Problems Before Solutions: Automated Problem Clarification at Scale

Soumya Basu
UC Berkeley
soumyab@berkeley.edu

Albert Wu
UC Berkeley
albert12132@berkeley.edu

Brian Hou
UC Berkeley
brian.hou@berkeley.edu

John DeNero
UC Berkeley
denero@berkeley.edu

ABSTRACT

Automatic assessment reduces the need for individual feedback in massive courses, but often focuses only on scoring solutions, rather than assessing whether students correctly understand problems. We present an enriched approach to automatic assessment that explicitly assists students in understanding the detailed specification of technical problems that they are asked to solve, in addition to evaluating their solutions. Students are given a suite of solution test cases, but they must first unlock each test case by validating its behavior before they are allowed to apply it to their proposed solution. When provided with this automated feedback early in the problem-solving process, students ask fewer clarificatory questions and express less confusion about assessments. As a result, instructors spend less time explaining problems to students. In a 1300-person university course, we observed that the vast majority of students chose to validate their understanding of test cases before attempting to solve problems. These students reported that the validation process improved their understanding.

Author Keywords

automated assessment; behavioral analytics; online learning

INTRODUCTION

Instructors of massive courses must be frugal and efficient with their attention to individuals in all aspects of course delivery. For assignments, automatic assessment of solutions can dramatically reduce or even eliminate the individual attention required for assigning scores, allowing instructors to focus on content delivery and interaction design. This paper describes the scaling benefits of another form of automated assessment for technical assignments: automatic verification of problem understanding. By validating each student's understanding of each problem statement automatically, our system substantially reduces the instructor intervention required for clarification and explanation of assignments.

This paper describes a particular implementation of our approach to validating problem understanding. Our system, called OK[1], performs automatic assessment for programming-based assignments. Each assignment in our course is distributed with a suite of test cases that are designed to verify a student's proposed solution; OK applies these test cases and summarizes the results. However, students are not able to view these results unless they first validate their understanding of each test case. As a result, they cannot check their solution until after they demonstrate that they are at least attempting to solve the right problem.

Through an interactive process, the student is presented with a series of short-answer questions that ask about the potential behaviors of their program. Each question corresponds to a "locked" test case. By responding with the correct intended behavior, they "unlock" the test case and can then apply it to their proposed solution.

This unlocking process is designed to encourage students to understand details of the problem specification fully before attempting to develop a solution. Automating this process gives students instant feedback about their interpretation of each problem's description, feedback that is intended to reduce the amount of time that students spend developing solutions for misinterpreted variants of a problem. We found that students overwhelmingly chose to validate their problem understanding using our system before attempting to solve each problem. In addition, 78% of students responded positively to the system in an end-of-semester survey, despite the fact that unlocking test cases is an extra step in the process of completing their projects.

Creating an unlocking process for an assignment requires only a small amount of additional effort beyond the creation of test cases for automatic assessment, but provides substantial time savings. When students more often arrive at correct interpretations of problems independently and have increased confidence in their interpretation, they ask fewer clarificatory questions. For example, forum questions about one problem

L@S 2015, March 14–18, 2015, Vancouver, BC, Canada.

ACM 978-1-4503-3411-2/15/03 ... $15.00
http://dx.doi.org/10.1145/2724660.2724679

[1] OK is hosted on `https://okpy.org`. Open-source development is hosted on `https://github.com/Cal-CS-61A-Staff/ok`.

in our course decreased by 79% in the semester after we deployed OK. Adding the unlocking process to all major assignments helped us scale our introductory computer science course[2] from 500 to 1300 students per semester in three years.

RELATED WORK

The sheer scale of massive open online courses (MOOCs) has increased demand for tools and techniques that automate aspects of course administration. Automated assessment now extends beyond multiple-choice to include free-response short answers [1]. Assessment at scale has also successfully leveraged the large pool of enrolled students as a source of human judgment [9]. These innovations in assessment allow for broader flexibility in assignment design.

The purpose of assignments is not only to assess progress, but also to support learning, and assessment is integral to the learning process [2]. Assignments are useful to students in validating their understanding [13]. In addition, assignments and their assessments highlight and even influence learning goals of a course [5]. Open-ended assignments that require integration of past concepts with current concepts are particularly effective for learning and retention [8]. This collection of prior research highlights the value of designing assessment systems that support learning directly. When learning to solve technical problems, a student will ideally receive feedback throughout the problem solving process.

Considerable prior research has focused on automated assessment of programming-based assignments, which are typically found in computer science courses, but increasingly appear in other technical fields. Automated assessment of programming assignments was first suggested more than 50 years ago [6]. In a recent survey, Ihantola *et al.* identify a wide range of recent features in automated program assessment systems, including test case construction assistance, submission management, automated scoring, and security features [7]. Noticeably absent from these features is any attempt to specifically provide feedback about problem understanding, rather than student solutions.

To scale a course to a massive size, instructors often rely on peer forums to assist students in understanding assignments. Indeed, online forums have been hailed as an essential component of a massive course [12]. However, the forums in massive courses often raise their own challenges. Mak *et al.* describe how large differences in skill between novices and experienced students can lead to friction and frustration among students, often requiring instructor moderation to maintain a productive discussion [12]. They found that half of students stopped participating in online forums, primarily because of "unacceptable behavior" from other participants. In our experience, peer forums can be particularly hostile to students who ask clarificatory questions that are similar to previous questions. Online forums certainly have a central role to play in massive courses, but the need for problem clarification may be addressed more effectively using an automated system that allows students to reach understanding independently or in small groups.

[2] `https://cs61a.org`

SOLVING TECHNICAL PROBLEMS

Technical problems appear in many disciplines and many varieties, but we focus here on problems that have descriptions written in natural language (e.g., English), but have solutions expressed in a formal language. For such problems, validating solutions is typically straightforward: the formal language is interpreted by a computer, which judges correctness based on a series of criteria. While programming problems are the focus of this paper, our approach could also apply to other assignments that use formal languages, such as constructing a logical proof.

Solving technical problems is an acquired skill that challenges many students. We seek to teach our students an effective multi-stage problem-solving process, and the OK system is designed to encourage and support that process.

Step 1: Problem Understanding

The first step in solving a technical problem is to understand the required properties of the solution from the problem description. Beyond just reading the prose, problem understanding involves identifying corner cases, clarifying details, and considering interactions with existing implementations.

For programming problems that involve implementing a function, the textbook *How to Design Programs* suggests first writing down a data representation, then writing down the purpose of the function, then illustrating its use with examples, then taking inventory of available inputs to the function [3]. All of these activities are examples of problem understanding, and they are useful activities that occur before any attempt at solving the problem.

In our experience, students often attempt to forego this step and focus immediately on writing a solution. With only a solution assessment tool at their disposal, they may attempt to reach a satisfactory solution through a process of trial-and-error. By contrast, including an automated assessment tool that explicitly targets this stage of the process not only encourages problem understanding before solution attempts, but also highlights problem understanding as a skill to be learned as part of a technical course.

Step 2: Planning

This second step involves developing a solution strategy, which may involve stating a natural language description or choosing among possible approaches. At this stage, the problem is understood, but the mechanics of how to solve it are still undetermined.

Assessment can also have a role to play here. The approach to designing a general solution can often be inspired by certain illustrative examples. The OK system asks students to predict the intended behavior of a program for specific cases. In doing so, they must work by hand through the process of computing some desired result. In our experience, selecting examples judiciously can provide automated guidance toward a solution.

We do not attempt to evaluate this effect, as the activity of solution planning is particularly difficult to measure. Indeed,

past research has shown that hinting toward solutions does not necessarily improve student performance [10].

Step 3: Implementation

Finally, the plan must be expressed in the formal language required to satisfy the original problem. In programming assignments, this step requires detailed knowledge of a programming language.

If steps 1 and 2 are carried out successfully, automatic assessment of solutions is often sufficient to support students in this final stage. Small issues with language syntax or boundary conditions can be identified by applying a suite of well-designed test cases. The OK system also provides this support, as do many other automated assessment tools.

Solution assessment can also provide feedback about earlier stages, as students may identify issues with their solution plan or problem understanding based on the failure of a solution test case. However, this feedback is only offered after the student has invested significant time developing a proposed solution. Moreover, students who misunderstand a problem may fail to attribute a test case failure to this earlier misunderstanding. Therefore, we hypothesized that providing earlier feedback would be beneficial to students.

EXAMPLE PROBLEMS

The remainder of this paper describes the details and empirical evaluation of the OK system, focusing on its application to a single programming assignment that was used in our course before and after deploying OK.

The assignment involves implementing a simulator and several strategies for a jeopardy dice game called Hog [11]. In this game, players alternate turns in which they select some number of dice and roll them to score points. If at least one of the dice outcomes is 1, then the current player scores only 1 point that turn. Otherwise, the player scores the sum of the dice outcomes.

The first problem in the assignment is to implement a function called `roll_dice` that computes a player's score on a single turn, using the player's choice of how many dice to roll and a random function that simulates each roll. The description of the problem includes a specification of exactly how many times to simulate a dice roll, a detail often overlooked by students.

The second problem we consider in our evaluation, which appears near the end of the project, is to implement a function called `swap_strategy`. This function encodes a strategy for playing the game by specifying how many dice to choose under various score conditions. The strategy can be illustrated effectively using a handful of test cases.

This assignment is implemented in Python. Students are given less than two weeks to complete all problems, which are released together during the second week of the course. The Appendix contains further details of the parts of the assignment used in our evaluations.

```
=====================================================================
Assignment: Project 1: Hog
OK, version v1.3.10
=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Unlocking tests

At each "? ", type what you would expect the output to be.
Type exit() to quit

---------------------------------------------------------------------
Question 1 > Suite 1 > Case 1
(cases remaining: 4)

>>> roll_dice(2, make_test_dice(4, 6, 1))
? 3
-- Not quite. Try again! --

? 10
-- OK! --

---------------------------------------------------------------------
```

Figure 1. An unlocking session where the student wishes to unlock question 1, initially provides an incorrect response, then provides a correct response.

SYSTEM DESIGN

The design of the OK assessment system includes a student interface, an instructor interface, and a mechanism for test-case locking and unlocking.

Interface for Students

The OK system integrates two kinds of assessments in a single interactive text-based interface presented in a terminal window and invoked from the command line.[3] Initially, test cases are locked, and the only available assessment focuses on problem understanding. As a byproduct of the understanding assessment, test cases are unlocked and become available for solution assessment.

Students start an assessment by selecting a problem. They are free to run the assessments at any time, before or after they attempt to implement a solution. When developing the system, we hoped that students would run the understanding assessment before attempting to solve the problem; observational results in the next section show that this order does indeed hold for most students.

Figure 1 contains a screenshot of this initial assessment, which we call an *unlocking session*. The system presents the student with a series of short-answer questions. Most of these questions consist of an expression to be evaluated and are answered by typing the correct value of the expression according to the problem description. In this example, the intended result is 10, the score received in a game of Hog by rolling a 4 and a 6. If the student is able to correctly state the intended result, then the interface advances to the next question. When students respond incorrectly, they are prompted to try again. This phase of the assessment is unaffected by whether or not the student has attempted a solution.

[3]The system could easily be adapted to the web, but one of our course goals is to build students' competence with command-line interfaces.

```
=====================================================================
Assignment: Project 1: Hog
OK, version v1.3.10
=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Running tests

---------------------------------------------------------------------
Question 1 > Suite 1 > Case 1

>>> from hog import *
>>> roll_dice(2, make_test_dice(4, 6, 1))

# Error: expected
#     10
# but got
#

---------------------------------------------------------------------
Question 1
    Passed: 0
    Failed: 1
    Locked: 3
[k.........] 0.0% passed

There are still locked tests! Use the -u option to unlock them

---------------------------------------------------------------------
Test summary
    Passed: 0
    Failed: 1
    Locked: 3
[k.........] 0.0% passed
```

Figure 2. While there are still some tests that are locked, the OK system applies only the unlocked test cases for each question, but also reports the number of test cases that are still locked.

```
=====================================================================
Assignment: Project 1: Hog
OK, version v1.3.10
=====================================================================

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Unlocking tests

At each "? ", type what you would expect the output to be.
Type exit() to quit

---------------------------------------------------------------------
Question 5 > Suite 1 > Case 1
(cases remaining: 12)

Q: The variables score0 and score1 are the scores for both
players. Under what conditions should the game continue?
Choose the number of the correct choice:
    0) While score0 and score1 are both less than goal
    1) While at least one of score0 or score1 is less than goal
    2) While score0 is less than goal
    3) While score1 is less than goal
? 3
-- Not quite. Try again! --

Choose the number of the correct choice:
    0) While score0 and score1 are both less than goal
    1) While at least one of score0 or score1 is less than goal
    2) While score0 is less than goal
    3) While score1 is less than goal
? 0
-- OK! --

---------------------------------------------------------------------
```

Figure 3. For this test case, it is rather difficult to predict the intended behavior without writing a program to do so. Therefore, we include conceptual questions in order to check the student's understanding, rather than requiring a tedious amount of computation by hand.

Each correctly answered question unlocks a corresponding test case. In Figure 1, the test case would evaluate a call to `roll_dice` using the student's implementation and report whether or not the result matched the intended result of 10.

After unlocking one or more test cases for a question, the student can run a solution assessment, as shown in Figure 2. This phase has the typical behavior of an automated program assessment tool. The student's solution is loaded automatically from the current directory. Test cases are executed in order. Each test case failure generates a description of the problem, typically comparing the expected and observed results.

Interactive questions to assess understanding are not limited to evaluating Python expressions. Some problems are sufficiently complex so evaluating an example by hand is arduous or unfeasible. Despite this constraint, the system can validate problem understanding by asking conceptual multiple-choice or short-answer questions. An example of a conceptual question is shown in Figure 3. In this example, rather than supplying the value of an expression, the student selects an answer choice. To dissuade random guessing, the order of answer choices is randomized.

Interface for Instructors

Instructors create test cases for both types of assessment: problem understanding and solution verification. However, the unlock-then-apply structure allows the same test case to be used for both phases of assessment. As a result, the additional effort required from the instructor to support both kinds of assessment is minimal, beyond the effort required just for solution assessment.

The only additional work needed is to construct the conceptual questions that substitute for particularly complex test cases. For the Hog project, we created 58 test cases and 5 conceptual questions. Thus, the only additional instructional effort came from designing the 5 conceptual questions and deciding which test cases were too complicated to predict by hand. In OK, test cases are specified as string pairs indicating expression and value, along with a small amount of set-up and comparison code.

Unlocking Mechanism

The design of the underlying unlocking mechanism was chosen to satisfy several constraints. First, we wanted to make sure that the unlocking phase was not trivial to circumvent by students. Second, we wanted to ensure that the test cases were correct once unlocked. In this way, all tests remain reliable, so students don't have to determine whether or not an issue lies within their implementation or the test itself. Our design makes it very difficult to extract the answers from the codebase while making sure that using the resulting answers only creates valid test cases for their solutions. In addition to the above, we designed our system to run locally on each student's computer, so that students can proceed through the project without a network connection.

Our mechanism relies on hashing in order to meet these constraints. Specifically, the staff knows every question Q (an input to a test case) and the answer to each question, A (the

output to a test case). Let H be an irreversible hash function. We chose a hash-based message authentication code (MAC) [4]. The instructor distributes both Q and $H(A)$ to the students, along with the OK system which includes an implementation of H. When the student provides a guess G for a question, OK computes $H(G)$. If $H(G) = H(A)$, then we assume $G = A$ and use G as the test-case output for Q.

Our hash-based unlocking mechanism provides a fully-local way to check student answers for our questions. In addition, the mechanism has proven resilient to any method of unlocking test cases automatically without knowing the correct responses, as far as we have been able to observe.

EVALUATION

We sought to measure the effectiveness of our approach on two dimensions: whether or not the instructor workload actually decreased and whether or not students used the unlocking system and found it valuable. In order to measure whether or not the instructor workload decreased, we monitored the class forum and kept track of the number of questions asked.

We measured our students' experience in two ways. First, we checked to see whether or not students were unlocking in the correct stage of the problem solving process—at the beginning. Then, we looked at the types of questions that were asked on the class message board. Finally, we asked the students how helpful they found the unlocking process in an online survey.

Usage Pattern

Students had the option to attempt a solution before assessing their understanding or vice versa. The OK system allowed us to measure this ordering directly. With permission from students, the system would send both their solution progress and unlocking session information to an instructional server. We received data on whether the student was unlocking a question or not, what question they were unlocking, and which problems they had attempted to solve from the assignment.

Our analysis was complicated by the fact that not all unlocking sessions were reported to the server. Students had the option of unlocking tests for all problems at once; if they did so, we collected no data. Students could also choose not to share their session data with us. In addition, students may have started to implement solutions in a way that we could not record. To determine when a student started a question, we looked for edits to the files submitted to our server by OK. It is possible that students began solutions in another external file, in which case we would have been unable to detect their progress. However, we believe that very few students did so, because of convenience. When students edited the files we tracked, they could easily check their solutions using OK.

For all of these events, we recorded the time when they happened, which allowed us to assess whether or not students unlocked test cases before attempting to solve problems.

In this class, we had 1287 students who received grades for this project and 1272 students who sent usage statistics to our server for at least one problem—the completion rate is lower than these numbers suggest because some students dropped

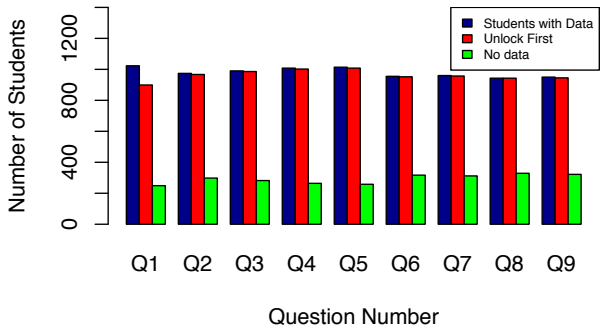


Figure 4. The total number of students who provided unlocking data for each question, as well as the number of students that interacted with the autograder, but did not provide data for that question.

the class during this project. Usage statistics were not collected when students opted out or had a slow network connection; attempts to contact our server were limited to less than a second.

Figure 4 shows the fraction of students for which we have data who unlocked test cases for a problem, verifying their understanding, before attempting to solve the problem. For Question 1, 88% of students unlocked the test cases before they started writing code. That fraction increased to 99.5% by the end of the project, where almost all students unlocked test cases before attempting a solution.

While these numbers are encouraging, we also note the large number of students with no unlocking data, and we believe that some of these students might also not follow our expected behavior. However, even a pessimistic assumption where all unknown students fail to follow our expected behavior shows that 70% to 79% of students meet our expectations, which is still a strong majority.

The usage statistics show students are using the unlocking mechanism at the beginning of the problem solving process, implying that it is serving its true purpose.

Class Forum

In our course, we use the Piazza online forum. A student can make a post with his or her question and receive a response from one of the staff members or a fellow student, usually in under an hour. The average response time in our course is about 11 minutes.

Due to the short response times, different students often make similar posts because they have the same misunderstanding about the project specifications. One common use of Piazza is for students to post their autograder test failures and ask for debugging help. If the initial response does not fully answer their question, then they are able to post a follow-up question.

To evaluate changes in student behavior, we counted and classified posts about two problems from the Hog assignment described earlier. In Fall 2013, students received test cases

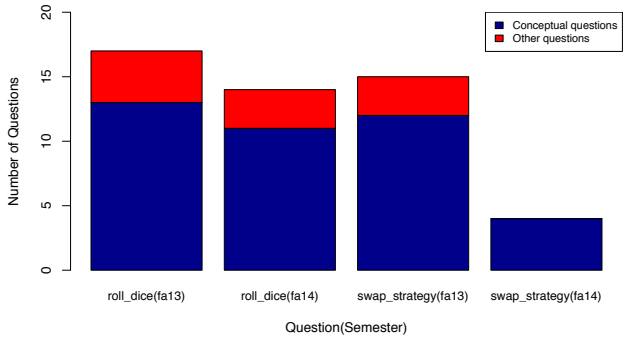


Figure 5. Number of forum questions posted about each problem, by semester. Each bar shows the total number of questions on Piazza, classified as conceptual questions (misunderstanding or clarification of the problem) or other kinds of questions.

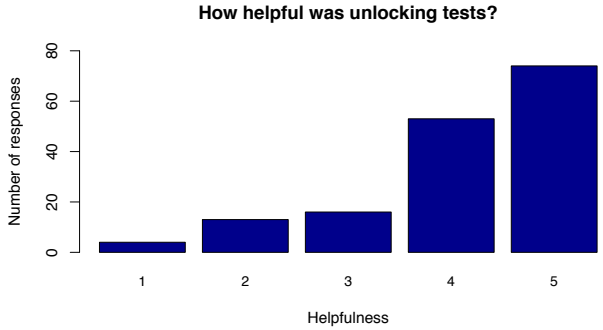


Figure 6. Survey results for the question "How helpful do you find unlocking test cases to be in improving your understanding of the project?" Answers are based on a scale from 1 ("completely unhelpful") to 5 ("very helpful").

embedded in a source code file, but they were not explicitly encouraged to read those tests. In Fall 2014, similar test cases were provided, but students had to unlock them before using them to check correctness. We focused our analysis on two problems that students historically had trouble understanding, `roll_dice` and `swap_strategy`. We classified all forum posts about these two problems as either conceptual questions (misunderstanding or clarification of the problem) or other questions (usually issues with implementation).

To find all of the relevant posts, we looked at all forum posts that were posted after the project was released until when the project was due. Two of this paper's authors independently went through all of the posts related to the project for each problem. A post was only classified for that problem if it specifically asked about that problem, not whether the final issue was actually in that problem. Both authors independently decided whether or not the question was a misunderstanding or clarification of the problem description (or some other type of question) based on the response and any follow-up questions that were asked. After finishing classification, the authors compared their judgments for each question and resolved discrepancies through discussion.

In the posts from Fall 2014, we classified questions that asked for help unlocking the tests as conceptual questions. Since students' Piazza questions about these two problems were usually a result of misunderstanding the prompt, we expected to see fewer questions asked about them.

In Fall 2013, we had 1011 students with a submission for this project. In Fall 2014, we had 1287 students. Despite this 27% increase in students, Figure 5 shows that the number of forum posts about these questions decreased substantially. In particular, Piazza questions about `swap_strategy` dropped significantly after unlocking was introduced. In addition, we have observed that many forum questions are repeated in other settings as well (discussion sections and office hours). These results imply that significant instructor time was saved through other venues.

The ratio of conceptual questions versus other questions remained fairly constant between semesters. However, in Fall 2014, the overwhelming majority of the conceptual Piazza questions were asked by students that were having trouble unlocking the tests, rather than by students who unlocked the tests and implemented a function that did not match our specifications. This trend imples that students who are able to unlock our tests understand what the problem statement was asking. In addition, since unlocking a test case is not implementation specific, these questions can be asked to other classmates and answered by instructors in small groups as well, without students sharing their code.

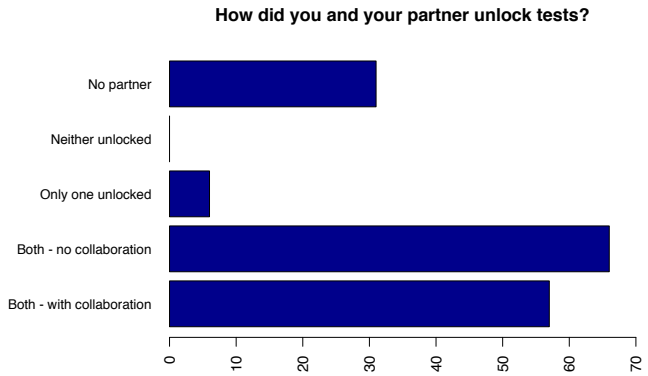
Survey Results

In the Spring 2014 offering of the course, we had unlocking partially implemented for all projects. While many simple questions were locked, we had not written any conceptual questions. After the completion of four projects that used the partially implemented unlocking mechanism, we issued a student survey to measure the effectiveness of the unlocking mechanism and to garner suggestions from students. Due to the partial implementation, these students were able to compare how much unlocking helped.

The results shown in Figure 6 indicate the unlocking mechanism reinforced students' conceptual understanding of project questions. Students were asked to answer the question "How helpful do you find unlocking test cases to be in improving your understanding of the project?" from a scale of 1 ("completely unhelpful") to 5 ("very helpful"). Out of 160 responses, 74 (46%) students chose a score of 5 and 53 (33%) students chose a score of 4.

In addition, we were also interested in how students interacted with unlocking when working in groups. Two of our four projects allowed students to work in pairs, the rationale being that project partners can provide each other with different insights on problem solving. In the context of unlocking, partners would ideally complete the unlocking procedure together and address any misunderstandings in the process.

Responses to the question "If you worked with a partner, how did you and your partner unlock tests?" are shown in Fig-



Both my partner and I unlocked tests with each other's help	**57**	36%
Both my partner and I unlocked tests without each other's help	**66**	41%
Only one partner in our group unlocked tests	**6**	4%
Neither of us unlocked tests	**0**	0%
I did not work with a partner	**31**	19%

Figure 7. Survey results for the question "If you worked with a partner, how did you and your partner unlock tests?"

ure 7. Out of 160 responses, only 6 students (4%) reported that "one partner in our group unlocked tests." Few students progressed through the projects without interacting with the unlocking mechanism. In order to simply gain access to the test case, the fastest way to do so would be to only have one partner be responsible for all of the unlocking. Since both partners went through the unlocking process, this suggests that a majority of students made a conscious decision to reinforce their conceptual understanding of the project prompts before advancing to program implementation. Of the project groups where both partners unlocked, 57 students (36%) "unlocked tests with [their partner's] help," while 66 students (41%) "unlocked tests without [their partner's] help."

We also included the option for students to provide free-form responses. The responses generally verified that unlocking "helped [students] work through the thought process" and "[got students] to understand what [they] need to do first." We had one project that did not use unlocking at all; one student suggested "add[ing] this feature" to that project, as unlocking aided in "understand[ing] the question...much better, especially since some of the instructions can be confusing or unclear." In addition to supplementing instructor guidance, the unlocking mechanism can also clarify unexpected ambiguities in assignment instructions.

One student noted that "unlock[ing does not] help with implementation" and instead is better suited for students who are "totally lost." While the comment may have been intended as a complaint, it actually reinforces the distinction between unlocking for assessing conceptual understanding and applying test cases for assessing implementation correctness.

CONCLUSION

As class sizes increase, automated feedback becomes increasingly important to both students and instructors. Our work complements prior efforts to assess solutions by also assessing problem understanding. This additional feedback encourages an effective approach to solving technical problems by providing feedback early in the problem solving process in order to make sure that students do not waste their time trying to solve an incorrect interpretation of the given problem.

Our approach has multiple benefits to both students and instructors. First of all, instructional effort is reduced while feedback is given instantaneously and consistently. Students save time by avoiding incorrect solutions based on misunderstandings, and instructors avoid helping students debug implementations that are based on misinterpretation.

Through our evaluations, we found this additional form of automated assessment to have positive effects. The number of questions instructors were asked decreased, and students stated that unlocking test cases was helpful. Perhaps most encouraging of all, the usage patterns we observed suggested that our students were adopting an effective approach to solving technical problems.

APPENDIX

The Game of Hog

The game of Hog is our first project and the project that we used during our evaluation of the unlocking mechanism. The rules were described to the students as follows:

In Hog, two players alternate turns trying to reach 100 points first. On each turn, the current player chooses some number of dice to roll, up to 10. Her turn score is the sum of the dice outcomes, unless any of the dice come up a 1, in which case the score for her turn is only 1 point (the **Pig out** rule).

To spice up the game, we will play with some special rules:

1. **Free bacon.** If a player chooses to roll zero dice, she scores one more than the largest digit in her opponent's score. For example, if Player 1 has 42 points, Player 0 gains 1 + max(4, 2) = 5 points by rolling zero dice. If Player 1 has 48 points, Player 0 gains 1 + max(4, 8) = 9 points.

2. **Hog wild.** If the sum of both players' total scores is a multiple of seven (e.g., 14, 21, 35), then the current player rolls four-sided dice instead of the usual six-sided dice.

3. **Swine swap.** If at the end of a turn one of the player's total score is exactly double the other's, then the players swap total scores. Example 1: Player 0 has 20 points and Player 1 has 5; it is Player 1's turn. She scores 5 more, bringing her total to 10. The players swap scores: Player 0 now has 10 points and Player 1 has 20. It is now Player 0's turn. Example 2: Player 0 has 90 points and Player 1 has 50; it is Player 0's turn. She scores 10 more, bringing her total to 100. The players swap scores, and Player 1 wins the game 100 to 50.

The project prompt and rules are the same for the two semesters that we are comparing.

Roll Dice Function

The `roll_dice` function is a problem about simulating simultaneous dice rolls. The exact prompt that we gave them is as follows:

Implement the `roll_dice` function in hog.py, which returns the number of points scored by rolling a fixed positive number of dice: either the sum of the dice or 1. To obtain a single outcome of a dice roll, call dice(). You should call this function exactly `num_rolls` times in your implementation. The only rule you need to consider for this problem is *Pig out*.

In Fall 2014, since we had the unlocking mechanism, we added a few questions in order to aid understanding of our testing suite. Specifically, the test scenarios that we checked in Fall 2013 were the following:

```
>>> counted_dice = make_test_dice(4, 1, 2)
>>> roll_dice(2, make_test_dice(4, 6, 1))
10
>>> roll_dice(3, make_test_dice(4, 6, 1))
1
>>> roll_dice(3, make_test_dice(1, 2, 3))
1
>>> roll_dice(3, counted_dice)
1
>>> roll_dice(1, counted_dice)
4
>>> test_dice = make_test_dice(4,2,3,3,4,1)
>>> roll_dice(5, test_dice)
16
>>> roll_dice(2, make_test_dice(1))
1
```

In addition, in Fall 2014, we added some more scenarios in order to help students understand the Fall 2013 tests that were only used for unlocking purposes:

```
>>> test_dice = make_test_dice(4, 1, 2)
>>> test_dice()
4
>>> test_dice() # Second call
1
>>> test_dice() # Third call
2
>>> test_dice() # Fourth call
4
```

Swap Strategy Function

The `swap_strategy` function asks students to implement a strategy that takes advantage of the Swine swap rule. The exact prompt that we gave them is as follows:

A strategy can also take advantage of the Swine swap rule. Implement `swap_strategy`, which

1. Rolls 0 if it would cause a beneficial swap that gains points.
2. Rolls `BASELINE_NUM_ROLLS` if rolling 0 would cause a harmful swap that loses points.
3. If rolling 0 would not cause a swap, then do so if it would give at least `BACON_MARGIN` points and roll `BASELINE_NUM_ROLLS` otherwise.

In Fall 2013, we had the following scenarios:

```
>>> swap_strategy(23, 60)
0
>>> swap_strategy(27, 17)
5
>>> swap_strategy(50, 80)
0
>>> swap_strategy(12, 12)
5
>>> swap_strategy(12, 34)
0
>>> swap_strategy(8, 9)
4
>>> swap_strategy(32, 43)
0
>>> swap_strategy(20, 32)
4
```

In Fall 2014, we had the following scenarios:

```
>>> swap_strategy(23, 60)
0
>>> swap_strategy(27, 17)
5
>>> swap_strategy(50, 80)
0
>>> swap_strategy(12, 12)
5
>>> swap_strategy(15, 34, 5, 4)
0
>>> swap_strategy(8, 9, 5, 4)
4
>>> swap_strategy(32, 40, 5, 4)
0
>>> swap_strategy(20, 32, 5, 4)
4
```

For all of the above tests, students had to unlock them in order to clarify their understanding.

ACKNOWLEDGMENTS

This work was supported by a Google Research Cloud Credits Award.

REFERENCES

1. Brooks, M., Basu, S., Jacobs, C., and Vanderwende, L. Divide and correct: Using clusters to grade short answers at scale. In *Proceedings of the First ACM Conference on Learning @ Scale Conference*, L@S '14, ACM (New York, NY, USA, 2014), 89–98.
2. Earl, L. M. *Assessment As Learning: Using Classroom Assessment to Maximize Student Learning*. Corwin, 2003.
3. Felleisen, M., Findler, R. B., Flatt, M., and Krishnamurthi, S. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, Cambridge, MA, USA, 2001.
4. H. Krawczyk, M. Bellare, R. C. HMAC: Keyed-Hashing for Message Authentication. RFC 2104.

5. Heywood, J. *Assessment in higher education: Student learning, teaching, programmes and institutions.* 2000.
6. Hollingsworth, J. Automatic graders for programming classes. *Commun. ACM 3*, 10 (Oct. 1960), 528–529.
7. Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, Koli Calling '10, ACM (New York, NY, USA, 2010), 86–93.
8. Karpicke, J. D., and Roediger, H. L. The critical importance of retrieval for learning. *Science 319*, 5865 (2008), 966–968.
9. Kulkarni, C. E., Socher, R., Bernstein, M. S., and Klemmer, S. R. Scaling short-answer grading by combining peer assessment with algorithmic scoring. In *Proceedings of the First ACM Conference on Learning @ Scale Conference*, L@S '14, ACM (New York, NY, USA, 2014), 99–108.
10. O'Rourke, E., Ballweber, C., and Popovií, Z. Hint systems may negatively impact performance in educational games. In *Proceedings of the First ACM Conference on Learning @ Scale Conference*, L@S '14, ACM (New York, NY, USA, 2014), 51–60.
11. Parlante, N., Zelenski, J., Dodds, Z., Vonnegut, W., Malan, D. J., Murtagh, T. P., Neller, T. W., Sherriff, M., and Zingaro, D. Nifty assignments. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, ACM (New York, NY, USA, 2010), 478–479.
12. Sui Fai John Mak, Roy Williams, J. M. Blogs and forums as communication and learning tools in a MOOC. In *Proceedings of the Seventh International Conference on Networked Learning 2010* (2010), 275–284.
13. Thorpe, M. Assessment and third generation distance education. *Distance Education 19*, 2 (1998), 265–286.