Mobile Grading Paper-Based Programming Exams: Automatic Semantic Partial Credit Assignment Approach

I-Han Hsiao(✉)

School of Computing, Informatics and Decision Systems Engineering,
Arizona State University, 699 S. Mill Ave., Tempe, AZ, USA
Sharon.Hsiao@asu.edu

Abstract. In this paper, we report a study of an innovative mobile application to support grading paper-based programming exams. We call the app – Programming Grading Assistant (PGA). It scans pre-generated QR-codes of paper-based question-and-concepts associations and uses OCR to recognize handwritten answers. PGA provides interfaces for teachers to calibrate recognition results, as well as to adjust partial credit assignment according to conceptual incorrectness of the answers. We evaluate the mobile grading process and the quality of grading results based on the assessed semantic information. The results demonstrate that the mobile grading approach keeps persistent traces of students' performance, including semantic feedback and ultimately enhances grading consistency.

1 Introduction

Today, the majority of programing classes are delivered via a blended instructional method with face-to-face instruction in the classrooms supported by online tools such as intelligent tutors, self-assessment quizzes, and course management systems etc. Such a blended instructional strategy, in contrast to pure on-line learning/instruction through massive open online courses, which still has inconclusive results [1, 2], allows teachers to focus on systematically instructing complex topics in class while supplying many supplemental exercises outside the classroom. The blended instructional classrooms still mainly rely on paper-based exams as the main method to assess students' knowledge in today's lower division programming courses. It is very challenging for teachers to provide personalized feedback on each individual test. The large size of the classes makes it impractical to discuss with each individual student on his/her exam paper. Instead, typically, teachers discuss on the returned exam in the class (hopefully thorough and detailed enough to cover all the students' misconceptions).

Although teachers still point out the common mistakes and try to pinpoint the key concepts related to the such mistakes, many desired detailed learning analytics are unavailable, such as *how did s/he receive partial credits*, *was it a single concept or multiple concepts mistake*, *a careless mistake or a long-term misconception* etc. As a result, students often focus solely on the scores they earned on the returned exams, but miss several learning opportunities, such as *identification of strength and weakness*,

K. Verbert et al. (Eds.): EC-TEL 2016, LNCS 9891, pp. 110–123, 2016.
DOI: 10.1007/978-3-319-45153-4_9

characterization of the nature of their errors or any recurring patterns if any, *assessment of appropriateness of their study strategies and preparation*, etc. Furthermore, from teachers' perspective, there is an increasing difficulty in managing paper-based exams. Teachers can hardly memorize all students' names, it is becoming even more challenging for teachers to manage all mistake patterns in students' exam answers. Thus, it is common for teachers to focus on common mistakes based on the history of a course rather than a specific exam. Moreover, with graders or teaching assistants being recruited to do the majority of grading in order to handle large classes, teachers could overlook the detailed course performances. In this case, varying level of training of the graders and potential inconsistency among graders are among additional factors that may further complicate students' learning.

In this work, we investigate an innovative method to capture paper-based programming exams in order to provide semantic personalized feedback in supporting large-scale auto-grading in blended instruction classrooms. We emphasize that paper-based approach is still one of the primary and preferred ways of delivering programming exams, due to the sake of easiness and other logistics or potential academic dishonesty issues to occur in online settings. The rest of the paper is organized with reviewing related work in Automatic Program Assessment and Technology Support for Blended Classroom in Computing Education. We then describe the study methodology and layout the study design. Finally, we present the evaluation results and discuss the approach with current limitations and future work.

2 Literature Review

2.1 Automatic Program Assessment

Automatic program evaluation is not a new topic. Special interest group on computer science education (SIGCSE) reports several work on automatic grading students' programming assignments in last couple decades. For instance, WEB-CAT [3], BOSS [4], ASSYST [5] and among many others. The common approach is to apply pattern-matching techniques to verify students' answers and the correct answers. Most of these systems are web-based evaluation tools; less is emphasized on automatic evaluation for paper-based programming formal assessments. There are a few relevant early innovations attempting to apply to process paper exams and hand written codes, such as tablet grading system [6, 7]. It uses tabletop scanners to digitalize the exam papers and provides a central grading interface on the tablet in assisting mass programming grading. It reports that a few benefits of digitizing paper exams (i.e. some default feedback can be kept on the digital pages; student's identity can be kept anonymous and potentially prevent graders' bias in recognizing names). There is also an adjacent related work attempted to address scaling up assessment production, it is called parameterized exercises. Parameterized questions and exercises use randomly generated parameters in every question template and produce many similar, yet sufficiently different questions. This approach not only automatically evaluates students' programs, but also dramatically reduces authoring efforts and creates a sizeable collection of questions to facilitate programming assessment. As demonstrated by a number of projects such as CAPA [8], WebAssign [9], QuizPack [10], and QuizJET [11] parameterized questions can be used effectively in a number of domains allowing to increase the number of assessment items, decrease authoring efforts and reduce cheating. Overall, the field of automatic program evaluation is less focused on grading paper-based programming problems, therefore, less support of personalization as such.

2.2 Technology and Instructional Support for Blended Classroom in Computing Education

In the field of Computer Supported Collaborative Learning (CSCL), researchers describe classroom orchestration as a field in transition, which defines how a teacher manages multilayered activities in real time and in a multi-constraints context. It discusses how and what research-based technologies have been adopted and should be done in classrooms [12]. We have begun to see more tabletops, wearable cameras, smart classrooms or interactive tools such as Classroom Response Systems (AKA: Clickers) etc. provide dynamic feedback and integrative students knowledge updates [13–15]. Such tools attempt to capture in-the-moment teaching pace and on–the-fly students' learning pace; however, they are usually highly customized to the content or require a large collection of content for teachers to start using the tools. Classes may still suffer from lacking comprehensive content collections due to high development or maintenance costs or run into *off-sync* issues when the students leave the interactive classrooms. Over the last decade, several educational technologies and instructional pedagogies have been proposed and studied to amend and assist large size of classrooms. For instance, flipped classroom model to promote utilization of class time for interaction [16]; peer instruction to facilitate students' conceptual reasoning [17, 18]; media computation to increase learning motivation [19, 20] etc. In the context of computing education, a dozen of research projects attempted to apply these methods in programming classes [21–24]. In spite of there were positive results reported, most of the findings were still in early stage and inconclusive. For example, instructors are supposed to utilize the class time to maximize student and teacher interactions, however, until today, it is still challenging to gather and interact with large amount of students with laptops in the lecture hall or in the computer lab [25].

3 Methodology

3.1 Mobile Grading Framework

In order to automatically evaluate programming problems on paper-based exams, we create a mobile PGA grading framework (Fig. 1) and develop an instance of android application[1] based on the framework. (1) Using a camera-enabled mobile device to scan questions, which are attached with pre-generated Quick Response codes (Fig. 2). The scanning can be done at a batch process by scanning multiple questions and

[1] The mobile grading app is currently available upon request.

multiple exams at a time before entering to auto-grading phase; (2) The grading service begins to process the scanned questions as images, which includes: (a) Converting pixel images to binary images; (b) Removing noises from the image in order to just focus on texts in the recognition step, but not to falsely remove the punctuations, which is considered an important element in code writing; (c) Defining character boundaries for later recognition to calculate word separation and alignments. (3) We deploy an open source OCR (Optical Characteristic Recognition) library[2] to recognize hand written and/or printed texts from the scanned images; (4) The app then compares the recognized answers to the correct answers; (5) The app assigns scores based on two grading schemes: (a) a binary function of correct or incorrect answer; (b) a partial credit assignment based on the proportion of recognized concepts to the overall concepts of the correct answer; (6) Finally, the app aggregates step 1 to 5 results to generate reports and updates analytics.

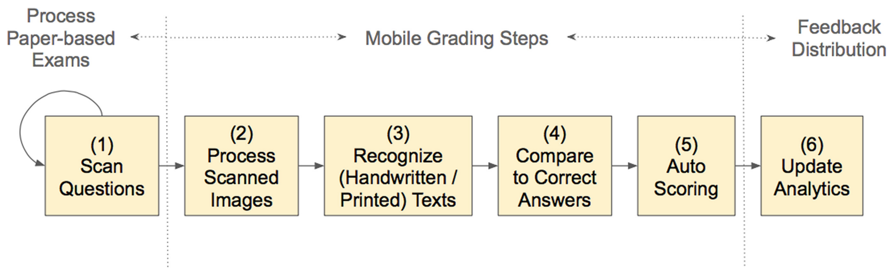

Process Paper-based Exams
Mobile Grading Steps
Feedback Distribution
(1) Scan Questions
(2) Process Scanned Images
(3) Recognize (Handwritten / Printed) Texts
(4) Compare to Correct Answers
(5) Auto Scoring
(6) Update Analytics

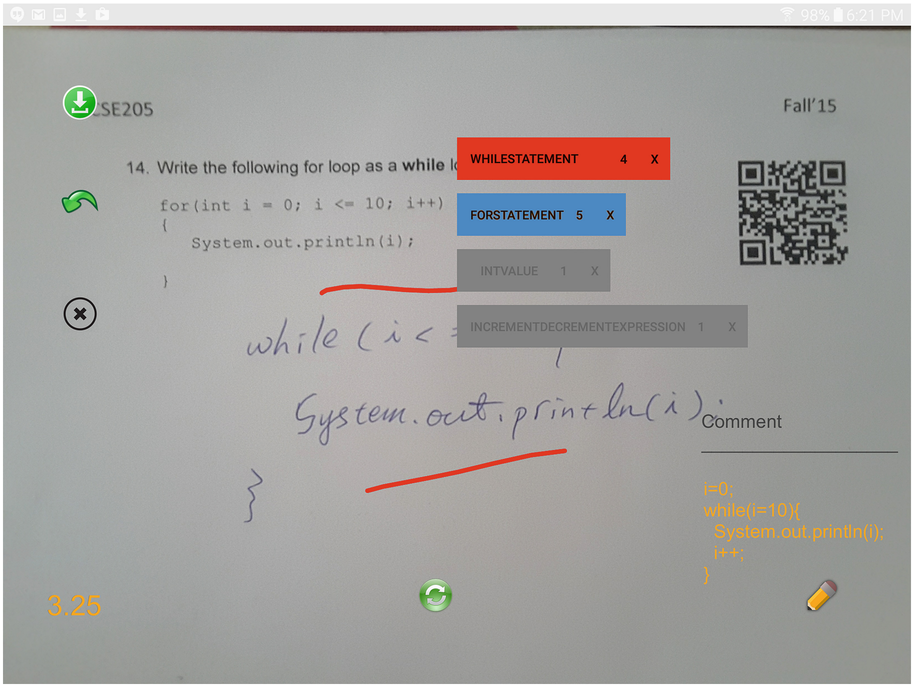
Fig. 1. Mobile grading framework.

3.2 Research Platform: Programming Grading Assistant

We develop an Android application and deploy it in a Samsung Galaxy Tab S2 8 device to support grading paper-based exams. Currently, we consider two major types of programming problems on the exams, multiple choice question (MCQ) and code writing question (CWQ). Due to the handwritten texts naturally carry various complications, such as scripts and prints mixture, word separation ambiguity, inconsistent word alignments etc. Based on the unique characteristics of these two types of questions (MCQ consists of limited handwriting texts and printed question texts; CWQ mainly includes substantial amount of handwriting texts and limited printed texts), we design two separate mobile grading modules to deal with different level of handwriting recognition complexity. In this paper, we focus on the CWQ module, which allows us to focus on the proposed technology in examining procedural knowledge of programming problem solving.

CWQ Grading Module. This module utilizes 2-dimensional quick response code (QR-code) to associate each question, answer, corresponding concepts and weights (the importance of the concepts). Figure 2 demonstrates the CWQ grading interface, where

[2] https://github.com/tesseract-ocr.



14. Write the following for loop as a while l

```
for(int i = 0; i <= 10; i++)
{
    System.out.println(i);
}
```

```
while (i <
  System.out.println(i);
}
```

```
i=0;
while(i=10){
  System.out.println(i);
  i++;
}
```

Fig. 2. Code writing question grading interface.

the grader can keep notes anywhere on the screen and to leave free-form feedback or just simply highlight the missing/incorrect codes with her fingertip. The pencil icon located at the lower right is the functionality to edit misconceptions. Grading is done by tapping on the concepts, which resembles the action of punishing misconceptions of incorrect codes. The grades are shown at the lower left corner. The graders can also manually adjust the grades as appropriately. Once graders are done with editing, they can press the top-left corner save icon. All the graded questions will be recorded into database and be exported as an xml format file to be ready to feed in learning analytics dashboard tool, such as EduAnalysis [26].

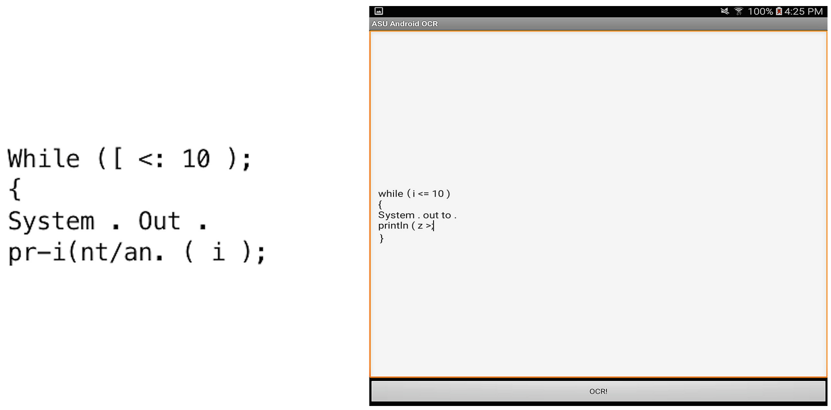
Semantic Feedback. All questions' associated concepts are parsed and indexed by Java Programming ontology[3] and weights are automatically indexed by EduAnalysis [26]. On toggling on and off the pencil icon, the associated concepts of the question will be brought out to the screen. The grader can tap to edit the weights to indicate the missing concepts and/or misconceptions. In Fig. 2, the red highlighted concept indicates the *misconception*, blue shows the *gained concept*, and grey shows *missing concepts*. In this CWQ example, the student has clearly missed to initialize a counter variable *i* (`int i=0`) and the *increment statement* (`i++`) in the *while* loop. Therefore, the *IntVariable* and *IncrementDecrementExpression* concepts are greyed out.

[3] Source of Java Ontology: http://www.pitt.edu/~paws//ont/java.owl.

Such grading process not only leaves conceptual feedback for each question, but also allows automatic partial credit computation (will be discussed in next sub-section).

3.3 Recognition Optimization

Students' handwritings are heterogeneous. Typically, OCR requires a training process to calibrate recognition. However, due to the target corpus in this experiment is programming language domain, we anticipate students will only be writing code syntactical texts. In other words, all recognized texts should be identified from Java glossary. For instance, Fig. 3(a) shows the OCR recognized results from the same example in Fig. 2. However, without proper training, the recognition fails to identify variable *i* or the *print* method. In order to minimize the training effort, we adopt spelling correction logic and implement *recognition correction* algorithm. We use *Damerau–Levenshtein* distance algorithm [27] to iteratively transpose, replace and insert the recognized characters from simultaneously referencing to Java glossary dictionary. In Fig. 3(b) illustrates a corrected recognition codes, which improves the readability of original recognition. Note that the punctuations and variables are still not yet optimized.



```
While ([ <: 10 );
{
System . Out .
pr-i(nt/an. ( i );
```

```
while ( i <= 10 )
{
System . out to .
println ( z >;
}
```

Fig. 3. Before (a) and after (b) optimized hand writing recognition.

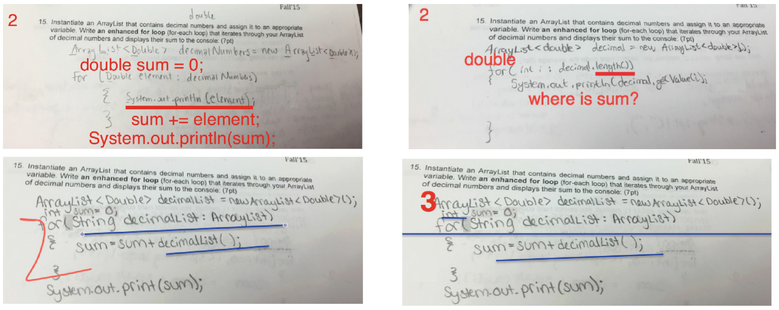
3.4 Semantic Partial Credit Algorithm

Assigning grades on the programming exams is not a trivia task. Each code-writing question can potentially have multiple solutions; each solution can have miscellaneous variations, such as local variable differences, utility methods, interfaces etc. Therefore, it may not be fair to assign scores by judging codes similarity between students' and teacher's codes. Typically, teachers will evaluate the code solutions and assign partial credits to award the logic soundness instead of code completeness. For instance, a common strategy is to give points to conceptual integrity and deduct points by punishing conceptual mistakes. The question is, *how many points are appropriate as partial credits?*

We discover there often exists the inconsistency in assigning partial credits. Figure 4 illustrates some of the inconsistent scenarios: (a-left) The student clearly implemented key concepts *ArrayList* and *Foreach-loop*, but missed to aggregate values from each iteration and to print out the final results. However, this student was being punished by missing minor concepts and suffered from major credits loss (5 out of 7). In case (a-right) shows the same grader gave same partial credits to another student, while s/he not only missed out the *sum* variable declaration and initialization, but actually wrongly implemented the *Foreach-loop*. In case (b), it shows a different grader gave different amount of partial credits on the exactly the same question. These examples demonstrate a series of grading inconsistency issues: *oversight on key concept misconception*, *over-emphasized on minor concepts*, *limited feedback* etc.

(a)

(b)



2 Fall'15
15. Instantiate an ArrayList that contains decimal numbers and assign it to an appropriate variable. Write **an enhanced for loop** (for-each loop) that iterates through your ArrayList of decimal numbers and displays their sum to the console: (7pt)

```
double
ArrayList<Double> decimalNumbers = new ArrayList<Double>();
double sum = 0;
for (Double element : decimalNumbers)
{   System.out.println(element);
}   sum += element;
System.out.println(sum);
```

2 Fall'15
15. Instantiate an ArrayList that contains decimal numbers and assign it to an appropriate variable. Write **an enhanced for loop** (for-each loop) that iterates through your ArrayList of decimal numbers and displays their sum to the console: (7pt)

```
ArrayList<double> decimal = new ArrayList<double>();
double
for( int i : decimal.length())
{   System.out.println(decimal.getValue(i));
where is sum?
}
```

Fall'15
15. Instantiate an ArrayList that contains decimal numbers and assign it to an appropriate variable. Write **an enhanced for loop** (for-each loop) that iterates through your ArrayList of decimal numbers and displays their sum to the console: (7pt)

```
ArrayList<Double> decimalList = newArrayList<Double>();
int sum= 0;
for(String decimalList: ArrayList)
{   sum=sum+decimalList();
}
System.out.print(sum);
```

3 Fall'15
15. Instantiate an ArrayList that contains decimal numbers and assign it to an appropriate variable. Write **an enhanced for loop** (for-each loop) that iterates through your ArrayList of decimal numbers and displays their sum to the console: (7pt)

```
ArrayList<Double> decimalList = newArrayList<Double>();
int sum= 0;
for(String decimalList: ArrayList)
{   sum=sum+decimalList();
}
System.out.print(sum);
```

Fig. 4. (a) left & right are two different students' answers that were graded by the same grader; (b) left & right are the same student's answer that were graded by two different graders.

In order to enforce consistency in assigning partial credits, we design a semantic partial credit algorithm to calculate students' proportional conceptual errors of a question (Table 1). There are 3 parameters to determine the partial credits: *Concept Similarity*, *Concept Saliency*, and *Miscellaneous*. We assume partial credit would be given based on conceptual inconsistency with the correct answer, therefore, *Concept Similarity* is calculated by the cosine similarity between student's and the correct one. We use *Concept Saliency* coefficient (Eq. 1) to highlight the importance of key concept and to demote peripheral conceptual mistakes. For instance, the question in Fig. 4 (a-left) deserves more credits when the key concepts are intact, and the peripheral concepts are missed; vice versa in Fig. 4(a-right). Finally, we reserve a *Miscellaneous* coefficient to capture all other mistakes that are not conceptual, such as careless mistakes.

3.5 Study Setup

We design a study to investigate paper-based programming exams grading process, specifically focus on the semantic partial credit assignment effects. We randomly sampled 20 students' exams from an Object-Oriented Programming & Data Structure

Table 1. Semantic partial credit algorithm

```
function pc = partial_credit(question){
if concepts incorrect then
        pc = $ConceptSimilarity_{Tester\&Correct}$ * ConceptSaliency;
        $ConceptSaliency = W_{c,q} / \sum_i W_{c_i,q}$                    (Eq.1)
        if pc<0 then return pc = 0;
else
        pc = 1 – $\varepsilon_{Miscellaneous}$;
        if pc<0 then return pc = 0;
}
        return pc;
```

class in 2015 Fall semester offered in Arizona State University. We recruited six graders who have been graders or teaching assistants for the same course at least once. Among the recruited 6 graders, there are 3 graduate students and 3 undergraduate students, 1 female and 5 male. All of them are either Computer Science major or Information Science major. They have 1 ~ 5 years Java programming experiences and have taken 3 ~ 8 programming courses. In addition, they all code multiple all-purposes programming languages on a daily basis (mainly, C, C++, and PHP).

Data Collection. We scan two questions and answers from the sampled 20 paper-based exams and use photo editing software to remove the original grading remarks from the scans. Thus, there are 40 questions (2 questions × 20 different students' exams) in total. Exam questions are presented in Table 2.

Table 2. Sampled exam questions & answer keys/grading scheme

Question	Answers & Grading Scheme

1. Write the following for loop as a **while** loop: (5pt)
 	
`for(int i = 0; i <= 10; i++)`
`{`
`System.out.println(i);`
`}`
	`// full score must be exactly the same codes`
	`// as the correct answer (except variable names)`
	`int i = 0;          // -1 incorrect initialization`
	`while (i<=10){      //-3 incorrect while statement`
	`System.out.println(i);`
	`i++;          // -1 incorrect increment statement`
	`}`
	`// -0.5 other errors`

2. Instantiate an ArrayList that contains decimal numbers and assign it to an appropriate variable. Write **an enhanced for loop** (for-each loop) that iterates through your ArrayList of decimal numbers and displays their sum to the console: (7pt)	`// -2 incorrect ArrayList`
	`ArrayList<Double> numList = new ArrayList<Double>();`
	`double sum = 0;`
	
	`// -4 incorrect enhanced for-loop statement`
	`for(Double d: numList){`
	`sum += d;`
	`}`
	`System.out.println(sum);`
	`// -0.5 other errors`

Study Procedure. In the lab study, we instructed graders to refer to the provided solution keys and grading schemes (Table 2) and assign grades based on their best judgment to all 40 questions. Noted that graders graded on the same 40 questions. The grading scheme was solicited from the same teacher who designed the exam. Graders were also instructed to mark or leave feedback as appropriately. They spent 10∼33 min to finish grading all the questions.

4 Evaluation Results

4.1 Semantic Partial Credit Accuracy

We compared all graders' grading and automatic partial credit algorithm calculation results to original teacher's grading. We set the threshold at 0.1 marks; where we consider it is a correct answer when the grades differences are less than the threshold. We found that given the grading schemes for graders (per Table 2), they could still make considerably inaccurate grading outcomes. The inaccuracy effect is especially noticeable in Q2, which is a more complex question than Q1. Q2 involves more key concepts and more overall concepts. Recall that the case of Fig. 4-(a), the student clearly knew how to implement *ArrayList* and *Foreach Loop*, but the grader penalized this answer as missing a lot codes, and neglected the grading scheme. Figure 5 illustrates all graders CWQ grading accuracy distribution by question compared to automatic method. Overall, we found that automatic partial credit algorithm improved 20% of the accuracy.

Additionally, We found that the complex question (Q2) consistently being taken more than 20 points off in total, and averagely each question was mis-graded respectively 0.658 and 1.491 in Q1 and Q2 (Table 3). Meanwhile, the automatic partial credit algorithm achieved low grades discrepancy in both Q1 and Q2. On average, they were graded only 0.122 and 0.370 off compared to teacher's grades. Such results suggested that if an exam consists of 10 code-writing questions, the variance between grader and teacher could be as large as one letter grade.

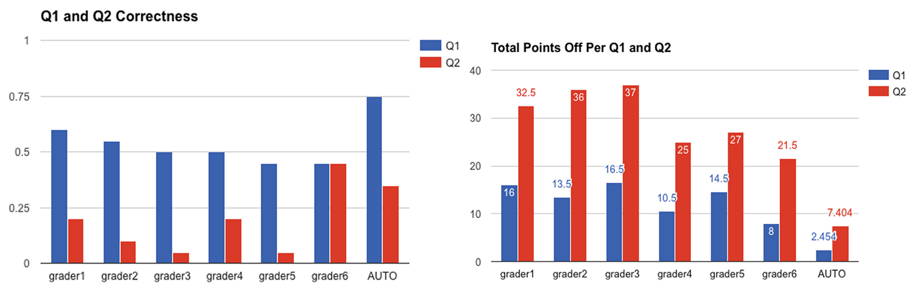


Fig. 5. Partial Credit Accuracy (left); Grades Discrency by Question Complexities (right).

Table 3. Grading discrepancy magnitude

Average	Grader	Auto PC Algorithm
Q1	0.658	0.122
Q2	1.491	0.370

4.2 Mobile Grading Enhances Grading Coherence

To gauge graders' coherence, we evaluated how did the graders grade questions; *do they give extra points than they should have?* or *do they penalize the students in deducting more points than they should have?* Figure 6 showed each grader's inconsistency. We found that grader 1, 3, 5, and 6 tend to give more credits; grader 2 and 4 tended to be stricter and give fewer credits than they should have. The gaps among graders are evident. We found that automatic partial credit algorithm achieved higher grading coherence (smaller gap between + & −).

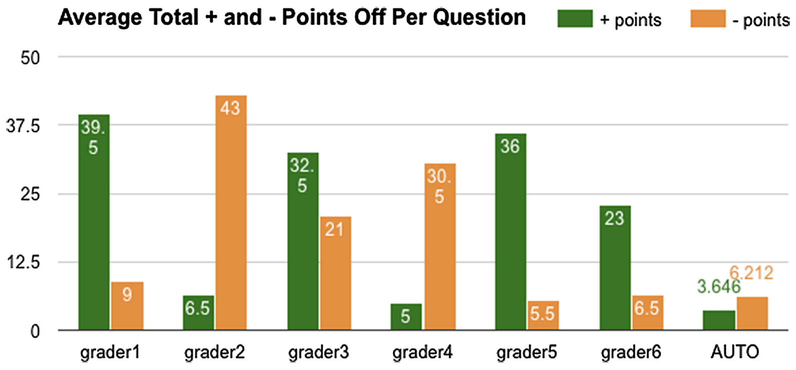


Fig. 6. Inconsistencies Among Graders, some graders graded loosely, some strictly.

In addition, noted that currently the auto-grading algorithm tends to give fewer points to students (-0.155 points on average), which means PGA is rather to grade slightly harshly than mercifully. On the other hand, graders give mixed signals in grading; either half point more or half point less (Table 4). Such results show prominent news for teachers, who may consider partial incorrectness as incorrect rather than giving false positive grading and mislead students.

Table 4. Grading discrepancy magnitude

Average	Grader	Auto PC Algorithm
+	0.594	0.091
−	0.481	0.155

4.3 Feedback Quality

We analyze graders' feedback on total 180 questions not-entirely-correct questions (30 out of 40 sampled questions per grader). We categorize six types of graders' feedback: *No feedback at all*, *Highlights on students' errors*, *(Partial) correct answers*, *Justifications on penalty*, *Conceptual feedback*, and *Wrong feedback* (Fig. 7).

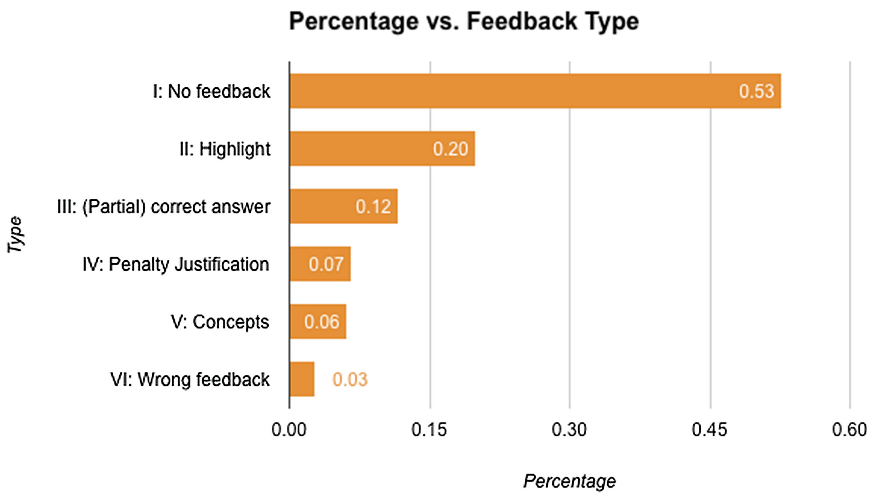
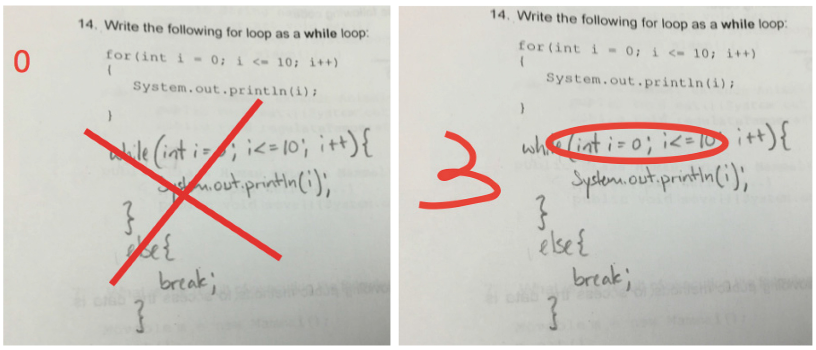


Fig. 7. Feedback types and percentage.

We found that the majority of the questions (52.8%) did not receive feedback from the graders at all. The cases are often occurred when the students had completely wrong implementation. They usually receive *a big red cross* and *zero* along with the question and nothing else (Fig. 8 Left). These students are usually the ones who have none or incomplete knowledge and demand for more support. However, they tend not to get any feedback at all on the paper-based exams. Nonetheless, the second type of feedback


0

14. Write the following for loop as a while loop:

```
for(int i = 0; i <= 10; i++)
{
    System.out.println(i);
}
```

```
while(int i=0; i<=10; i++){
    System.out.println(i);
}
else{
    break;
}
```

3

14. Write the following for loop as a while loop:

```
for(int i = 0; i <= 10; i++)
{
    System.out.println(i);
}
```

```
while(int i=0; i<=10; i++){
    System.out.println(i);
}
else{
    break;
}
```

Fig. 8. Left: No feedback at all; Right: Highlights on errors.

is to highlight students' errors (20.0%) (Fig. 8 Right). In such scenario, students could potentially obtain point-of-interests to focus on mistakes, but no further guidance. Unfortunately, these two types of feedback are not only shallow, but also very common strategies for grading paper-based exams.

Type III feedback is that graders directly write down the answers or partial answers on students' exams (11.7%). For example, Fig. 3(a). Type IV feedback lists the reasons why there are points of deduction (6.7%), for instance, graders left comments on the exams *"your sum is not computed", "results is not displayed", "your output is misplaced", "where is sum?"* Type V explains the misconceptions semantics (6.1%), for example, *"Wrong while condition", and "no initialization"*. These three types (Type III ~ V) are considered more substantial feedback. However, in the context of learning, the correct solutions may not necessarily the best next steps for all learners. It is harder to provide personalized feedback on paper-based exams, due to the lack of understanding on students other learning performances. Finally, Type VI shows a 2.8% of messages are actually wrong feedback to the students.

5 Conclusions

5.1 Summary

In this work, we design and evaluate an innovative mobile application to investigate automatic grading paper-based programming exams. We call it Programming Grading Assistant (PGA), it utilizes mobile's inbuilt-camera to scan question and answers. We use OCR technology to recognize students' handwriting answers and design interfaces to calibrate recognition or log misconceptions. We use 2-dimensional quick response code (QR-Code) to associate each code-writing question, answer, and their corresponding concepts and importance. Based on semantics associations with the exam content, a partial credit assignment algorithm is constructed to leverage grading inconsistency and to be utilized to provide semantic feedback.

Study results show that human graders exhibit multiple grading inconsistencies and provide insufficient and shallow feedback. Meanwhile, PGA not only elevates the grading consistency, but also systematically assigns partial credits and improves the grading coherence. It also reveals human graders provide insufficient feedback, while the proposed approach provides consistent semantics remarks as feedback. In addition, handwriting recognition is currently not optimized, but can be improved with recognition correction logic. Overall, PGA's auto-grading framework via mobile devices shows promising results in capturing paper-based programming exams for advanced learning analytics.

5.2 Limitations and Future Work

In spite of several promising findings, there are a few limitations in current study. First of all, the handwriting recognition requires *good* lighting (i.e. natural sun light) and ball pen writing. In our experiment, we found that indoor lighting often resulted in recognition failure, which was also one of the reasons that took slightly longer time than we expected. In addition, penciled texts of students' answers also resulted in recognition failure. However, programming exams typically require iterative problem solving and trial-and-error, thus, students usually prefer to use pencils than pens. We recognize these challenges with OCR technology, and have begun to instruct students to write their code carefully by following sound Object-Oriented Programming principles and coding conventions.

Secondly, we did not measure codes recognition accuracy yet, since the recognition is not yet optimized. It is in our research agenda, to expedite fully automatic grading process and to reach reliable consistent grading outcome. We have begun training to use designated underscores for uppercases and whitespaces to increase word separation recognition.

In the near future, we anticipate providing personalized feedback to students from their paper-based exams. We are currently developing APIs to synchronize grading results to the learning analytics dashboards [26]. We plan to conduct more user studies and larger scale of field studies to explore PGA with graders in UI related issues and measure the effects for grading entire class exams.

References

1. Zhenghao, C., et al.: Who's Benefiting from MOOCs, and Why. Harvard Business Review (2015)
2. Kolowich, S.: Puts MOOC Project with Udacity on Hold, in The Chronicle of Higher Education (2013)
3. Edwards, S.H., Perez-Quinones, M.A.: Web-CAT: automatically grading programming assignments. In: ACM SIGCSE Bulletin. ACM (2008)
4. Joy, M., Griffiths, N., Boyatt, R.: The boss online submission and assessment system. J. Educ. Res. Comput. (JERIC) **5**(3), 2 (2005)
5. Jackson, D., Usher, M.: Grading student programs using ASSYST. In: ACM SIGCSE Bulletin. ACM (1997)
6. Bloomfield, A., Groves, J.F.: A tablet-based paper exam grading system. In: ACM SIGCSE Bulletin. ACM (2008)
7. Bloomfield, A.: Evolution of a digital paper exam grading system. In: 2010 IEEE Frontiers in Education Conference (FIE). IEEE (2010)
8. Kashy, E., et al.: Using networked tools to enhance student success rates in large classes. In: Proceedings of the 27th Annual Conference Frontiers in Education Conference, 1997. Teaching and Learning in an Era of Change. IEEE (1997)
9. Titus, A.P., Martin, L.W., Beichner, R.J.: Web-based testing in physics education: methods and opportunities. Comput. Phys. **12**(2), 117–123 (1998)
10. Brusilovsky, P., Sosnovsky, S.: Individualized exercises for self-assessment of programming knowledge: an evaluation of QuizPACK. J. Educ. Res. Comput. (JERIC) **5**(3), 6 (2005)
11. Hsiao, I.-H., Sosnovsky, S., Brusilovsky, P.: Guiding students to the right questions: adaptive navigation support in an E-Learning system for Java programming. J. Comput. Assist. Learn. **26**(4), 270–283 (2010)
12. Dillenbourg, P.: Design for classroom orchestration. Comput. Educ. **69**, 485–492 (2013)

13. Martinez-Maldonado, R., et al.: Capturing and analyzing verbal and physical collaborative learning interactions at an enriched interactive tabletop. Int. J. Comput. Support. Collaborative Learn. **8**(4), 455–485 (2013)
14. Roschelle, J., Penuel, W.R., Abrahamson, L.: Classroom response and communication systems: research review and theory. In: Annual Meeting of the American Educational Research Association (AERA). San Diego, CA, pp. 1–8 (2004)
15. Slotta, J.D., Tissenbaum, M., Lui, M.: Orchestrating of complex inquiry: three roles for learning analytics in a smart classroom infrastructure. In: Proceedings of the Third International Conference on Learning Analytics and Knowledge. ACM (2013)
16. Bishop, J., Verleger, M.: The flipped classroom: a survey of the research. In: 120th ASEE Annual Conference & Exposition. Atlanta, GA (2013)
17. Crouch, C.H., Mazur, E.: Peer instruction: ten years of experience and results. Am. J. Phys. **69**(9), 970–977 (2001)
18. Fagen, A.P., Crouch, C.H., Mazur, E.: Peer instruction: results from a range of classrooms. Phys. Teach. **40**(4), 206–209 (2002)
19. Guzdial, M.: Exploring hypotheses about media computation. In: Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research. ACM (2013)
20. Porter, L., et al.: Success in introductory programming: what works? Commun. ACM **56**(8), 34–36 (2013)
21. Simon, B., et al.: Experience report: peer instruction in introductory computing. In: Proceedings of the 41st ACM Technical Symposium on Computer Science Education. ACM (2010)
22. Simon, B., et al.: Experience report: CS1 for majors with media computation. In: Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education. ACM (2010)
23. Sarawagi, N.: Flipping an introductory programming course: yes you can! J. Comput. Sci. Coll. **28**(6), 186–188 (2013)
24. Amresh, A., Carberry, A.R., Femiani, J.: Evaluating the effectiveness of flipped classrooms for teaching CS1. In: 2013 IEEE Frontiers in Education Conference. IEEE (2013)
25. Rosiene, C., Rosiene, J.: Flipping a programming course: the good, the bad, and the ugly. In: Frontiers in Education Conference. IEEE (2015)
26. Hsiao, I.-H., Govindarajan, S.K.P., Lin, Y.-L.: Semantic visual analytics for today's programming classrooms. In: The 6th International Learning Analytics and Knowledge Conference. ACM, Edinburgh, UK (2016)
27. Brill, E., Moore, R.C.: An improved error model for noisy channel spelling correction. In: Proceedings of the 38th Annual Meeting on Association for Computational Linguistics. Association for Computational Linguistics (2000)