due to the fact that the full marking tool displays the code script to participants, and as such, it is more similar to a traditional approach of marking than the partial marking tool. Subsequently, the participants used the partial marking tool. The two semi-automated approaches were not evaluated against a manual approach. The reasons for this were largely that of the pilot nature of the tools being developed, and the limited evaluation sample sizes discussed in the next section. Given markers existing familiarity with manual approaches it was felt that a fair comparison of manual / semi-automated techniques could not be made without a much longer

longitudinal study to overcome the built-in familiarity, potential bias, and marking efficiency of experienced academics.

4.1. Participants and questions

[2] state that a small sample size (5-20) is sufficient for usability assessment and [11] highlights that researchers can collect valuable results using a minimum of 5 participants (subject experts) in a main experiment. Furthermore, [26] used 5 subject experts in order to provide feedback for programming solutions in their main experiment. Given the difficulties associated with finding subject expert willing to spend sufficient time to run the experiments the sample size is fairly low. More than 80 requests were sent to different subject experts to join this experiment, however only 5 subject experts accepted to participate. It may be the case that the workload on markers is so great that finding the time to trial new technologies to reduce it is difficult. As a result, eight participants were used in this (main) experiment. The experiment employed five subject experts, and three markers non-subject experts. If more subject experts had accepted to join this experiment, it could have provided a more accurate result. In this sense, 8 participants were used in total. The existing literature lacks clarity and detail regarding a subject expert's experience (in terms of how many years of experience are required). In this study, subject experts were required to have at least ten years of marking experience.

The participants were required to provide feedback for 30 code scripts using the full and partial marking tools, respectively. The 30 code scripts consist of 50 code segments in total. In addition, 17 groups were crated based on the grouping process. Finally, all participants provided feedback for 480 (240 * 2) code scripts in total using both tools, which means that 800 code segments (400 * 2) were marked in total. The marking sample originally consisted of 165 code scripts. However, to accommodate the busy schedules of expert markers, the number of code scripts were reduced from 165 to 30. While a larger number of code scripts may have

improved the accuracy of the results pragmatism had to applied when considering the time available to markers for the experiment. Furthermore, code segment structure, programming errors, code layout and comments were important factors in choosing code scripts for this experiment.

Three kinds of programming problems were asked to students in the class test. The problems required students to make use of code segment types such as control (if, else-if, else) and loop statements.

4.2. Measurements

The measurements taken in this experiment were the time spent and the feedback consistency of both the full and partial marking approaches, in order to learn which approach is more efficient in the marking process. In this sense, the initial and revised marking time for semantically similar code segments need to be measured. Participants check the correctness of the re-used (automatically generated) comments to provide appropriate feedback. In this context, the checking time for each code segment is referred to as the revised-marking time in this study. Furthermore, the number of automatically commented (re-used comments) code segments and the number of modified (revised) code segments are recorded in the experiment in order to measure the feedback consistency. Thus, the two marking models can be compared based on the measurements to learn which of them offers a more efficient marking process. Since the participants had varying levels of marking experience, the feedback quality provided varied between each participant. Therefore, this research did not focus on the feedback quality.

4.3. Data collection

In this experiment, data was collected through two methods. Initially, the participants commented on students' code scripts through the marking tools. The marking time spent on code segments was measured and recorded. Subsequently, the participants' thoughts were captured on segmented marking, as well as on the full and partial marking approaches.

5. EXPERIMENTAL EVALUATION AND RESULTS

This section presents and discusses the experimental results based on measurements of the marking time spent and the feedback consistency, to decide which marking approach offers a more efficient marking process.

5.1. Comparison of marking times (workload) based on two marking techniques

The marking time indicates the average marking time spent on code segments. The marking time is important for understanding the efficiency of the marking tools based on the semi-automated assessment approaches [10]. Table 2 presents univariate descriptive statistics for the time the participants spent marking 400 code segments using the full and partial marking approaches. It can be seen that the average time spent marking the 400 code segments was 1704 seconds for the full marking tool and 1262 seconds for the partial marking tool.

TABLE 2 Univariate descriptive statistics for the task times (n = 8)

	Min	Max	Mean	SD
Full-Marking approach (seconds)	1380	2160	1704	282.13
Partial-Marking approach (seconds)	1020	1620	1262	212.85

The mean time scores of the full and partial marking approaches suggest that the participants saved more time using the partial marking approach. Markers can therefore dedicate more focus on providing more detailed feedback on code segments in the partial-marking approach. This is particularly true for marking long code scripts, since the human marker sees only a code segment on the screen.

5.2. Comparison of feedback consistency based on two marking techniques

Markers may edit automatically marked code segments as highlighted in Section 3.2 and 3.3. The numbers of edited and automatically marked code segments are illustrated in Table 3 a, b, c, d, e, f, g, and h related to each participant separately. Each of them refers to a contingency table. A contingency table is used in statistics to provide a tabular summary of categorical data,

where the cells in the table are the number of occasions that a particular combination of variables occurs together in a set of data. The relationship between variables in a contingency table are often investigated using Chi-squared tests [9]. The simplest contingency table with two variables has two levels for each of the variables. In in Table 3(a-h), 'P' refers to the participant id, 'A' to the number of automatically marked code segments, 'M' to the number of modified code segments in the revision process, 'FM' to the full-marking tool, 'PM' to the partial marking tool, and 'T' to the total.

TABLE 3a Contingency tables of participant 1

P1	A	M	T
FM	31	2	33
PM	27	6	33
T	58	8	66

TABLE 3b Contingency tables of participant 2

P2	A	M	T
FM	29	4	33
PM	29	4	33
T	58	8	66

TABLE 3c Contingency tables of participant 3

P3	A	M	T
FM	30	3	33
PM	30	3	33
T	60	6	66

TABLE 3d Contingency tables of participant 4

P4	A	M	T
FM	29	4	33
PM	25	8	33
T	54	11	66

TABLE 3e Contingency tables of participant 5

P5	A	M	T
FM	27	6	33
PM	22	11	33
T	49	17	66

TABLE 3f Contingency tables of participant 6

P6	A	M	T
FM	31	2	33
PM	26	7	33
T	57	9	66

TABLE 3g Contingency tables of participant 7

P7	A	M	T
FM	29	4	33
PM	26	7	33
T	55	11	66

TABLE 3h Contingency tables of participant 8

P8	A	M	T
FM	29	4	33
PM	24	9	33
T	53	13	66

As can be seen from Table 3(a-h), 66 code segments are automatically marked or modified by each marker (N = 66). The function 'fisher.test()' is used to perform Fisher's exact test when the sample size is small, so as to avoid using an approximation that is known to be unreliable for small samples [19, 22]. The results of Fisher's exact test are presented in Table 4 and are obtained using the R programming language, which is an open-software platform.

TABLE 4 Results of Fisher's exact test

Participant	Odds Ratio	Confidence Interval(CI)	p-value
1	3.383684	(0.546, 37.024)	0.2576
2	1	(0.169, 5.922)	1
3	1	(0.124, 8.080)	1
4	2.290864	(0.536, 11.683)	0.3389
5	2.222357	(0.631, 8.576)	0.26
6	4.088773	(0.697, 43.673)	0.1487
7	1.932409	(0.432, 10.073)	0.5105
8	2.678179	(0.647,13.432)	0.2149

In the null hypothesis, the assumption is that the full and partial marking models are independent. As such, the assumption in the alternative hypothesis is such that the full and partial models are dependent. Table 4 shows that all the resulting p-values using Fisher's exact test are greater than $\alpha = 0.05$. Therefore, we would fail to reject the null hypothesis of equal proportions at the $\alpha = 0.05$ level. In addition, when the confidence interval (CI) crosses 1 (e.g. 95% CI 0.9-1.1), this implies that there is no difference between using the partial or full marking models. Even if the marker modified more code segments using the partial marking approach, there was no significant difference between the number of edited code segments using both marking approaches, as illustrated in Table 3(a-h). Thus, these results represent an encouraging and positive outcome for this study, since the participants provided nearly identical feedback, in terms of feedback consistency, using both marking approaches. However, in terms of marking time, the participants were found to save more time using the partial marking tool, as highlighted in the marking time comparison above. This makes the partial marking approach more efficient than the full marking approach in terms of marking workload.

5.3. Comparison of participants' comments on the two marking techniques

The participants generally provided positive comments about the marking tools. However, they have the similar concerns regarding the partial-marking approach. They stated that the partial marking tool can result in providing unsuitable comments for some code segments. The partial marking environment may cause this concern since it does not allow participants to link between code segments in a code script in the marking process. However, participants may be able to mitigate such a risk through the review process whilst retaining the efficiency gains since they are able to access the code scripts in the review process (see Figure 3). Overall, the participants highlighted that both marking tools are helpful for make comments based on the

formative assessment approach. Furthermore, they found that both marking approaches are innovative ideas. They also indicated that although the partial-marking tool is more efficient than the full-marking tool, they preferred the latter over the former. This could be due to the fact that the full-marking tool is more similar to the traditional way of marking compared to partial-marking.

6. CONCLUSIONS AND DIRECTIONS FOR FUTURE WORK

Two new marking models based on semi-automated assessment have been presented in this study. An experiment was carried out to learn which marking model is more efficient, and to understand whether the marking models are capable of reducing markers' workload in the marking process. The findings show that both marking models enable the provision of consistent (and nearly identical) feedback. However, despite the participants appreciating the value of both marking approaches, the partial marking approach was shown to offer a more efficient marking process, since it enables the provision of feedback in a shorter time period compared to the full-marking approach. Furthermore, the efficiency of the partial-marking approach can be further improved when it is utilised to mark long code scripts.

In this study, the two marking models were developed to provide feedback for novice programmers' solutions, which are typically short code scripts. However, in future work, long code scripts (including functions etc.) can be efficiently marked based on the developed marking models, especially using the partial marking approach. This will enable long code scripts to be assessed efficiently, and will save markers significant time while providing personalised and consistent feedback.

References

1. A. Gerdes, et al. Ask-Elle: an adaptable programming tutor for Haskell giving automated feedback. *International Journal of Artificial Intelligence in Education*, *27*(1) (2017), 65-100.
2. A. Holzinger, Usability engineering methods for software developers. Communications of the ACM, 48(1) (2005), 71-74.
3. A. Weinberger, A. (2011). Principles of transactive computer-supported collaboration scripts. *Nordic Journal of Digital Literacy*, 6(3) (2011), 189-202.
4. B. Auffarth, et al. System for automated assistance in correction of programming exercises (sac). In International Congress University Teaching and Innovation (CIDUI), 2008, 104-113.
5. D. Insa, J. Silva, Semi-automatic assessment of unrestrained Java code: a library, a DSL, and a workbench to assess exams and exercises. In Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education, 2015, 39-44.
6. D. Jackson, A semi-automated approach to online assessment. In Proceedings of the 5th annual SIGCSE/SIGCUE ITiCSE conference on Innovation and technology in computer science education, 32(3) 2000, 164-167.
7. H. Suleman, Automatic marking with Sakai. Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology, 2008, 229-236. ACM.
8. J.A. Sant, Mailing it in: email-centric automated assessment. *In ACM SIGCSE Bulletin*, 41(3), 308-312, ACM.
9. J. Ludbrook, Analysis of 2× 2 tables of frequencies: matching test to experimental design. *International journal of epidemiology*, *37*(6), (2008),1430-1435.
10. J. Carter, et al. How shall we assess this? ACM SIGCSE Bulletin 35(3) 2003, 107-123.

11. J. W. Creswell, *Qualitative inquiry and research design: Choosing among five traditions.* Thousand Oaks, CA: Sage Publications,1998.
12. K.A. Reek, The TRY system-or-how to avoid testing student programs. *ACM SIGCSE Bulletin*, 21(1) (1989),112-116, ACM.
13. K. Georgouli, P. Guerreiro, Incorporating an automatic judge into blended learning programming activities. In International Conference on Web-Based Learning, Springer, Berlin, Heidelberg, 2010, 81-90.
14. K. K. Sharma, et al. Automated checking of the violation of precedence of conditions in else-if constructs in students' programs. MOOC, Innovation and Technology in Education (MITE) IEEE, 2014, 201-204.
15. K. M. Ala-Mutka, A survey of automated assessment approaches for programming assignments. *Computer science education*, 15(2) (2005), 83-102.
16. M. Benson, Machine assisted marking of programming assignments. ACM SIGCSE Bulletin, 17(3) 1985, 24-25.
17. M. Joy, M. Luck, M. The BOSS On-line Submission System, 1998.
18. M. Kaya, S. A. Özel, Integrating an online compiler and a plagiarism detection tool into the Moodle distance education system for easy assessment of programming assignments. Computer Applications in Engineering Education, 23(3) (2015), 363-373.
19. M. Raymond, F. Rousset, F, An exact test for population differentiation. *Evaluation*, ,49(6) (1995), 1280-1283.
20. M. Resnick, et al. (2009). Scratch: programming for all. Communications of the ACM, 52(11) (2009), 60-67.
21. M.T. Helmick, Interface-based programming assignments and automatic grading of java programs. *In ACM SIGCSE Bulletin*, 39(3) (2007), 63-67, ACM.

22. R.A. Fisher, On the interpretation of χ 2 from contingency tables, and the calculation of P. *Journal of the Royal Statistical Society*, *85*(1), (1992), 87-94.
23. S. Buyrukoglu, F. Batmaz, R. Lock, A new marking technique in semi-automated assessment. In 12th International Conference on Computer Science and Education (ICCSE) IEEE, 2017, 545-550.
24. S. H. Edwards, M. A. Perez-Quinones, Web-CAT: automatically grading programming assignments. In ACM SIGCSE Bulletin, 40(3) 2008, 328-328.
25. S. H. Tung, T. Lin, Y. Lin, An exercise management system for teaching programming. *Journal of Software*, 8(7) (2013), 1718-1725.
26. S. Nutbrown, C. Higgins, Static analysis of programming exercises: Fairness, usefulness and a method for application. *Computer Science Education*, 26(2-3) (2016),104-128.
27. S. Y. Lye, J. H. L. Koh, Review on teaching and learning of computational thinking through programming: What is next for K-12? *Computers in Human Behavior*, 41 (2014), 51-61.
28. T. Ahoniemi, T. Reinikainen, ALOHA-a grading tool for semi-automatic assessment of mass programming courses. In Proceedings of the 6th Baltic Sea conference on Computing education research, 2006, 139-140, ACM.
29. T. Wang, et al. Ability-training-oriented automated assessment in introductory programming course. *Computers & Education*, *56*(1) (2011), 220-226.
30. V. Pieterse, Automated assessment of programming assignments. In Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research, 2013, 45-56.