Providing Meaningful Feedback for Autograding of Programming Assignments

Georgiana Haldeman
Rutgers University
mgh80@cs.rutgers.edu

Andrew Tjang
Rutgers University
atjang@cs.rutgers.edu

Monica Babeş-Vroman
Rutgers University
babes@cs.rutgers.com

Stephen Bartos
Rutgers University
scb167@scarletmail.rutgers.edu

Jay Shah
Rutgers University
js2279@scarletmail.rutgers.edu

Danielle Yucht
Rutgers University
dry11@scarletmail.rutgers.edu

Thu D. Nguyen
Rutgers University
tdnguyen@cs.rutgers.edu

ABSTRACT

Autograding systems are increasingly being deployed to meet the challenge of teaching programming at scale. We propose a methodology for extending autograders to provide meaningful feedback for incorrect programs. Our methodology starts with the instructor identifying the concepts and skills important to each programming assignment, designing the assignment, and designing a comprehensive test suite. Tests are then applied to code submissions to learn classes of common errors and produce classifiers to automatically categorize errors in future submissions. The instructor maps the errors to concepts and skills and writes hints to help students find their misconceptions and mistakes. We have applied the methodology to two assignments from our Introduction to Computer Science course. We used submissions from one semester of the class to build classifiers and write hints for observed common errors. We manually validated the automatic error categorization and potential usefulness of the hints using submissions from a second semester. We found that the hints given for erroneous submissions should be helpful for 96% or more of the cases. Based on these promising results, we have deployed our hints and are currently collecting submissions and feedback from students and instructors.

KEYWORDS

Autograding, concepts/skills-based hints, error categorization

ACM Reference format:
Georgiana Haldeman, Andrew Tjang, Monica Babeş-Vroman, Stephen Bartos, Jay Shah, Danielle Yucht, and Thu D. Nguyen. 2018. Providing Meaningful Feedback for Autograding of Programming Assignments. In *Proceedings of SIGCSE '18: The 49th ACM Technical Symposium on Computing Science Education, Baltimore , MD, USA, February 21–24, 2018 (SIGCSE '18),* 6 pages.
https://doi.org/10.1145/3159450.3159502

SIGCSE '18, February 21–24, 2018, Baltimore , MD, USA

ACM ISBN 978-1-4503-5103-4/18/02…$15.00
https://doi.org/10.1145/3159450.3159502

1 INTRODUCTION

The current demand for computing professionals is unprecedented, with a short supply in the US, and the gap between demand and supply is predicted to increase even more in the next few years [13]. Correspondingly, universities are seeing tremendous enrollment growths in computing classes and are faced with the challenge of teaching programming at scale [4].

According to Ambrose et al., *"goal-directed practice coupled with targeted feedback are critical to learning"* [1]. Programming assignments are the tools of choice for providing students with hands-on practice in many computing classes. However, as autograding systems are deployed to scale the grading of programming assignments in increasingly large classes, providing meaningful feedback remains an open problem. Many deployed autograding systems do not provide feedback. Even when feedback is available, the feedback provided does not assist students in correcting errors [10] and does not address underlying misconceptions. A recent work seeks to address these issues using a hybrid approach that combines program synthesis and instructor code review and feedback generation [7].

In this paper, we propose a methodology that is similar in spirit to the approach in [7], but focuses on linking errors to concepts and skills central to a programming assignment and does not depend on compilation tools. Rather, our approach depends on collecting and analyzing assignment submissions to generate hints that can be used in future semesters.[1] More specifically, when designing an assignment, we propose that the instructor explicitly define the set of concepts and skills, called the *knowledge map*, that students need to master to complete the assignment. After the assignment has been written, a comprehensive test suite should be developed with the goal of teasing out misunderstandings of the knowledge map along with correctness testing for grading.

Next, the assignment is released and student submissions are collected. The instructor runs the test suite against the submissions and manually inspects sets of submissions with the same outcome patterns (called signatures) to identify errors, refining the test suite as needed or desired. Finally, the instructor maps the errors to specific items in the knowledge map and generates hints to guide

[1] Supporting the evolution of assignments while still making use of collected data is an important extension that we intend to explore in the near future.

the students toward reviewing and improving their understanding and mastery of the applicable concepts and skills. The observed signatures can be used to produce a classifier that automatically categorize future erroneous submissions, allowing the autograding system to provide the corresponding hints as meaningful feedback.

The above methodology encapsulates two key ideas: (1) the inspection of a set of submissions with the same signatures for a comprehensive, well designed suite of tests can help the instructor identify high-level logical errors, and (2) the mapping of errors to concepts and skills in the knowledge map can aid in the generation of hints that direct students to conceptually reconsider their code, rather than making local, code specific suggestions that can lead to highly convoluted solutions (e.g., many unnecessary nested conditional statements). Understanding common errors in light of the knowledge map can also aid the instructor in adjusting classroom teaching in the current and future offerings of the course.

We applied our proposed methodology to two assignments in our Introduction to Computer Science course. We collected large numbers of submissions for the two assignments across two semesters, Spring 2016 and Spring 2017. We designed test suites for the assignments and used submissions from Spring 2016 to learn classes of common errors, produce classifiers for automatic error categorization of future submissions, and write hints. We then used the classifiers to attach hints to erroneous submissions from Spring 2017. Four of the authors manually reviewed the results of this experiment, and found that over 91% and over 87% of the hints for the two assignments, respectively, fully captured the errors in the corresponding submissions. These percentages rise to over 96% when we count hints that partially captured the errors. Based on these promising results, we have deployed the error categorization and corresponding hints for the two assignments in the Fall 2017 semester, and are currently collecting submissions and feedback from students and instructors.

2 RELATED WORK

Our work explores the extension of autograding systems to provide feedback to students. Web-CAT [5] and Autolab [12] are two autograding tools that we have used in our Introduction to Computer Science course. Web-CAT includes a library that allows a hint to be attached to each test case. When a submission fails a test, the associated hint is returned as feedback. In our experience, it is difficult to balance between writing hints that are overly specific and overly vague based on the result of one test case. Thus, our approach uses the results from an entire test suite for identifying common errors and providing hints.

One research effort for automated grading has focused on defining a problem-independent grammar of features, and by using a supervised learning algorithm trained on teacher graded examples to assign grades to new student submissions [18]. This work is related but focuses on grading rather than providing feedback.

Commonly, the generation of feedback is a two step process: first, the code is assessed to identify mistakes or bad practices; then, appropriate feedback is generated based on the assessment from the first step. Methods for assessing students' code are unit testing, stylistic checking [10], property-based testing [2], execution traces [14], and program synthesis [19]. Automated generation of feedback often relies on the assumption that students in large classes share common errors [6]. Student submissions are clustered based on failed test cases, compiler errors or runtime exceptions [6], stylistic checking [17], AST tree edit distance [9], probabilistic semantic equivalence [15] or program synthesis [3]. Our categorization approach is based on testing and manual review of submitted code.

Providing personalized feedback in Intelligent Tutoring Systems (ITS) is done by using rule-based and constraint-based methods [8]. However, this approach does not scale well for assignments that have multiple solutions (classes 2 and 3 in [11]). Data-driven procedures improve existing methods. One approach [16], for example, uses the AST distance to guide students from an erroneous submission to a correct solution. Their work is complementary to our work. Another recent effort focuses on designing a system that propagates the feedback of the teacher on one submission to the rest of the submissions that need the same code fixes by using program synthesis [7]. We already discussed how our system is different in Section 1.

3 CONCEPT AND SKILLS BASED FEEDBACK GENERATION FRAMEWORK (CSF^2)

We propose the Concepts and Skills based Feedback Generation Framework (CSF^2) for designing programming assignments based on the concepts and skills that students are required to master, and analyzing submissions with respect to these concepts and skills to generate hints for assisting students in correcting their errors. The approach is described as a sequence of steps, but the ordering can be modified as discussed in Section 6. The proposed steps are as follows.

- Step 1 Carefully consider the knowledge map (concepts and skills) that students need to master to complete the assignment.
- Step 2 Write the assignment with the knowledge map in mind.
- Step 3 Design a test suite to assess student submissions. Full path coverage of a reference solution is typically a good start (but will need expansion). Each test should output a code representing the outcome when it is run against a submission.
- Step 4 Release the assignment and collect student submissions.
- Step 5 Automatically run the test suite against the collected submissions and group submissions into buckets based on the outcome signatures, where a signature is the concatenation of the codes output by all the tests in the test suite. Each signature may indicate one or more logical errors. Our hypothesis is that submissions with the same signatures are likely to have similar logical errors.
- Step 6 Manually inspect each bucket of submissions to see whether subsets of submissions have different logical errors. If yes, then refine the test suite to separate these subsets into different buckets. The knowledge map may also need to be refined (for example, to include a concept that has to be mastered for the assignment but was accidentally omitted in Step 1).
- Step 7 Repeat Step 5 and Step 6 as needed.
- Step 8 Map the errors identified for each bucket to concepts and skills in the knowledge map. Then, manually inspect and combine buckets of submissions with the same errors or

knowledge deficiencies. Our hypothesis is that mapping errors to the knowledge map will help instructors reduce the number of hints that need to be written, as well as write hints that provide conceptual guidance rather than very specific coding changes.

Step 9 Write a hint for each bucket. The outcome of Step 8 and Step 9 is a classifier, manually trained on past student submissions, that maps the outcome signature of a well designed test suite to meaningful hints.

Step 10 When the assignment is run again, the autograding system can use the classifier to automatically categorize errors and provide hints for submissions that fail one or more tests.

While the work in this paper is focused on generating meaningful autograding feedback, the above process is also useful in gaining a better understanding of students' errors and misconceptions. The instructor can use this information to improve the quality of classroom teaching, for example, in identifying concepts and skills that should be reinforced, as well as adjusting the teaching of the concepts and skills in the future.

As described above, CSF^2 is most useful when an assignment is reused over multiple offerings of a class, with previous submissions being used to generate and improve hints for subsequent uses of the assignment. Currently, CSF^2 does not directly support the evolution of an assignment (for example, changes to the assignment to circumvent cheating or to improve the assignment) across time (although we have successfully experimented with changing *TwoSmallest* to an isomorphic assignment while still making use of analysis results from previous submissions; see Section 6). This is an important challenge that we plan to address in the future.

Finally, while the manual inspection of assignments is labor intensive, it is possible to get help from advanced undergraduate students when CSF^2 is applied to early computing classes. We have successfully done this with our case studies: three of the authors are undergraduate students who helped with exactly this task.

4 CASE STUDIES

In this section, we describe the application of CSF^2 to two programming assignments in the Introduction to Computer Science course at Rutgers University. Even though CSF^2 proposes that an instructor starts with a knowledge map when designing an assignment, our case studies work with existing assignments because we already collected a large number of student submissions across several semesters. In essence, we reversed the ordering of Steps 1 and 2 in CSF^2, and extracted the knowledge maps from the existing assignments.

Steps 1 and 2: Extracting the Knowledge Maps

We studied two assignments, *PayFriend*, a class 2 assignment according to [11], which means that it has one solution strategy with multiple possible implementations, and *TwoSmallest*, a class 3 assignment, which means that it has multiple solution strategies. *PayFriend* asks the students to compute the fee associated with making an e-payment when given a tiered fee structure, with different fees for four payment ranges, requiring a good understanding of conditional statements. *TwoSmallest* asks the students to read a sequence of floating point values that starts and ends with a sentinel value and output the two smallest values in the sequence, requiring algorithmic thinking and understanding of initialization and loops (in addition to conditional statements).

Each assignment requires the solution to be implemented inside a method with a prescribed method signature. Students are also asked to submit code in a specific format, including predefined file and class names and to omit all `package` and `import` statements. Failure to follow all instructions results in compilation errors and zero credit.

When we started this research, *PayFriend* and *TwoSmallest* had already been designed and used several times. We worked with the lead instructor to determine the knowledge maps he had in mind when designing them. Some of the concepts and skills in the knowledge maps are shown in the right column of Table 2 below.

Table 1: Number of students and assignment submissions.

	Spring 2016	Spring 2017
Number of students	511	437
***PayFriend* Submissions**	1152	723
***TwoSmallest* Submissions**	1339	870

Step 3: Designing the Test Suites

We developed a reference solution for each assignment and designed 13 test cases for *PayFriend* and 20 for *TwoSmallest* that led to a full path coverage of the reference solutions. Next, we considered more challenging inputs, especially for novice programmers. For example, it is well known that many programming bugs involve the handling of boundary cases. Thus, for *PayFriend*, we designed test cases with input values close to the tier boundaries, as well as values from the middle of the tiers. This test design process was iterative (as discussed in Steps 5–7 of CSF^2). After all refinements were made, *PayFriend*'s test suite contained 20 test cases and *TwoSmallest*'s contained 30. In previous offerings of the assignments, *PayFriend* and *TwoSmallest* were graded using 10 and 7 test cases, respectively.

Step 4: Collecting Student Submissions

We collected student submissions for the two assignments during the Spring 2016 semester using Web-CAT [5] and during the Spring 2017 semester using Autolab [12]. Table 1 shows the number of students and submissions collected for each assignment during the two semesters. Each submission included anonymized student information, a time stamp, and code. In this study, we only looked at the submitted code; all submissions were anonymized by removing all information, including comments, other than the actual code. The submissions from Spring 2016 were used for Steps 5–8 of CSF^2. The submissions from Spring 2017 were used to evaluate our ability to correctly identify the most important errors associated with incorrect submissions, with the ultimate goal of providing meaningful hints. This was a partial evaluation of the effectiveness of Step 10.

Steps 5–7: Refining Tests and Partitioning Submissions

We generated a test outcome signature for each submission by running the test suite developed in Step 3 against the submitted code. Then, submissions with the same signature were grouped in

Table 2: Mapping of common errors to concepts and skills in the knowledge map.

Code	Error	Concept / Skill
Both assignments		
COMP	compilation errors	writing compilable code
INS	errors regarding the required format, e.g., incorrect filename	following instructions
IO	IO errors, e.g., wrong numbers or types of inputs/outputs	data representation / following instructions
INF	used infinite loops	control flow
PayFriend		
CF	output only in some branches	control flow
COND	used incorrect conditional statements	translating word problems into conditional statements
FORM	used incorrect calculation inside a range	translating word problems into formulas
TwoSmallest		
SEQ	read and processed incorrectly a sequence of values	data representation / following instructions
INIT	wrongly initialized min values	algorithmic thinking
UPDT	wrongly updated min values	algorithmic thinking

the same bucket. Each signature could indicate one or more logical problems. A bucket could be mapped to two or more independent errors with their own associated hints, but all the submissions in the same bucket needed to have the same errors so that a meaningful corresponding hint could be generated. If, for a given bucket, some of the submissions had one error while others had a different error, we refined the test suite to obtain a better partitioning.

We manually inspected the students' code in each bucket to identify the main reason for the code failing one or more tests. For buckets containing submissions with *different* knowledge deficiencies, we refined or extended our test suite to further partition the buckets. We iterated through Steps 5–7 once for *PayFriend* and several times, making small refinements each time, for *TwoSmallest*. We found that iterations with small refinements were easier to think about. By the end of the process, we added 7 additional test cases for *PayFriend* and 10 for *TwoSmallest*. The final test suites led to 109 non-empty buckets for *PayFriend* (using 20 tests) and 137 non-empty buckets for *TwoSmallest* (using 30 tests).

Steps 8 and 9: Combining Buckets and Generating Hints

Next, we manually mapped the main reason for code failure in each bucket to a deficiency in a concept or skill as shown in Table 2. As already mentioned, we found that many of the buckets were different manifestations of similar knowledge deficiencies. Thus, we grouped many of the buckets together, leading to 8 "super-buckets" for *PayFriend* and 7 for *TwoSmallest* (Table 3). Clustering student submissions for assignments in class 2 and 3 is a particularly challenging task because the solution space can be large. Since unit testing mostly focuses on the functionality of the code rather than the style, we were able to cluster stylistically different student submissions with the same logical error.

Finally, we wrote a hint for each bucket, and the hints have been deployed in Autolab for the Fall 2017 semester. We are still in the process of collecting data, including student feedback, to assess the impact of our hints on student learning.

5 RESULTS

In this section, we present the most relevant findings from our case studies. We first assess the accuracy of the automatic error categorization of submissions and whether the hints capture these errors. Then, we show examples of common errors and corresponding hints. We discuss how the outputs of the test suites help identify logical errors and how the knowledge maps help produce hints on a more conceptual level. We have improved some of the hints (over the ones deployed for Fall 2017) in the process of writing this paper, although we did not fundamentally change any of the hints.

5.1 Accuracy of Error Categorization and Hints

As described in Section 4, we used our test suites and submissions from Spring 2016 to manually generate an error classifier for each of *PayFriend* and *TwoSmallest*. We also wrote hints to be given as feedback from the autograder. We then used the same test suites, classifiers, and hints to identify, categorize, and attach a hint to each erroneous submission from Spring 2017.

Three of the authors manually reviewed the Spring 2017 submissions to assess the accuracy of the classifier and potential efficacy of the corresponding hints. These authors are currently enrolled in the undergraduate Computer Science program at Rutgers, and have previously taken the Introduction to Computer Science course. We believe that having done the assignments themselves while taking the course gave them a good perspective in this evaluation. Their task was to label each (erroneous submission, hint) pair as *Correct*, *Partially Correct*, or *Incorrect*. *Correct* meant that the automatic diagnosis and corresponding hint fully captured the logical errors in the submission, and so would potentially provide useful guidance to the student. Note that we say "potentially" since the labeling was done by people other than the owners of the submissions. *Possibly Correct* meant that the diagnosis and hint only partially captured the errors in the submission. *Incorrect* meant that the errors were misdiagnosed and so the hint was misleading. Each submission was inspected and evaluated by at least two authors. The results were analyzed by a fourth author to resolve conflicts.

Table 3 presents the results. Examination of *Incorrect* cases for both assignments revealed errors that would be difficult to detect with black box testing. Thus, improving accuracy would likely require the use of additional complementary or more powerful methods for assessment (for example, compiler analysis techniques).

We conclude that the automatic error categorization and corresponding hints are appropriate for the vast majority of the erroneous submissions, and so we have deployed them in the Fall 2017 semester.

5.2 Examples of Common Errors and Hints

Example 1. For *PayFriend*, common errors include incorrect conditional expressions leading to incorrect answers for boundary values,

Table 3: Accuracy of error categorization and hints for Spring 2017 submissions. Each row presents statistics for a bucket of submissions with the same errors.

Error Codes	Automatic Categorization	*Correct*		*Partially Correct*	
PayFriend					
COMP	34	34	100%	0	0%
INS	111	111	100%	0	0%
IO	119	114	95.8%	3	2.5%
INF	7	4	57.1%	3	42.9%
CF	38	26	68.4%	4	10.5%
COND	44	39	88.6%	4	9.1%
COND, FORM	91	82	90.1%	9	9.9%
FORM	83	72	86.7%	4	4.8%
Total	**527**	**482**	**91.5%**	**27**	**5.1%**
TwoSmallest					
COMP	39	39	100%	0	0%
INS	105	104	99%	1	1%
SEQ	51	47	92.2%	4	7.8%
INIT	67	56	91.8%	5	8.2%
UPDT	158	129	87.2%	19	12.8%
SEQ, INIT	157	137	91.9%	12	8.1%
SEQ, UPDT	176	145	85.8%	24	14.2%
Total	**767**	**671**	**87.5%**	**65**	**8.5%**

and incorrect formulas for one or more fee tiers leading to incorrect answers for entire tiers. It is useful to differentiate between the two errors when giving students hints. We were able to make this distinction using the combined outputs of multiple tests.

More specifically, if a submission fails test cases with input values inside one tier, *I*, but passes test cases with input values in other tiers, it is likely that the code does not correctly calculate the fee for tier *I*. This signature leads to the following hint for tier ($100 to $1000): *"It seems that you are not correctly calculating the fee for payments in the range (100, 1000). Review the assignment instructions, check that your formula for computing the fee is correct, then follow the steps used in the calculation of the fee in your code and make sure that they implement the correct formula."*

On the other hand, if the submission passes test cases with input values inside *I*, but fails test cases with inputs near the upper or lower boundaries of *I*, it is likely that the code uses incorrect conditional expressions. This may arise from misunderstandings of conditional statements and expressions, and/or misunderstandings of the assignment write-up, hence the mapping to the skill *translating word problems into conditional statements*. For example, discriminating between $\geq$ and $>$ in a conditional expression requires understanding of boundary values and how they differ among data types. For integers, $x < 100$ is equivalent to $x \leq 99$, but this is not true for real values. We thus wrote the following hint: *"It seems that you did not split the input intervals correctly, where some values at the boundary between intervals may have been included under the wrong formula/rule; that is, your conditional expressions may be incorrect, for example you may have $\geq$ 101 instead of > 100 which are not equivalent expressions for double values."*

Example 2. For *TwoSmallest*, given the material that has been taught in class, most students develop algorithms that have two major steps: (1) initialize two variables used to hold the two smallest values, and (2) read the input sequence and update the variables. Many students make the algorithmic error of not considering what the initial values of the variables should be, and often initialize them to inappropriate values such as 0. Comparing the results of several test cases with input values that are positive, negative, and mixed can tell us whether or not a submission has this error. Also, the mapping of this error to *algorithmic thinking* in the knowledge map reminds us to view the error in light of the student's algorithmic design effort. This leads to the hint: *"It seems that you did not initialize the variables used to hold the minimum and secondMinimum to reasonable values. Think about how the starting values would affect your algorithm for finding the two smallest values. In particular, what would happen if the input values in the sequence were greater, equal or less than the starting values for your minimum and secondMinimum."*

Updating the two variables in *TwoSmallest* requires algorithmic thinking, and can be a challenge for students new to programming. Many students tend to think about the update process in fragmented, poorly coordinated pieces. To assess if the update of the variables is done correctly, we test input sequences that are permutations of two and three numbers. If the submitted code passes all the test cases with a valid input of size two but fails the test cases where the third value is less than the minimum value, then it is highly likely that the student is not updating the minimum value correctly. The mapping of the error to "algorithmic thinking" again leads us to write a hint designed to steer students toward developing this skill: *"It seems that you did not update the variables holding the minimum and/or secondMinimum values correctly. Think carefully about the algorithm that you are developing to update your variables. It may help to think about what would happen if the sequence had the same number appearing multiple times; for example, all possible permutations of 3 numbers with repetition."*

6 DISCUSSION

CSF^2 is designed to be flexible enough to serve the needs of many programming courses that use autograding. It can assist instructors in various tasks from creating assignments, to reviewing course material, to encouraging interactions between students through discussion of the hints. In this section, we discuss possible modifications to CSF^2 and the use of the knowledge map.

6.1 Modifications to the Framework

The process described in Section 3 is meant to be a guideline for best practices. As written, it describes a top-down approach to the design of assignments using knowledge maps. The advantage of this approach is that assignments are carefully and systematically written to target specific course material. However, in practice and as we see in our case studies, instructors have their assignments already written and have already used them over the course of a few semesters. In these situations, a "bottom-up" approach can be used with a few advantages: previously collected submissions can be used to guide the extraction of the knowledge map, design or improve the test suite, and generate the error classifiers and hints.

Much of the manual work done for this research is labor intensive. While we have manually reviewed all 527 and 767 erroneous submissions for *PayFriend* and *TwoSmallest* in this work, it may be that reviewing only a sample of submissions would be sufficient. We are currently exploring the use of random subsets of various sizes to gauge the sensitivity of the error categorization and hint generation on sample size.

6.2 Using the Knowledge Map

One advantage of using a knowledge map and a classifier is the ability to query which concepts and skills students are struggling with, similar to the results shown in Tables 2 and 3. With this knowledge, instructors can devise specific interventions targeting knowledge deficiencies that can be deployed prior to the administration of the assignment. Some of these proposed interventions may include additional exercises, textbook references, and/or video lessons.

Instructors can also take advantage of the knowledge map when assigning partial credit to autograded assignments. Traditionally, grading rules have been very rigid, following the boundaries of unit testing and leading to a linearly scaled grade based on the percentage of passed test cases. Relying on test cases alone to grade submissions resulted in some unexpected behaviors. Instructors at our university reported that students at either end of the scoring spectrum (that is, those who received no credit and those who earned nearly full credit despite still having important misconceptions and mistakes), gave up working on and completing their programming assignments. With our proposed knowledge map, instructors can assign scores based on the importance they allot to each concept or skill. For example, for *PayFriend* we weighted all the tested concepts starting with compilation and ending with conditional statements. Students who understood the assignment but previously would have received low grades due to failed test cases, were more fairly assigned scores that reflected their understanding of the problem and its solution using the knowledge map. Conversely, we found submissions that previously would have received near perfect scores since they only failed a few tests. The weighted knowledge map scoring lowered their scores because these test failures showed misunderstandings of core concepts, serving as a motivation for the students to improve their solutions.

Finally, the knowledge map and buckets of common errors can aid in the generation of isomorphic assignments while reusing the error classifier and hints. For example, in Fall 2017, we modified *TwoSmallest* into *TwoLargest*, where the students were asked to output the two largest values in a sequence. In this instance, we were able to reuse the test suite, classifier, and hints with only small changes. This can be a starting point for extending CSF^2 to support the evolution of assignments over time (for example, to circumvent cheating) while reusing classifiers and hints.

7 CONCLUSIONS

In this paper, we presented a methodology which bridges the gap between autograding and the knowledge assessment of programming assignments to provide meaningful feedback to students. Our methodology asks the instructor to systematically analyze programming assignments with respect to knowledge maps to ensure course cohesion between the specific challenges posed to students by the programming assignments and the material taught in class. The methodology also outlines an approach for finding common errors in a set of submissions for an assignment, and generating an error classifier and hints that can be used by an autograder to give feedback for future submissions. This process can also give instructors insight on how to adjust class material to address observed knowledge deficiencies. We have applied our methodology to two assignments in our introductory course, and found that the hints should provide useful feedback for the vast majority of incorrect submissions.

Acknowledgement. This work was partially supported by a Google Computer Science Capacity Award.

REFERENCES

[1] S. A. Ambrose, M. W. Bridges, M. DiPietro, M. C. Lovett, and M. K. Norman. *How Learning Works: Seven Research-based Principles for Smart Teaching*. Jossey-Bass, 2010.
[2] C. Benac Earle, L. A. Fredlund, and J. Hughes. Automatic Grading of Programming Exercises Using Property-Based Testing. In *Proceedings of the 2016 ITiCSE Conference*, 2016.
[3] S. Bhatia and R. Singh. Automated Correction for Syntax Errors in Programming Assignments using Recurrent Neural Networks. *CoRR*, 2016.
[4] Computing Research Association. Generation CS: Computer Science Undergraduate Enrollments Surge Since 2006. https://cra.org/data/Generation-CS/.
[5] S. H. Edwards and M. A. Perez-Quinones. Web-CAT: Automatically Grading Programming Assignments. In *ACM SIGCSE Bulletin*, volume 40, 2008.
[6] E. L. Glassman, A. Lin, C. J. Cai, and R. C. Miller. Learnersourcing Personalized Hints. In *Proceedings of the 2016 CSCW Conference*, 2016.
[7] A. Head, E. Glassman, G. Soares, R. Suzuki, L. Figueredo, L. D'Antoni, and B. Hartmann. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the 2017 Conference on Learning @ Scale*. ACM, 2017.
[8] J. Holland, A. Mitrovic, and B. Martin. J-LATTE: a Constraint-based Tutor for Java. In *The 2009 International Conference on Computers in Education*, 2009.
[9] J. Huang, C. Piech, A. Nguyen, and L. Guibas. Syntactic and Functional Variability of a Million Code Submissions in a Machine Learning MOOC. In *Proceedings of the 2013 Workshop on Massive Open Online Courses at the 16th Annual Conference on Artificial Intelligence in Education*, 2013.
[10] H. Keuning, J. Jeuring, and B. Heeren. Towards a Systematic Review of Automated Feedback Generation for Programming Exercises. In *Proceedings of the 2016 Conference on Innovation and Technology in Computer Science Education*, 2016.
[11] N.-T. Le and N. Pinkwart. Towards a Classification for Programming Exercises. In *Proceedings of the 2nd Workshop on AI-supported Education for Computer Science*, 2014.
[12] D. Milojicic. Autograding in the Cloud: Interview with David O'Hallaron. *IEEE Internet Computing*, 2011.
[13] National Center for Women & Information Technology. Projected Computing Jobs and CIS Degrees Earned. http://www.ncwit.org/sites/default/files/file_type/usnatgraphic2022projections_10132014.pdf.
[14] B. Paaßen, J. Jensen, and B. Hammer. Execution Traces as a Powerful Data Representation for Intelligent Tutoring Systems for Programming. In *Proceedings of the 2016 International Conference on Educational Data Mining*. International Educational Datamining Society, 2016.
[15] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. J. Guibas. Learning Program Embeddings to Propagate Feedback on Student Code. *CoRR*, 2015.
[16] K. Rivers and K. R. Koedinger. Data-Driven Hint Generation in Vast Solution Spaces: A Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education*, 27, 2017.
[17] S. Rogers, D. Garcia, J. F. Canny, S. Tang, and D. Kang. ACES: Automatic Evaluation of Coding Style. Master's thesis, EECS Department, University of California, Berkeley, May 2014.
[18] G. Singh, S. Srikant, and V. Aggarwal. Question Independent Grading Using Machine Learning: The Case of Computer Program Grading. In *Proceedings of the 2016 SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016.
[19] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. *SIGPLAN Not.*, 48(6), 2013.