CSF^2: Formative Feedback in Autograding*

GEORGIANA HALDEMAN, Rutgers University, Computer Science Department, USA
MONICA BABEŞ-VROMAN, Rutgers University, Computer Science Department, USA
ANDREW TJANG, Rutgers University, Computer Science Department, USA
THU D. NGUYEN, Rutgers University, Computer Science Department, USA

Autograding systems are being increasingly deployed to meet the challenges of teaching programming at scale. Studies show that formative feedback can greatly help novices learn programming. This work extends an autograder, enabling it to provide formative feedback on programming assignment submissions. Our methodology starts with the design of a knowledge map, which is the set of concepts and skills that are necessary to complete an assignment, followed by the design of the assignment and that of a comprehensive test suite for identifying logical errors in the submitted code. Test cases are used to test the student submissions and learn classes of common errors. For each assignment, we train a classifier that automatically categorizes errors in a submission based on the outcome of the test suite. The instructor maps the errors to corresponding concepts and skills and writes hints to help students find their misconceptions and mistakes. We apply this methodology to two assignments in our Introduction to Computer Science course and find that the automatic error categorization has a 90% average accuracy. We report and compare data from two semesters, one semester when hints are given for the two assignments and one when hints are not given. Results show that the percentage of students who successfully complete the assignments after an initial erroneous submission is three times greater when hints are given compared to when hints are not given. However, on average, even when hints are provided, almost half of the students fail to correct their code so that it passes all the test cases. The initial implementation of the framework focuses on the functional correctness of the programs as reflected by the outcome of the test cases. In our future work, we will explore other kinds of feedback and approaches to automatically generate feedback to better serve the educational needs of the students.

CCS Concepts: • **Social and professional topics** → **Computing education**; **CS1**; *Student assessment*; • **Applied computing** → **Computer-assisted instruction**; • **Human-centered computing** → *Interactive systems and tools*.

Additional Key Words and Phrases: Autograding, Computer Science Concepts and Skills Map, Error Extraction and Categorization, Code Clustering, Formative and Corrective Feedback

ACM Reference Format:
Georgiana Haldeman, Monica Babeş-Vroman, Andrew Tjang, and Thu D. Nguyen. 2021. CSF^2: Formative Feedback in Autograding. *ACM Trans. Comput. Educ.* 21, 3, Article 21 (May 2021), 32 pages. https://doi.org/10.1145/3445983

21

*This work is an extension of the *Providing Meaningful Feedback for Autograding of Programming Assignments* paper published in *SIGCSE* 2018. This work was partially supported by a Google Computer Science Capacity Award.

Authors' addresses: Georgiana Haldeman, Rutgers University, Computer Science Department, 110 Frelinghuysen Rd, Piscataway, NJ, USA, mgh80@cs.rutgers.edu; Monica Babeş-Vroman, Rutgers University, Computer Science Department, 110 Frelinghuysen Rd, Piscataway, NJ, USA, babes@cs.rutgers.com; Andrew Tjang, Rutgers University, Computer Science Department, 110 Frelinghuysen Rd, Piscataway, NJ, USA, atjang@scarletmail.rutgers.edu; Thu D. Nguyen, Rutgers University, Computer Science Department, 110 Frelinghuysen Rd, Piscataway, NJ, USA, tdnguyen@cs.rutgers.edu.

1946-6226/2021/May-ART21 $15.00
https://doi.org/10.1145/3445983

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 [1]. Correspondingly, universities are seeing tremendous enrollment growth in computing classes and are faced with the challenge of teaching programming at scale [2].

Programming assignments are tools of choice in many computing classes, giving students hands-on coding practice. 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 autograding systems, such as Web-CAT [3] and Autolab [4] do not provide feedback by default. Even when feedback is available, it often does not assist students in correcting errors and it does not address underlying misconceptions [5]. In this paper, we aim to answer these specific research questions:

(1) Can we identify patterns of passed and failed test cases which point to logical errors in the students' code? Furthermore, can we partition the students' code based on logical errors using these patterns of passed and failed test cases?
(2) How accurate are these patterns of passed and failed test cases for partitioning the students' code?
(3) If it is possible to partition the students' code, can we write hints for an autograding system to give meaningful feedback that actually helps students identify their errors and make progress?
(4) How are the hints associated with each partition perceived by students and instructors?

Although these research questions have been explored in other works [6–8], in this paper we seek to develop a systematic way for instructors to follow when designing assignments. We propose a methodology that is similar in spirit to a previous approach [9], but, instead, focuses on linking errors to the concepts and skills required for solving a programming assignment. Our work leverages code testing, which is the most common method for grading and assessment in programming. Since concepts and skills are shared among assignments, our approach offers the potential for reusing some of the work done in previous assignments.[1] Our approach relies on collecting and analyzing assignment submissions to generate hints that can be used during future semesters. More specifically, when designing an assignment using our approach, the instructor explicitly defines the set of concepts and skills that students need to master to complete the assignment. After the assignment has been written, a comprehensive test suite is developed to tease out programming misconceptions and to test the correctness of the code 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 pattern (which we call the *signature*) to identify errors. In this process, the test suite can also be refined as needed or desired. Finally, the instructor maps the errors to specific concepts and skills and writes hints that are designed to guide the students toward correcting their code and improving their understanding of the corresponding concepts and skills. The obtained signatures can be used to produce a classifier that automatically categorizes erroneous submissions, enabling the autograding system to provide hints as feedback to the student code submissions.

The above methodology encapsulates two key ideas: (1) the instructor can identify high level logical errors by inspecting sets of submissions with the same error signature if the error signatures are generated using a comprehensive and well designed test suite, and (2) the mapping of errors to concepts and skills can produce hints that encourage students to think about their code conceptually,

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

as opposed to suggesting local, code-specific changes that can lead to highly convoluted solutions, such as solutions containing unnecessary nested conditional statements. Understanding common errors in light of the concepts and skills they map to can also help the instructor adjust her classroom teaching.

We applied our proposed methodology to two assignments in our Introduction to Computer Science course and collected a large number of submissions for each assignment during three semesters, Spring 2016, Spring 2017, and Spring 2018. We designed a test suite for each assignment and used submissions from Spring 2016 to learn classes of common errors, produce classifiers for the automatic error categorization of future submissions, and write hints. Then, we used the classifiers to attach hints to the erroneous submissions from Spring 2017. Four researchers manually reviewed the results of the classifiers and found that over 91% of the hints for the first assignment and over 87% of the hints for the second assignment fully captured the errors in the corresponding submissions. These percentages rose to over 96% for both assignments when we counted hints that partially captured the errors. Based on these promising results, starting with Fall 2017, we deployed the error categorization and corresponding hints for the two assignments. We compared submissions from two semesters, one when hints were not provided (Spring 2017) and one when hints were provided (Spring 2018), and found that when hints were provided, students submitted more often, more students made progress, and the overall progress toward completing the assignment was faster. For example, the percentage of students who successfully completed the assignments after an initial erroneous submission was three times greater when hints were given compared to when hints were not given.

Moreover, during Spring 2018, we asked students to complete a survey regarding the usefulness of the hints. We report the students' responses and their comments about the hints in Section 5.4.2. Lastly, we asked the course instructors for feedback about the hints. We found that students and instructors thought that the hints were helpful, but also that they could be improved, in particular the information provided in the hints as well as their wording. These are highly debatable subjects that go beyond the scope of our current research.

2 RELATED WORK

Formative Feedback

Shute [10] reviews feedback given to learners with a focus on formative feedback and concludes that *"feedback used in educational contexts is generally regarded as crucial to improving knowledge and skill acquisition."* Shute further defines formative feedback as *"information communicated to the learner that is intended to modify his or her thinking or behavior to improve learning."* According to Ambrose et al. [11], *"goal-directed practice coupled with targeted feedback are critical to learning."* Hints provided by autograding systems can be an important source of formative feedback, but their impact is not well understood. Marwan et al. [12] explore whether automatically generated hints can lead to better outcomes for students and whether other forms of feedback such as textual explanations and self-explanation prompts can improve hints. Our work is complementary to theirs and their findings can be used to further improve the hints given to students.

Domain Model for Computer Programming

The ability to reason about a specific domain is at the heart of teaching and learning. The process of autograding is based on using a submitted assignment to build a model of the students' reasoning about a specific problem. In CS, an autograder is a system that uses such a model to give students an accurate grade and, potentially, feedback. Woolf [13] calls these models *domain models* and defines them as qualitative representations of expert knowledge in a specific domain. Woolf further

explains that compiling a domain model for computer programming is very challenging because programming is a complex, ill-structured design domain. In the context of our class, what helps alleviate this issue is that we study the assignments of students across semesters and in larger groups. These larger groups of student share common errors and misconceptions as it has been observed in prior research [6–8, 14].

Mayer et al. [15] explore the relationship between thinking skills and programming skills. Brennan and Resnick [16] propose three dimensions of computational thinking: computational concepts, computational practices, and computational perspectives. In this paper, we use a combination of concepts and skills necessary to solve two programming assignments similar to [15, 16]. Programming assignment problems vary in their difficulty, for example, in the number of solution strategies they accept [17, 18]. The assignment problems we studied accept multiple solution implementations and multiple solution strategies which correspond to classes 2 and 3 according to [17, 18].

Tools that Support Feedback at Scale

Several existing tools are designed to help instructors manage the large number of students enrolled in introductory programming courses. Autograders, for example, are designed to automatically grade the students' programming assignments. In our Introduction to Computer Science course we have adopted Web-CAT [3] and Autolab [4] . To employ these tools, the instructor provides a grading scheme and an executable file which the systems use to grade the students' code. Singh et al. [19] offer an assignment-independent solution to grading by introducing a problem-independent grammar of features. They use supervised learning to train a classifier on teacher-graded examples. The classifier maps new student code submissions to grades. Similarly, our work explores the extension of an autograding system to provide formative feedback by using the results of test cases as features and a discrete hybrid (unsupervised and supervised) learning algorithm. There are three categories of systems that provide feedback at scale: tools that support program repair at scale, tools that generate next-step hints at scale, and tools that support instructor feedback at scale. Next, we summarize related work in each of these categories. For an in-depth and comprehensive comparison of various types of tools for automated programming hint generation, we recommend a recent survey on the subject [20].

Tools that Support Program Repair Feedback at Scale. Providing automatically generated feedback is a very attractive idea but also a very challenging one. The majority of existing tools can only give program repair hints to students. Autograder [14] requires a reference solution and an error model consisting of possible corrections to errors that students might make. Then, it tries to find the smallest possible number of corrections to the students' code using a constraint solving program synthesis approach. Qlose [21] is similar to Autograder, except it does not require an error model. Instead, it tries to find the best correction by minimizing both the syntactic and semantic distances to the reference solution. Sarfgen [22] is a data driven program repair framework that takes advantage of a large number of available student submissions and tries to find minimal fixes by aligning incorrect programs with similar programs that are correct. MistakeBrowser [9] tries out various previously seen transformations of the student's code until the program is correct. One extension to MistakeBrowser generates hints based on the synthesized correct program [23]. Although such systems have shown great potential in generating the program repair closest to what a programmer would suggest, this kind of feedback does not cause students to think abstractly about their solutions.

Tools that Generate Next-Step Hints at Scale. The main difference between the tools that generate next-step hints and the tools that support program repair is in the granularity of the attempted fix. While the tools for program repair attempt to fix the issues in the student's program all at once,

the tools that generate next-step hints use a step-by-step approach to fixing errors. Hint Factory [24] uses peer data and a Markov Decision Process to produce next-step hints for the students. A follow-up approach, the Continuous Hint Factory [25] uses the weighted sum of peer edits to select the best next step. Zhi et al. [26] use features based on the code structure to extract worked example pairs which are, then, offered as feedback to the students. Gerdes et al. [27] define a strategy language that specifies how parts of a solution may be built up from others and uses it to generate next steps rather than using existing steps from peer data. While the step-by-step approach may be beneficial due to the breaking down of the complexity of the entire fix, the feedback produced is very similar to the one generated using program repair.

Tools that Support Instructor Feedback at Scale. CodeOpticon [28] enables instructors to monitor multiple students simultaneously as they code and give them assistance online. OverCode [29] and Foobaz [30] cluster student code and report common and uncommon student choices on syntax and style. AutoStyle [31] clusters correct submissions using the ABC software metric and propagates hints to each cluster focused on writing simpler code. MistakeBrowser [9] gives teachers a high-level overview of the students' misconceptions and the bugs in their code. FixPropagator [9] learns transformations from incorrect to correct code from teachers fixing bugs in incorrect student submissions. Teachers can, then, write feedback to a single submission or to a cluster of submissions and the feedback gets propagated to all submissions that can be fixed by the same transformation. Our system uses a similar process to propagate the instructor's feedback on one submission to all the submissions with similar error signatures, but we use different features to cluster submissions.

Lastly, a survey on the automated assessment of programming assignments found that many of the existing tools are either not open to the public or too cumbersome to be adopted by instructors [32]. In our approach, the instructor is the one designing the feedback by using testcases to assess the students' code, a flexible approach that should be easy to adopt.

Clustering Student Submissions and Bugs

Using Test Cases to Cluster Submissions. One approach [8] clusters bug fixes by failed test cases. Web-CAT [3] includes a library that attaches a hint to each test case. When a submission fails a test case, the associated hint is returned as feedback. In our experience, it is difficult to write hints that are not overly specific or overly vague based on the result of one test case. Thus, our approach uses results from an entire test suite to identify common errors and provide hints.

Using Other Methods to Cluster Submissions. Clustering similar student code submissions in a robust, general way is challenging. Huang et al. [33] cluster student submissions using the abstract syntax tree (AST) edit distance. Nguyen et al. [6] cluster functionally equivalent but syntactically distinct code phrases using probabilistic semantic equivalence. Piech et al. [7] cluster submissions using neural networks to learn program embeddings. Kaleeswaran et al. [34] cluster dynamic programming (DP) submissions using static analysis to detect how students manipulate arrays. Rather than clustering based on behavioral or syntactic similarity, MistakeBrowser and FixPropagator [9] cluster incorrect submissions based on the transformation that corrects them. Finally, instead of clustering code, HelpMeOut [35] clusters bug fixes by compiler error or runtime exceptions. Our system clusters submissions based on the outcome of a test suite designed to detect common misconceptions of the course material. Each cluster corresponds to a different set of logical errors and gets assigned a hint which is returned to the student to assist in correcting her code and misconceptions.

Algorithmically Generating Feedback

Providing personalized feedback in Intelligent Tutoring Systems (ITS) is done via constraint-based methods [36]. However, this approach requires teaching expertise and, often, assignment specific implementations, which does not scale with the number of assignments. Data-driven procedures improve existing methods. Newer approaches, ITAP [37] and Codewebs [6] leverage the statistical properties of a large number of student submissions by extracting patterns that can be used for refining the clustering and providing improved feedback. These works are complementary to ours in that they provide automated techniques for providing feedback that could potentially reduce the human effort required by our framework.

Our research is similar in spirit to other research efforts which support instructor feedback at scale, in particular to FixPropagator [9]. One important benefit of our approach is that it explicitly tries to bridge the teaching of abstract concepts with hands-on programming practice. This is accomplished by linking student programming errors to the concepts and skills taught in the class. Furthermore, we provide a systematic way to tease out both concepts and skills and the student errors. This information can be used to further explore the relationship between teaching programming concepts and student coding errors. For example, the instructors can easily tailor their teaching of concepts and skills based on the students' performance on the programming assignments.

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 having taken the class. These concepts and skills are used to generate hints for assisting students in correcting errors. The approach is described as a sequence of steps, but the ordering can be modified as discussed in Section 4.3. The proposed steps are as follows.

(1) Carefully list the set of concepts and skills that students need to master.
(2) Write the assignment to evaluate these concepts and skills.
(3) Design a test suite to assess student submissions. A full path coverage of a reference solution is typically a good start (but will need expansion). To test a submission, each test case should output a code representing the outcome of the test.
(4) Release the assignment and collect student submissions.
(5) Automatically run the test suite against the collected submissions and group submissions into "buckets" based on their outcome signatures, where a signature is the concatenation of the codes output by all the test cases in the test suite. Each signature may indicate one or more logical errors. Our hypothesis is that submissions with the same signature are likely to have similar logical errors.
(6) Manually inspect each bucket of submissions to see whether subsets of submissions have different logical errors. If this is the case, then add test cases to the test suite to separate these subsets into different buckets. The list of concepts and skills may also need to be refined (for example, to include a concept that has to be mastered for the completion of the assignment but was accidentally omitted in 1).
(7) Repeat 5 and 6 until submissions in each bucket have the same logical errors.
(8) Map the errors identified for each bucket to concepts and skills. Then, manually inspect and combine buckets of submissions with the same errors or knowledge deficiencies. Our hypothesis is that mapping errors to concepts and skills will help instructors reduce the number of hints that will need to be written, as well as write hints that provide conceptual guidance rather than very specific code changes.

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

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 the 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 more challenging concepts and skills.

As described above, CSF^2 is most useful when an assignment is given during multiple semesters, with previous submissions used to generate and improve hints for subsequent semesters. Currently, CSF^2 does not directly support the evolution of an assignment over time (for example, changes to the assignment to discourage cheating or to improve the assignment), although we have successfully experimented with changing one of the assignments studied in this paper to an isomorphic assignment while still making use of analysis results from previous submissions (Section 4.3). 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. As part of our research, we had three undergraduate students who helped with our case studies.

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 a large public research university. Even though CSF^2 proposes that an instructor start with a list of concepts and skills when designing an assignment, our case studies start with existing assignments because all assignments are built on an implicit list of concept and skills and we had already collected a large number of student submissions across several semesters before the start of this research. In essence, we reversed the order of Steps 1 and 2 in CSF^2, and extracted the concepts and skills from existing assignments.

4.1 Application of CSF^2 to Two Programming Assignment

Steps 1 and 2: Extracting the Concepts and Skills

We studied two assignments, *PayFriend*, a class 2 assignment [18], 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, as well as multiple possible implementations for each strategy. *PayFriend* asks students to compute the fee associated with making an e-payment when given a tiered fee structure, with different fees for four payment ranges.*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.

Each assignment requires that the solution be implemented as a method with a prescribed 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 any of these instructions results in compilation errors and a score of zero.

Table 1. The mapping of common errors to concepts and skills for two programming assignments, *PayFriend* and *TwoSmallest*. The base score is used for calculating grades as discussed in Section 5.1.

Code	Error	Base Score	Concept or Skill
Both assignments			
COMP	has compilation errors	0	writing code that compiles
INS	has errors regarding the required formatting, for example, incorrect file name	5	following instructions
IO	has IO errors, for example, wrong types of inputs or outputs	10-30	data representation, following instructions
INF	uses infinite loops	5-40	control flow
PayFriend			
CF	outputs only in some branches	50	control flow
COND	uses incorrect conditional statements	50	translating word problems into conditional statements
FORM	uses an incorrect calculation inside an interval	50	translating word problems into formulas
TwoSmallest			
SEQ	reads and processes incorrectly a sequence of values	40	data representation, following instructions
INIT	initializes min values incorrectly	40	algorithmic thinking
UPDT	updates min values incorrectly	40	algorithmic thinking

When we started this research, *PayFriend* and *TwoSmallest* had already been assigned during several semesters. We worked with the lead instructor to determine the concepts and skills corresponding to these assignments. Some of these concepts and skills are shown in the right column of Table 1.

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 corresponding reference solution. Next, we considered more challenging inputs, especially for novice programmers. For example, it is well known that many programming bugs involve an incorrect handling of boundary values. 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. The process of designing a test suite was iterative (as discussed in Steps 5–7 of CSF^2). After all the refinements were made, the test suite for *PayFriend* contained 20 test cases and the one for *TwoSmallest* contained 30. When these assignments were used during previous semesters, *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 [3] and during the Spring 2017 and 2018 semesters using Autolab [4]. Table 2 shows information about our data sets. Each submission included anonymized student information, a time stamp, and code. In this study, we looked at the submitted code—all submissions were anonymized by removing all information, including comments, other than the actual code—and used the anonymized student information to link submissions from each unique student in a

Table 2. Summary of the data sets used in our study. During each semester, students were allowed to submit each assignment without penalty up to five times during Spring 2016 and up to three times during Spring 2017 and 2018. The penalty for each subsequent submission was 5 points.

	PayFriend			*TwoSmallest*		
	Spring 2016	**Spring 2017**	**Spring 2018**	**Spring 2016**	**Spring 2017**	**Spring 2018**
Autograder used	Web-CAT	Autolab		Web-CAT	Autolab	
Number of submissions	1152	719	936	1339	870	1071
Number of students who submitted at least once	511	432	487	488	423	476
Average number of submissions per student	2.3	1.7	1.9	2.7	2.1	2.3
% of students who submitted at least twice	62.0%	36.4%	53.0%	72.3%	54.1%	60.1%

semester (students can submit each assignment multiple times). The submissions from Spring 2016 were used for Steps 5–9 of CSF^2 to build an error classifier and generate a set of hints for each assignment (as detailed below). The submissions from Spring 2017 and Spring 2018 were used to evaluate the accuracy of the classifiers and the effectiveness of the hints as described in Section 5.

Steps 5–7: Refining Tests and Partitioning Submissions

We used the test suite developed in Step 3 to test and generate the outcome signature for every code submission. Then, submissions with the same signature were grouped in the same bucket. Each signature could indicate one or more logical errors. 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 any given bucket, some of the submissions had one error while others had a different error, we added test cases to separate submissions with different errors into different buckets.

We manually inspected the students' code in each bucket to determine the main reason why the code failed one or more test cases. 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 resulted in 109 non-empty buckets for *PayFriend* (using 20 test cases) and 137 non-empty buckets for *TwoSmallest* (using 30 test cases).

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 1. As already mentioned, we found that many of the buckets were different manifestations of similar knowledge deficiencies. We merged buckets accordingly, leading to 8 "super-buckets" for *PayFriend* and 7 for *TwoSmallest* (Table 3). Clustering student submissions for assignments in classes 2 and 3 [18] is a particularly challenging task because their solution space can be large. Since unit testing mostly focuses on the functionality of the code rather than its style, we were able to cluster stylistically different student submissions in the same bucket.

Table 3. Final buckets (classes) of errors for *PayFriend* and *TwoSmallest*. The table shows statistics for errors found in Spring 2017 submissions together with the accuracy of the error classification and hints. The latter is discussed in Section 5.3.

Error Codes	Automatic Classification	Correct		Partially Correct	
PayFriend					
	Total Count	**Count**	**% of Total**	**Count**	**% of Total**
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					
	Total Count	**Count**	**% of Total**	**Count**	**% of Total**
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%**

We used the clustering of the student submissions from Spring 2016 and the signatures for each bucket to develop classifiers for every assignment. Then, we ran the classifiers on the student submissions from Spring 2017 and manually evaluated their accuracy as described in Section 5.3. Finally, we wrote a hint for each bucket. This hint was given to students after every submission, based on the result of the assignment's classifier. The next section provides examples of common errors and the corresponding hints given to students during the Spring 2018 semester.

4.2 Examples of Common Errors and Hints

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

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, $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 a misunderstanding of conditional statements and expressions, or a misunderstanding of the assignment instructions, or both, hence the mapping to the skill *translating word problems into conditional statements*. For example, discriminating between $\geq$ and $>$ in a conditional expression requires understanding 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. Thus, we wrote the following hint for this class of errors: *"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 for storing the two smallest values, and (2) read the input sequence and update the variables correspondingly. Many students do not consider what values they should use to initialize the variables and end up using improper initial values such as 0. By using the results of several test cases with input values that are positive, negative, and mixed, we can tell whether or not a submission has this mistake. We map this error to *algorithmic thinking*, which 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 given 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" leads us, again, to 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."*

4.3 Discussion

CSF^2 was developed in the context of an introductory course, but we believe that its design is sufficiently flexible to be extended and adapted to serve the needs of other programming courses that use autograding. It can assist instructors in various tasks from creating assignments to reviewing course material and encouraging informal interactions between students through discussion of the hints. In this section, we discuss possible modifications to CSF^2 and to the use of the set of concepts and skills for each of the assignments.

4.3.1 Modifications to the Framework. Top-down and bottom-up are two well known strategies of information processing and knowledge ordering, and our framework can be modified to employ

either of them. The top-down approach starts with the whole problem and decomposes it into individual steps. The bottom-up approach pieces together individual parts into bigger parts. 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 specific sets of concepts and skills. The advantage of this approach is that assignments are carefully and systematically written to target specific course material. However, in practice, and as we saw in our case studies, instructors may already have the assignments written and used. In these situations, a bottom-up approach can be applied. This approach has a few advantages: previously collected submissions can be used to guide the derivation of the set of concepts and skills that map to the specific assignment, it provides a way to design or improve the test suite, and it gives instructors a way to generate the error classifiers and hints.

Much of the manual work done for this research was labor intensive because we manually reviewed all the erroneous submissions for *PayFriend* and *TwoSmallest*. As an alternative to reviewing all the submissions, it may be sufficient to use random subsets of varying sizes to determine how the error categorization and hint generation change with sample size. We will consider this analysis in our future work.

4.3.2 Using the Set of Concepts and Skills which map to Specific Assignments. One advantage of using a specific set of concept and skills which map to an assignment and an error classifier is the ability to query which concepts and skills students are struggling with, similar to the results shown in Table 3. With this information, instructors can design interventions targeting specific knowledge deficiencies. These interventions can be used prior to the administration of the assignment. Some of these proposed interventions may include additional exercises, textbook references, or video lessons.

Instructors can also take advantage of the set of concepts and skills when assigning partial credit to autograded assignments. Traditionally, grading rules for autograders 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 has resulted in some unexpected behaviors. For example, 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 approach, instructors can assign scores based on the importance they allot to each concept or skill. For example, for *PayFriend* we weighed all the tested concepts. Students who understood the assignment but would have, previously, received low grades due to failed test cases were assigned scores more fairly and the scores better reflected their understanding of the problem and its solution. Conversely, we found submissions that failed a few test cases covering core concepts and would have previously received a nearly perfect score. The weighted scoring scheme lowered the scores of these submissions because the test case failures showed important misunderstandings and served as a motivation for students to improve their solutions.

Finally, the set of concepts and skills and buckets of common errors can aid in the generation of isomorphic assignments while reusing the error classifier and hints. For example, during the Fall 2017 semester, we modified *TwoSmallest* to *TwoLargest*, an assignment that asked students 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 the classifiers and hints.

5 FINDINGS FROM CASE STUDIES

In this section, we present findings from our case studies. We first assess the accuracy of the automatic error classification and whether the hints captured the errors in the code submissions collected during Spring 2017. Recall that the classifiers were developed using the submissions collected during Spring 2016 as described in the last section. Then, we used code submissions from Spring 2017 and Spring 2018 together with results from a survey conducted in 2018 to assess the usefulness of the hints.

5.1 Datasets, Feedback, and Grading

As mentioned before, Table 2 summarizes the data sets used in our evaluation, with submissions for Spring 2016 collected using Web-CAT and submissions for Spring 2017 and Spring 2018 collected using Autolab. Web-CAT was configured to test the students' code using 10 test cases. Students were allowed to submit each assignment multiple times - five times without penalty, then with a 5 point penalty for each subsequent submission. For each submission, Web-CAT provided the following feedback: a score, whether the submission passed each test t in the test suite, and, in case of failure, a hint associated with the specific test case.[2] The instructor wrote the test cases and the hints at the same time. Scores were calculated as a weighted sum of the passed test cases.

Instructors made a few observations during Spring 2016 which led to changes during Spring 2017 and Spring 2018:

(1) when given feedback on which test cases passed and failed, students were guessing what were the failed test cases and adjusted their code to correct for those specific test cases rather than think about their overall solution.
(2) some students who had gotten a good start but whose code did not compile or whose code failed all the test cases would give up because of a low score.
(3) some students who had gotten a sufficiently high score would stop working on their code because they were more motivated by getting a "good enough" score than by arriving at a complete solution.

For both semesters when it was used, Autolab was configured to test the students' code using 20 test cases for *PayFriend* and 30 test cases for *TwoSmallest*, generated as described in Section 4.1. Again, the students were allowed to submit each assignment three times without penalty, then with a 5 point penalty for each subsequent submission. During Spring 2017, the only feedback that the students received for their submission was a "progress signal" as described below. During Spring 2018, feedback for each submission included a progress signal and hints generated as described in Section 4. In our assessment of the usefulness of the hints, we focused on comparing the two semesters that were most similar, Spring 2017 and Spring 2018, although, for completeness, we also include the data and results from Spring 2016.

The above observations lead to the following changes for the Spring 2017 and 2018 semesters:

(1) After each submission, students were given a *progress signal* instead of a score. Scores were revealed only after the assignment due date when students were not allowed to submit anymore.
(2) Scores were calculated using a scheme in which the weighted sum of the test results was added to a base score (Table 1).
(3) During Spring 2018, as part of the feedback, students also received hints associated with the error class corresponding to the errors in their submission.

[2]Students were not given information on the test cases themselves.

Progress signal. To discourage students from targeting their code to specific test cases, starting with Spring 2017, instructors changed the feedback given to students to a signal indicating overall progress instead of an actual score or information about the number of passed and failed test cases. In this scheme, a submission would be tagged with a red "light" for a score below 20, a yellow "light" for a score between 20 and 60 for *PayFriend* and between 20 and 80 for *TwoSmallest*, and a green "light" for a score above 60 for *PayFriend* and above 80 for *TwoSmallest*. Instructors explained to the students that red meant that a submission was very far from a correct solution, yellow meant that a submission was on the right track but was still giving the wrong answer for many test cases, and green meant that the submission was definitely on the right track, but there was no guarantee of a perfect score or that the submission had passed all the test cases. The latter was used to encourage students to think about comprehensive test plans rather than gaming the system to try to get a perfect score. The final scores were released to the students after the assignment deadline.

Scheme used for calculating scores. For each error class, instructors assigned a base score, shown in Table 1. Scores for each submission were calculated by adding the base score associated with the error class to the weighted sum of the passed test cases. The weights used for each passed test case were 1 for *PayFriend* and 1.5 for *TwoSmallest*. For example, if a code submission for *PayFriend* was labeled with COND and it passed 15 test cases, its score became $50 + 15 * 1 = 65$. This grading scheme made the grades approximately follow a normal distribution because it moved the scores on submissions that did not pass any test cases from zero to some partial credit and submissions that were near completion from a near perfect score to a score that was less than 85 for each assignment. This scheme increases the scores for early but significant efforts to encourage students to keep trying and it also increases the value of "solving the few remaining bugs" to encourage students who are doing well to keep trying.

5.2 Significance Test

In our analysis, we use Pearson's χ^2 test to determine the significance of the difference between two proportions, and the Kolmogorov-Smirnov test to determine the significance of the difference between two cumulative distributions (e.g., comparing numbers from Spring 2017 and 2018). We use a significance level of 0.05.

5.3 Accuracy of the Error Classification

We begin our evaluation by assessing the accuracy of the error classifiers. These classifiers were produced from submissions collected during Spring 2016 and run on submissions from Spring 2017. After classifying Spring 2017 submissions, we asked three undergraduate students who had previously taken our Introduction to Computer Science course to carefully review the submissions and the errors produced by the classifiers and assess the accuracy of the classifications and the potential efficacy of the corresponding hints. At the time this paper was written, these students were enrolled in our computer science program or had recently graduated with a computer science degree. We believe that having done the assignments themselves while taking the course gave these students a good perspective in the evaluation of the automatic error classification.

Human evaluation approaches can be subject to biases (e.g., knowing categories ahead of time can lead to conformity bias). We took some measures to reduce or minimize such biases. In particular, we asked the evaluators to assess the code and write an appropriate hint before deciding if the "autogenerated" hint was suitable. The hints they wrote were very similar to the autogenerated hints. For example when a student got one of the formulas wrong, one of the evaluator's hint was "check your math." Hints were not given when the evaluators took the class, and so they were not biased that way. Finally, the evaluation presented here only provides evidence for the accuracy

of the error classifiers. Ultimately, the impact of the error classification and of the corresponding hints on the students, reported below, is the most important.

To evaluate the classifiers, each erroneous submission and its corresponding error class and hint was labeled either *Correct*, *Partially Correct*, or *Incorrect*. The *Correct* label 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. The *Partially Correct* label 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 or would not have made sense. Each submission was inspected and evaluated by at least two people. The results were analyzed by one of the authors (who did not do the labeling) to resolve conflicts and to ensure consistency between evaluations. We computed the inter-rater reliability score by assigning 1 to all the instances where the reviewers agreed and 0 otherwise, summed for all the submissions and divided by the total number of submissions. The inter-rater reliability score we obtained was 93%.

Table 3 summarizes the results of the manual evaluation of the accuracy of the error classification. In particular, it shows that the classification was correct for 91.5% of the code submissions for *PayFriend* and for 87.5% of the code submissions for *TwoSmallest*. Moreover, the manual inspection of the submissions and their classification for both assignments revealed errors that would have been difficult to detect with the gray box testing we used. We call it "gray box testing" because we had limited knowledge about each student's code at the time when the test cases were designed. Furthermore, the test cases were designed for a full path coverage of the reference solution as described in Section 4.1, which may or may not be close to the student solutions.

To demonstrate the inherent limitations of gray box testing for *PayFriend*, consider the student code in Figure 1. This code passes only the test cases for inputs smaller than 100. The classifier labels it as FORM (see Table 1), because it is using incorrect calculations inside three out of the four intervals. The student seems to have a poor understanding of control flow and of the difference between consecutive *if*s and *if-else*. An accurate hint would tell the student that, for payments greater than 100, the code may change the fee multiple times which would be considered incorrect behavior. The simplest fix is to add an *else* after each *if* and to put the following conditional expressions inside that *else*. The code example in Figure 1 shows that, because of the semantic and functional brittleness of code [38], slight deviations in the code's block structure can result in an error signature that the classifier is not able to properly label. As such, to improve the accuracy of the classifier, we would need more powerful methods of assessment (for example, code analysis techniques).

Despite the above limitations, the high accuracy of the error classification and corresponding hints provides strong evidence that they are appropriate for the vast majority of erroneous submissions. Thus, we deployed the classifiers and corresponding hints during the Spring 2018 semester.

5.4 Usefulness of the hints

We compare the data from Spring 2017 and Spring 2018 because these two semesters are the most similar – they used the same autograder, the same progress signal ("light" colors instead of scores), the same grading scheme, and the same number of submissions allowed without penalty. The difference between the two semesters was in whether students were given hints or not. We added data from Spring 2016 for interest, but do not include it in the analysis of the usefulness of the hints because the two semesters when students received hints (Spring 2016 and Spring 2018) were very different. As explained above, during Spring 2016, a different autograding system was used that associated a hint to each test case and a different formula was used to compute the score for each submission. Note that the scores shown for Spring 2016 in this section are not the actual

```
public class PayFriend {
    public static void main(String [] args) {
        double payment=IO.readDouble();
        double fee=0;

        if (payment>15000) {
            fee+=(10000*0.01)+(5000*0.02)+((payment-15000)*0.03)+5;
        }
        if (payment>10000) {
            fee+=(10000*0.01)+((payment-10000)*0.02);
        }
        if (payment>1000) {
            if ((payment*0.01)>15) {
                fee+=payment*0.01;
            } else {
                fee+=15;
            }
        }
        if (payment>100) {
            if ((payment*0.03)>6) {
                fee+=payment*0.03;
            } else {
                fee+=6;
            }
        }
        if (payment<100){
            fee+=5;
        }

        IO.outputDoubleAnswer(fee);
    }
}
```

Fig. 1. Example of student code that was improperly labeled by our classifier.

scores that the students received, but rather the scores that students would have received with the grading scheme from Spring 2017 and Spring 2018 (Section 5.1). *Overall, the data from Spring 2016 is consistent with observations from the comparison between Spring 2017 and Spring 2018, that is, the performance of students on programming assignments improves when they receive hints.*

We use two measures to evaluate the usefulness of the hints generated using our framework: *empirical usefulness* and *perceived usefulness*. We say that hints are *empirically useful* if there are statistically significant differences between the semester when students did not receive hints (Spring 2017) and the semester when students received hints (Spring 2018) in terms of resubmission rate, final score, score difference between the first and last submission, and score difference between consecutive submissions. We determine whether hints are *perceived* as *useful* from the students' responses to a survey completed at the end of the Spring 2018 semester. We examine the distribution of student responses to specific questions, their comments in response to the hints, and anecdotal information from instructors.

5.4.1 Empirical usefulness of the hints. To assess the empirical usefulness of the hints, we analyzed the differences between Spring 2017 and Spring 2018 in terms of the cumulative percentage of students who resubmitted their assignment, the final scores, the score differences between the students' first and final submission, and the score difference between consecutive submissions.

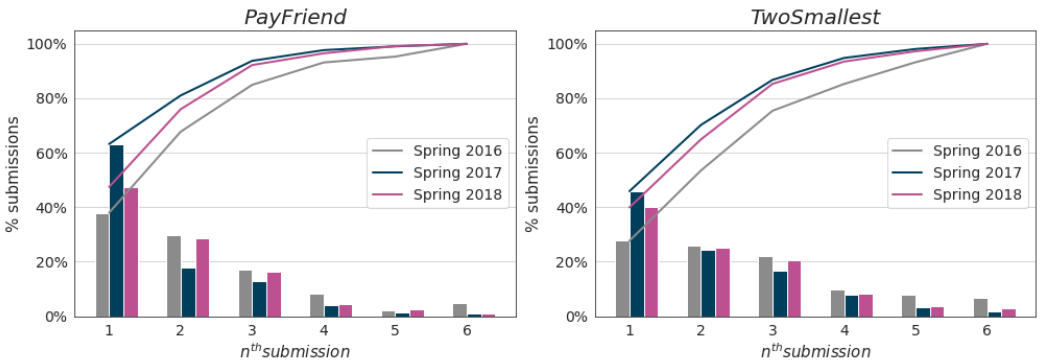

Fig. 2. Cumulative distributions and histograms showing the percentage of submissions that are the n^{th} submission from a student out of all submissions. A "lower" CDF indicates higher percentages of later submissions, corresponding to students submitting *more* times. For Spring 2016, students were allowed to submit up to 5 times without penalty, compared to 3 times for 2017 and 2018, which is likely the reason for the observed highest rates of re-submissions in 2016.

First, we observe an increase between Spring 2017 and Spring 2018 in the number of assignment resubmissions, both in terms of the percentage of students who resubmitted (shown in the last row of Table 2) and the cumulative distribution of the percentage of submissions that are the n^{th} submission from a student out of all submissions (shown in Figure 2). The last row in Table 2 shows that the percentage of students who resubmitted increased between Spring 2017 and Spring 2018, but the difference is statistically significant only for *PayFriend*. This increase is represented in a rightward translation of the line, up to three submissions. Then, the lines converge because of points being deducted after more than 3 submissions, that is *5 points* of the student's grade were subtracted from the score for every submission after the third one. We believe that the higher resubmission rate for Spring 2016 can be explained by the fact that students were able to submit their code 5 times without a penalty instead of 3 times during the other semesters.

The rate of resubmissions is important but it gives only one piece of evidence for the usefulness of the hints. We further look at additional usefulness indicators.

We define two other measures which we believe point to the empirical usefulness of the hints: the *efficacy of the hints* as the overall progress students made between their first and last submission measured as a difference in score and the *efficiency of the hints* as the progress with each resubmission measured, again, as a difference in score. Our assumption is that the increase in overall score and the increase in score with each resubmission are evidence of the usefulness of the hints.

Efficacy of the hints. We measured the efficacy of the hints by the progress students made toward completing their assignment when they were given hints after each submission. We evaluated progress by the percentage of students who successfully completed the assignment and by the increase in score between the initial and final submissions.

We noticed a substantial difference between the percentages of students who successfully completed the assignment in one submission, the percentages of students who successfully completed

Table 4. Percentages of students who completed (that is, received a prefect score for) each assignment and the number or submissions it took (one or multiple). † marks a statistically significant change between Spring 2017 and Spring 2018.

	PayFriend			*TwoSmallest*		
	Spring 2016	**Spring 2017**	**Spring 2018**	**Spring 2016**	**Spring 2017**	**Spring 2018**
Students who successfully completed the assignment in one submission	26.8%	30.4%	22.5%	15.6%	14.4%	22.5%
Students who successfully completed the assignment after resubmitting	34.4%	11.6%	31.8%†	30.9%	9.2%	28.8%†
Total students who successfully completed the assignment	61.2%	42.0%	54.3%	46.5%	23.6%	51.3%†
Total students who did not complete the assignment	38.8%	58.0%	45.7%	53.5%	76.4%	48.7%†

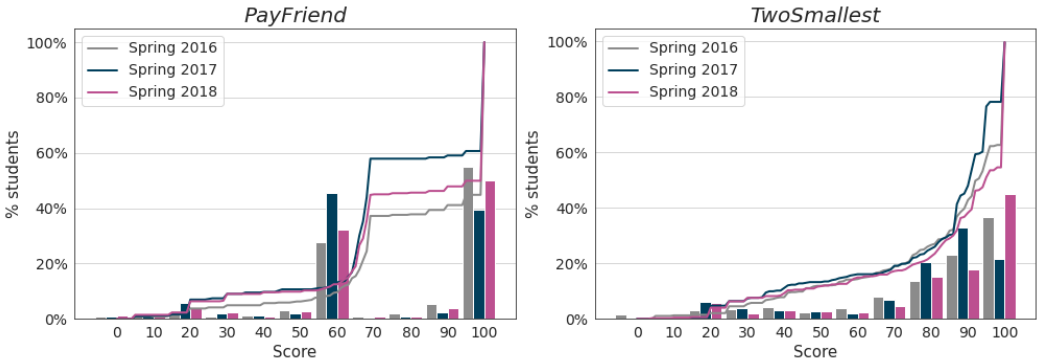

Fig. 3. The cumulative distribution of students' final scores for each assignment along with the histogram showing the distribution of grades in buckets of 10 points. For example, at 60, the bars show the percentages of scores between 60 and 69. At 100, the bars indicate the percentages of scores equal to 100.

the assignment using multiple submissions, and the percentages of students who did not complete the assignment (Table 4). Although we see some differences between the percentages of students who successfully completed the assignments in one submission, analyzing them goes beyond the scope of our study given that the two populations of students come from two different semesters. These differences are also not statistically significant. When hints were provided, the percentage of students who successfully completed their assignment after resubmitting almost tripled for both assignments compared to the semester during which no hints were provided (these numbers are shown in the second row of Table 4), and the difference is statistically significant.

Figures 3 and 4 show the distributions of the students' final scores. In particular, for students who submitted multiple times (Figure 4), for *PayFriend*, about 5% more students received a final score between 20 and 55 and about 20% more students received scores above 70 during Spring 2018 than during Spring 2017. This difference is statistically significant. Interestingly, the CDFs for *PayFriend*

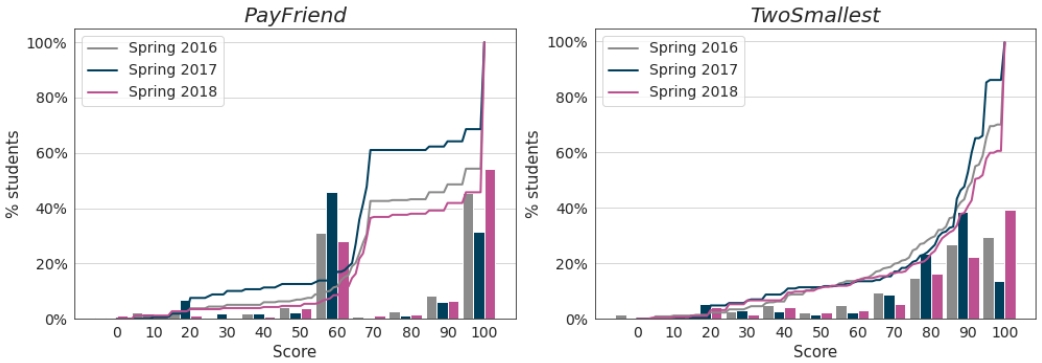

Fig. 4. The cumulative distribution of final scores for students who submitted multiple times along with the histogram showing the distribution of grades in buckets of 10 points. For example, at 60, the bars show the percentages of scores between 60 and 69. At 100, the bars indicate the percentages of scores equal to 100.

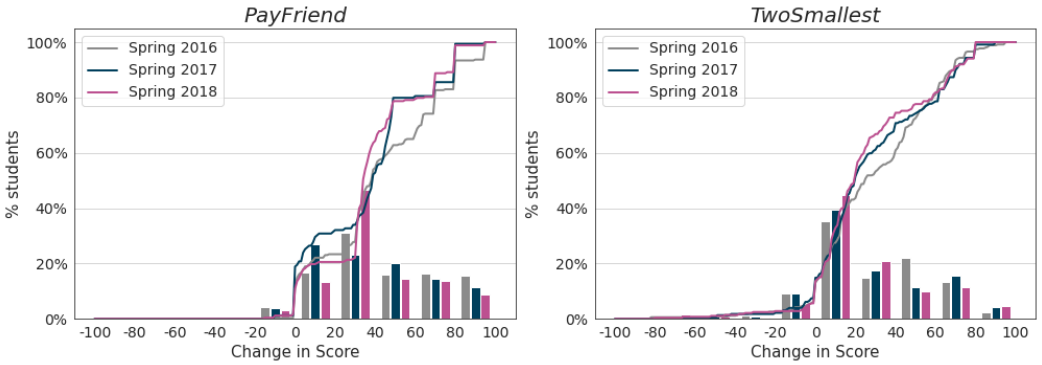

Fig. 5. Cumulative distributions and histograms of differences in score between the final and first submissions for all students who submitted multiple times. Numbers are shown for 20 percentage point increments (for example, at 30, the bars show the percentages of students with a difference in score between 20 and 40. The *x-axis* indicates the score difference and the *y-axis* indicates the percentage of students whose score difference matches x.

show sharp rises around 20, 60 and 100. We believe that these sharp rises (and the relative flatness in between) can be attributed to the fact that: (1) the assignment is simple, and so students tend to earn points in large chunks, and (2) 20 and 60 are boundary scores when the progress signal changes from red to yellow and from yellow to green for Spring 2017 and Spring 2018. Many of the students stopped working on their code when they received the green light signal, even though the instructors explained that getting a green light did not guarantee a perfect score. Students understood this guideline much better for TwoSmallest, which explains why more students received higher scores in the "green light" range for the second assignment. For *TwoSmallest*, we see a similar trend with about 10-20% more students receiving final scores above 85 during Spring 2018 than during Spring 2017. These differences in the cumulative distribution of the students' final scores suggest that the hints were useful.

Table 5. Percentages of students who were able to fix all their errors after one resubmission (out of all the students who did not achieve a perfect score on their first submission). † marks a statistically significant difference between Spring 2017 and Spring 2018.

	PayFriend†	***TwoSmallest***
Spring 2016	27.34%	19.86%
Spring 2017	22.83%	21.00%
Spring 2018	37.62%	23.90%

Figure 5 shows, for each assignment, three cumulative distributions, one for each semester of interest, of the differences in score between the final and first submissions. During the semester when hints were given, a higher percentage of students made moderate progress as reflected in an increase in their score between 30 and 45 for *PayFriend* and between 20 and 40 for *TwoSmallest*.

Finally, we note that it appears that hints are most useful for students who are close to getting a perfect score. In addition, for *PayFriend*, the hints seem to also help students with a score between 20 and 30. These are students who do not use the IO module properly; the IO module is an interface for receiving input and supplying output.

Efficiency of the hints. Intuitively, the *efficiency of the hints* has to do with how much quicker students make progress toward completing the assignment when they receive hints compared to when they do not receive hints.More specifically, for all the students who submitted multiple times, we used their submission history to create pairs *(i,i+1)* of consecutive submissions. Next, we counted all the pairs for which the first submission *i* and the following submission *i+1* were labeled with the same error class. We show the results in Table 5 and Figure 6.

Firstly, we observe an increase in the probability that a student will complete their assignment (that is, their submitted code will pass all the test cases) with every resubmission as reflected by an increase in the percentage of students who fixed all their errors in one resubmission for the semester when hints were provided, which is statistically significant for *PayFriend*. In Table 5, for each assignment, we show the percentages of students who fixed all their errors in one resubmission, by assignment. To calculate these percentages, we counted all the pairs of consecutive submissions for which the first submission *i* was labeled with an error class and the following submission *i+1* had no error. During the semester with hints, more students were able to complete their assignment after one round of hints, but only for *PayFriend* was the difference between semesters statistically significant.

Secondly, we see a decrease between Spring 2017 and Spring 2018 in the percentages of submissions that were stuck in the same error class from one submission to the next except for INS and COND for *PayFriend* and COMP and UPDT for *TwoSmallest* as shown in Figure 6. We calculated the percentage of submissions stuck in each error class by using the number of pairs of submissions which started and ended in that error class and divided it by the number of all the pairs of submissions for which the first submission was in that error class. For *PayFriend*, we see progress for all error classes, except for INS, which is the error class in which the student's code was saved under the wrong filename, the class or method names were wrong, and COND, which is the error class in which the student's code failed one or multiple test cases for boundary values. It is difficult to point out these errors without giving away the answer or revealing the test cases, which we are trying to avoid. For *TwoSmallest*, we see a smaller progress for the majority of the error classes, with a slight increase for UPDT and a bigger increase for COMP. We think the smaller progress is linked to the greater complexity of the assignment. For *TwoSmallest*, the solution strategy comprises three main tasks: differentiate valid entries from terminating values (SEQ), initialize the variables to store the

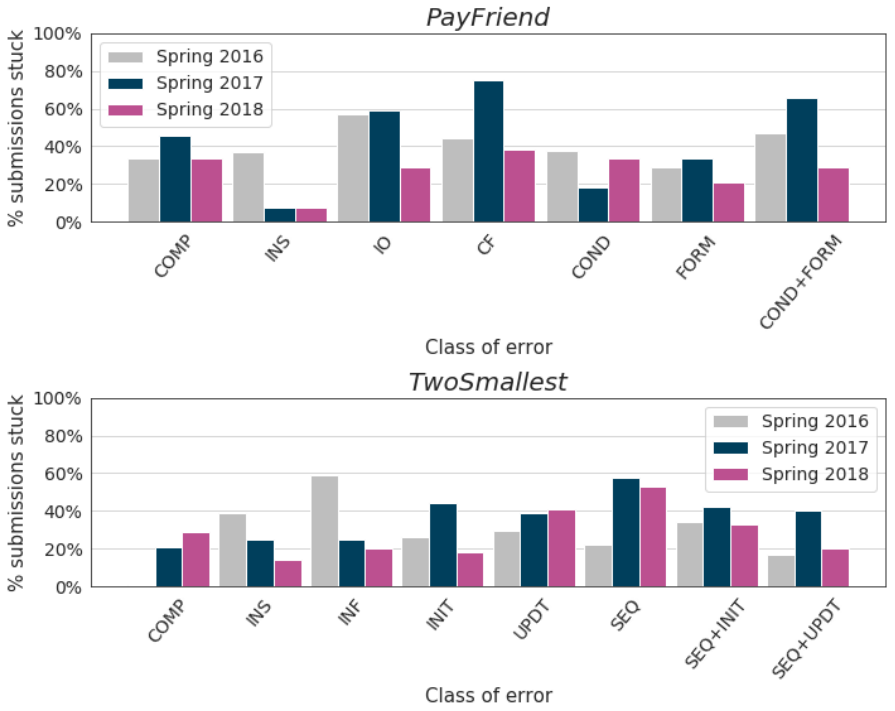

Fig. 6. Histograms showing the distribution of submissions stuck in an error class from one submission to the next. The *x-axis* indicates the error class and the *y-axis* indicates the percentage of students who were not able to make progress from the previous submission. For Spring 2016, COMP column looks like it's missing because of the graph scale and the fact that it is only 1%.

minimum values (INIT), and update these variables (UPDT). INIT and UPDT point to parts of the strategy that are separate, that is INIT ends when the first two valid values are read, and UPDT starts with the third valid value. SEQ points to the part of the strategy that is interleaved with INIT and UPDT, but the behavior of SEQ changes from INIT to UPDT as follows: during INIT, the terminating values are discarded and new values are being read, whereas during UPDT, the read stops when the terminating value is read. To test for errors in the class INIT, students need to test for all the possible combinations of two numbers with repetition, that is 3 or 4 test cases; the two numbers represent the first two valid entries in the sequence. For the error class UPDT, in the least, students need to test for all the possible combinations of three numbers with repetition (three or more, anything more than two), that is 12 or more different test cases. Finally, an additional four or more test cases are required to test the behavior of UPDT (incorrect update of min values, Table 1), using number sequences with additional terminating values interleaved between the first input (the terminating value), and the first valid input (different from the terminating value), and number sequences with additional terminating values between the first two valid entries. This analysis shows that *TwoSmallest* is far more complex than *PayFriend*. However, for both assignments, we see that fewer submissions were stuck in the same error classes during Spring 2018 than during Spring 2017.

Finally, we note that hints appear to be helping at least some of the students make progress in addressing errors in their code. As shown above, for the semester with hints, there was a decrease

Table 6. Assignment completion rates for all students and for students who completed the TAM survey. "One submission" encompasses students who have submitted each assignment only once; "multiple submissions" means that students submitted at least one of the assignments multiple times.

	Assignment completion		*All Students (459 total)*	*Survey Respondents (287 total)*
	PayFriend	***TwoSmallest***		
One Submission	yes	yes	22 (4.8%)	15 (5.2%)
	yes	no	16 (3.5%)	9 (3.2%)
	no	yes	23 (5.0%)	13 (4.5%)
	no	no	34 (7.4%)	14 (4.9%)
Multiple Submissions	yes	yes	135 (29.4%)	99 (34.5%)
	yes	no	77 (16.8%)	43 (15.0%)
	no	yes	59 (12.9%)	41 (14.3%)
	no	no	93 (20.3%)	53 (18.3%)

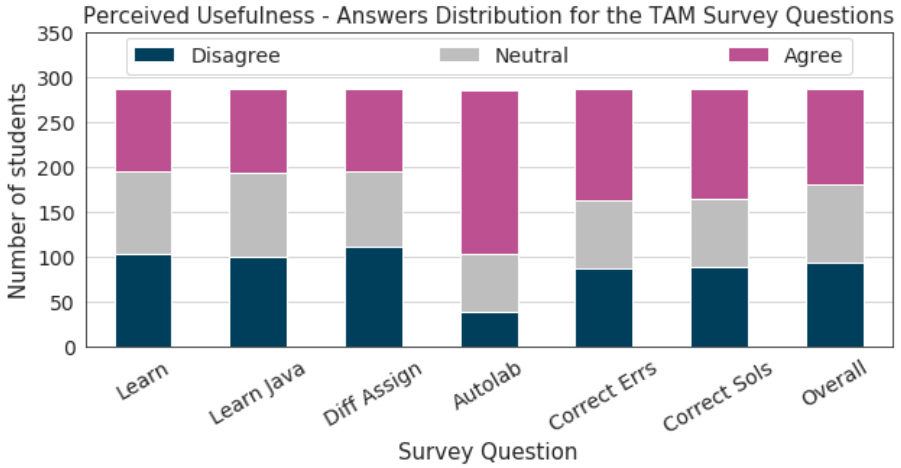

Fig. 7. The distribution of student answers to questions in our survey: Learn - The hints provided by Autolab helped me learn, Learn Java - The hints provided by Autolab were useful for learning to program in Java, Diff Assign - The hints provided by Autolab were helpful when I was working on a difficult assignment, Autolab - Autolab improved my abilities as a Java programmer, Correct Errs - The hints provided by Autolab helped me correct my errors, Correct Sols - The hints provided by Autolab helped me write correct assignment solutions, Overall - Overall, I found the hints provided by Autolab very useful.

in the percentage of students who were stuck in the same error class, and more students were able to complete their assignment after resubmitting just once, compared to the semester without hints.

In conclusion, the hint system we implemented in Autolab as proof of concept for our proposed framework appears to help students make progress from one submission to the next, as well as improve their final grade. However, they do not seem to help students equally: it appears to provide more help to students whose code is in certain error classes and those who are near the completion of their assignment.

5.4.2 Perceived usefulness of the hints. To assess the perceived usefulness of the hints, we look at the distribution of the students' responses to a survey completed at the end of the Spring 2018

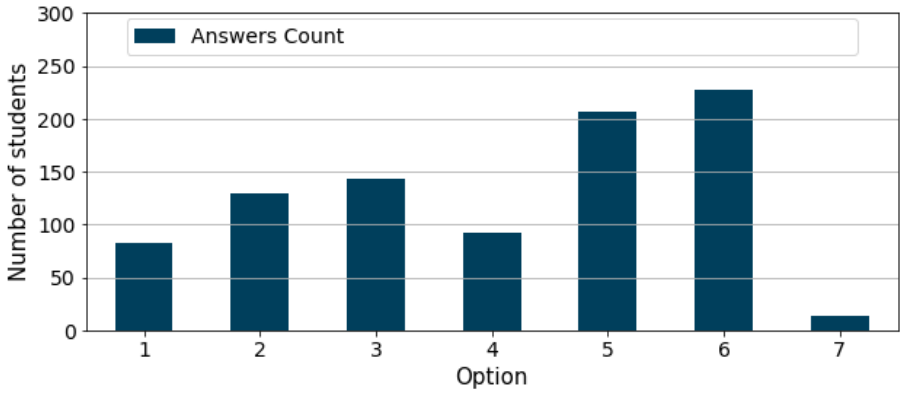

Fig. 8. Perceived usefulness of Autolab and the hints, as reflected by the responses to the TAM survey; The plot shows what Autolab features students thought could be improved: 1 - Having more help when learning how to use Autolab, 2 - Having more assistance when having difficulties with Autolab, 3 - Having more help when something goes wrong in Autolab, 4 - Not having to use the IO module, 5 - The wording of the hints, 6 - The information provided by the hints, 7 - other, typed by the student.

Table 7. Percentage of students who agreed with survey statements. "One Submission" means that students submitted each assignment once and "Multiple Assignments" means that students submitted at least one assignment multiple times. Note that students submitting just once would still have gotten hints unless they received a perfect score.

	One Submission	**Multiple Submissions**	
		Neither Completed	**Completed One or Both**
Having more help when learning how to use Autolab	35.3%	**35.8%**	21.3%
Having more assistance when having difficulties with Autolab	45.1%	**50.9%**	39.3%
Having more help when something goes wrong in Autolab	47.1%	**56.6%**	44.8%
Not having to use the IO module	29.4%	28.3%	30.1%
The wording of the hints	**60.8%**	**67.9%**	**71.6%**
The information provided by the hints	**60.8%**	**69.8%**	**82.0%**
Other	5.9%	**9.4%**	3.3%
No Answer	0.0%	0.0%	1.1%

semester, their comments on the hints in Autolab, and the anecdotal information provided by the instructors of the course.

Survey Results. We built a survey using the Technology Assessment Model (TAM) [39] to assess the perceived usefulness and ease of use of Autolab, with focus on the hints.[3] At the end of the Spring 2018 semester, we asked the students to complete this survey for a small credit toward their

[3]A copy of the survey can be found here: https://drive.google.com/file/d/1rX5tmIY38Eb3NqmKO4xo_7d3V8ZkdkEx/view?usp=sharing

final grade. Out of the 459 students who submitted at least one of the assignments in Autolab, 287 students completed the survey. Table 6 shows assignment completion rates for survey respondents and for all students. An assignment is considered complete when it passes all the test cases and receives a score of 100. Given that the two distributions ("All Students" and "Survey Respondents") are similar, we conclude that the sample of survey respondents is representative of the student population who submitted their assignments to Autolab during Spring 2018. Next, we present results from the analysis of the students responses.

We start by using 7 of the survey questions, 6 of them asking about the hints and one of them asking how much the students agreed that Autolab improved their abilities as Java programmers (Figure 7). Our first observation is that nearly two thirds of the students agreed that Autolab improved their abilities as Java programmers, whereas only between about one third and about one half of the students agreed that the hints were useful. Among questions about the usefulness of hints, a slightly higher number of students (about *10%* more) agreed that the hints were useful for correcting errors and for writing correct assignment solutions. It is unclear whether students who did not find the hints useful had that perception because they were looking for more detailed and explicit feedback. Such feedback would be against the spirit of our approach: we want to give hints that enable students to *think* about the problem and then make progress, rather than follow detailed instructions on how to correct their code. It would be interesting to obtain more information about the students' thoughts on this matter in the future.

Secondly, we observe that about two thirds of the students thought that the wording of the hints and the information provided by them needed improvement (Figure 8). When combining the percentage of students who agreed with the survey statements broken down by number of submissions (one vs multiple) and the assignment completion rates shown in Table 7, we see that the most selected options for improvement could be categorized as: those regarding using the autograding system and those regarding the hints. Students who submitted multiple times but did not complete any of the assignments expressed more often that the autograding system needed improvement, whereas students who completed at least one of the assignments more often expressed that the language of the hints and the information provided by the hints needed improvement. Finally, fourteen students selected *Other* to answer the question of what needed improvement and typed their suggestions: five students mentioned the way files were uploaded in Autolab, five students mentioned the color "lights" shown instead of scores, three students wrote that test cases would have been more useful feedback than the color "lights", and finally two students mentioned that Autolab was slow, especially when approaching the assignment submission deadlines.

Thirdly, in Figure 9, we see that the likelihood that a student perceived the hints as helpful increased with their score as shown in the graphs titled *Learn* and *Correct Errs*, and that more students who obtained higher scores had neutral answers to the survey questions as shown in the graph titled *Neutral Answers*. The score is shown as the light color corresponding to the average between the student's score on the final submission for *PayFriend* and the score on the final submission for *TwoSmallest*. Moreover, we see that, as the average number of submissions increased, students were more likely to agree that the hints were useful in correcting their errors (second row, right side graph in Figure 9), and that fewer of them had neutral answers (forth row, right side graph).

Students' comments on hints. In Autolab, students had the option to write a comment or feedback every time they received a hint. Students wrote 25 comments for *PayFriend* and 19 comments for *TwoSmallest*. Most commonly, the students' comments expressed disagreement with or confusion about the hints. One student's comment expressed that the error in the code had been correctly

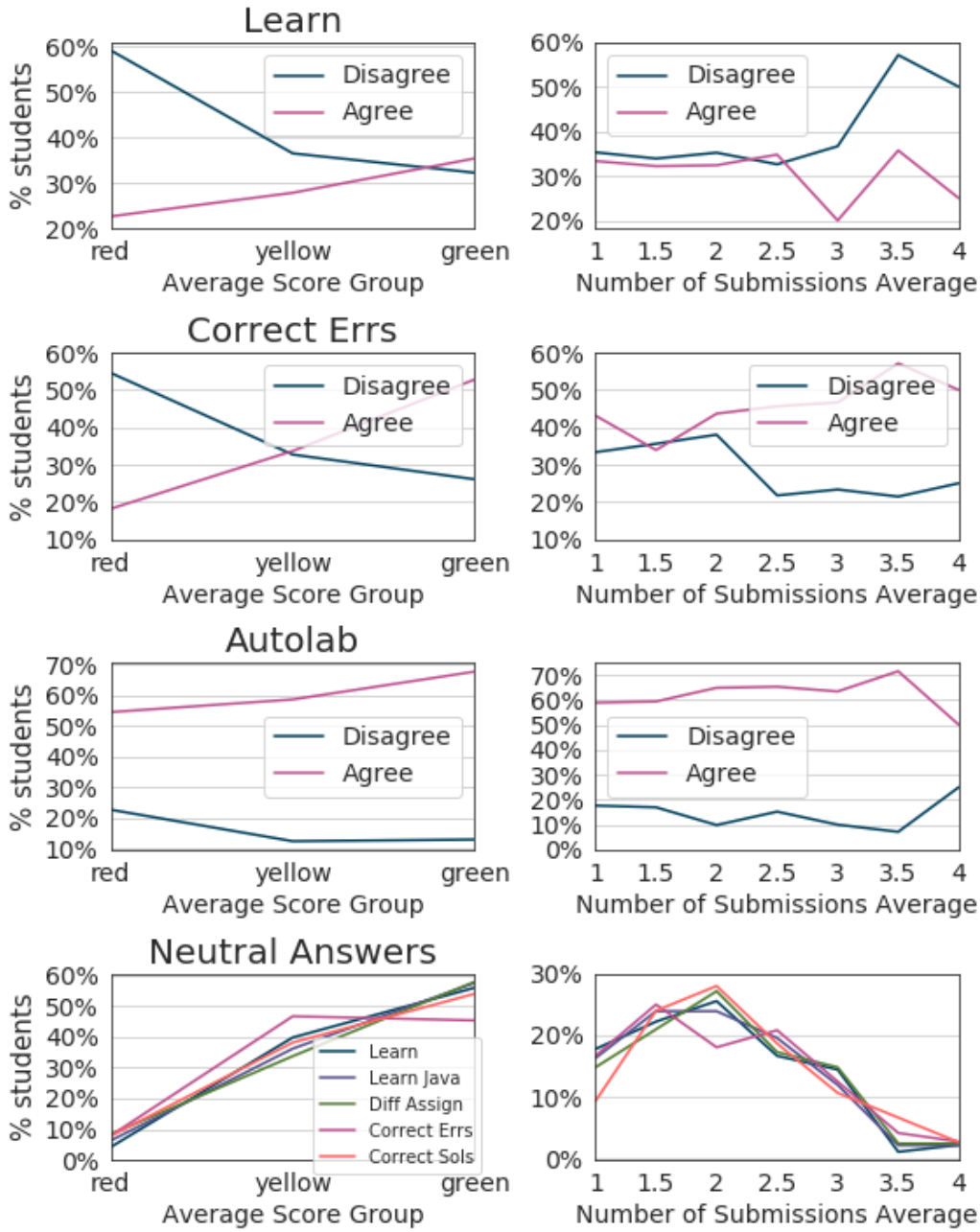

Fig. 9. Distribution of survey responses by average grades and by average number of submissions. The first column shows the distributions of the average score between the final submissions for PayFriend and TwoSmallest from lowest (red) to highest (green) by survey responses. The second column shows the distribution of the average number of submissions by survey responses.

identified by the hint. Most often, the students said that they disagreed with the hints they received because their code had passed the test cases and they often asked for the test cases that their code had failed. They also asked for more details about their errors, such as what was the line number in their code containing the error. Finally, many students thought that the assignment descriptions were not clear.

Anecdotal information from instructors. Overall, the instructors communicated that they found the hints to be useful, but that they were not enough to correct the students' thinking. They offered suggestions for improving the hints such as: changing the wording of the hints, improving the accuracy of the hints, and improving the content of the hints. For example, they thought that providing failed test cases along with the wording of the hint would be very useful for the students. However, figuring out what is the best information and the best way to present it to students is an open question beyond the scope of this research. Our aim was to write hints that gave students enough information to correct the logical errors in their code without giving away the answers. Moreover, the hints were provided only for a subset of the assignments and instructors reported that students were asking for hints for assignments that did not provide hints, suggesting that students at least perceived the hints to be useful (whether they actually were or not).

In conclusion, both students and instructors were in agreement that the hints were useful, but that they could be improved, in particular the information provided in them as well as their wording. We plan to explore these directions in our future work.

6 CONCLUSIONS

In this paper, we presented a methodology that 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 on future submissions. This process can also give instructors insight into how to adjust class material to address knowledge deficiencies. We have applied our methodology to two assignments and found some evidence suggesting that the hints provide useful feedback for many students to make progress after submitting incorrect programs. The assignments used in our analysis are relatively simple assignments used in an introductory computer science course. In our future work, we plan to explore the application of our methodology to more complex programming assignments.

REFERENCES

[1] 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.

[2] Computing Research Association, "Generation CS: Computer Science Undergraduate Enrollments Surge Since 2006." https://cra.org/data/Generation-CS/.

[3] S. H. Edwards and M. A. Perez-Quinones, "Web-CAT: Automatically Grading Programming Assignments," in *ACM SIGCSE Bulletin*, vol. 40, 2008.

[4] D. Milojicic, "Autograding in the Cloud: Interview with David O'Hallaron," *IEEE Internet Computing*, 2011.

[5] 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.

[6] A. Nguyen, C. Piech, J. Huang, and L. Guibas, "Codewebs: Scalable homework search for massive open online programming courses," in *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14, (New York,

NY, USA), pp. 491–502, ACM, 2014.
[7] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas, "Learning program embeddings to propagate feedback on student code," *arXiv preprint arXiv:1505.05969*, 2015.
[8] E. L. Glassman, A. Lin, C. J. Cai, and R. C. Miller, "Learnersourcing personalized hints," in *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing*, CSCW '16, (New York, NY, USA), pp. 1626–1636, ACM, 2016.
[9] 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.
[10] V. J. Shute, "Focus on formative feedback," *Review of Educational Research*, vol. 78, no. 1, pp. 153–189, 2008.
[11] 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.
[12] S. Marwan, J. Jay Williams, and T. Price, "An evaluation of the impact of automated programming hints on performance and learning," in *Proceedings of the 2019 ACM Conference on International Computing Education Research*, pp. 61–70, ACM, 2019.
[13] B. P. Woolf, *Building Intelligent Interactive Tutors: Student-centered Strategies for Revolutionizing e-Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.
[14] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," *SIGPLAN Not.*, vol. 48, no. 6, 2013.
[15] R. E. Mayer, J. L. Dyck, and W. Vilberg, "Learning to program and learning to think: What's the connection?," *Commun. ACM*, vol. 29, pp. 605–610, July 1986.
[16] K. Brennan and M. Resnick, "New frameworks for studying and assessing the development of computational thinking," in *Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada*, pp. 1–25, 2012.
[17] N. Le, F. Loll, and N. Pinkwart, "Operationalizing the continuum between well-defined and ill-defined problems for educational technology," *IEEE Transactions on Learning Technologies*, vol. 6, no. 3, pp. 258–270, 2013.
[18] 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.
[19] 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.
[20] J. McBroom, I. Koprinska, and K. Yacef, "A survey of automated programming hint generation - the hints framework," *ArXiv*, vol. abs/1908.11566, 2019.
[21] L. D'Antoni, R. Samanta, and R. Singh, "Qlose: Program repair with quantitative objectives," in *International Conference on Computer Aided Verification*, pp. 383–401, Springer, 2016.
[22] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: Data-driven feedback generation for introductory programming exercises," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, (New York, NY, USA), pp. 481–495, ACM, 2018.
[23] R. Suzuki, G. Soares, E. Glassman, A. Head, L. DâĂŹAntoni, and B. Hartmann, "Exploring the design space of automatically synthesized hints for introductory programming assignments," in *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA âĂŹ17, (New York, NY, USA), p. 2951âĂŞ2958, Association for Computing Machinery, 2017.
[24] J. Stamper, T. Barnes, L. Lehmann, and M. J. Croy, "The hint factory: Automatic generation of contextualized help for existing computer aided instruction," 2008.
[25] B. Paaßen, B. Hammer, T. W. Price, T. Barnes, S. Gross, and N. Pinkwart, "The continuous hint factory - providing hints in vast and sparsely populated edit distance spaces," *CoRR*, vol. abs/1708.06564, 2017.
[26] R. Zhi, S. Marwan, Y. Dong, N. Lytle, T. W. Price, and T. Barnes, "Toward data-driven example feedback for novice programming," in *EDM*, 2019.
[27] A. Gerdes, B. Heeren, J. Jeuring, and L. T. van Binsbergen, "Ask-elle: an adaptable programming tutor for haskell giving automated feedback," *International Journal of Artificial Intelligence in Education*, vol. 27, no. 1, pp. 65–100, 2017.
[28] P. J. Guo, "Codeopticon: Real-time, one-to-many human tutoring for computer programming," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, (New York, NY, USA), pp. 599–608, ACM, 2015.
[29] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller, "Overcode: Visualizing variation in student solutions to programming problems at scale," *ACM Trans. Comput.-Hum. Interact.*, vol. 22, pp. 7:1–7:35, Mar. 2015.
[30] E. L. Glassman, L. Fischer, J. Scott, and R. C. Miller, "Foobaz: Variable name feedback for student code at scale," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, UIST '15, (New York, NY,

USA), pp. 609–617, ACM, 2015.

[31] J. B. Moghadam, R. R. Choudhury, H. Yin, and A. Fox, "Autostyle: Toward coding style feedback at scale," in *Proceedings of the Second (2015) ACM Conference on Learning @ Scale*, L@S '15, (New York, NY, USA), pp. 261–266, ACM, 2015.

[32] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "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 âĂŹ10, (New York, NY, USA), p. 86âĂŞ93, Association for Computing Machinery, 2010.

[33] 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.

[34] S. Kaleeswaran, A. Santhiar, A. Kanade, and S. Gulwani, "Semi-supervised verified feedback generation," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, (New York, NY, USA), pp. 739–750, ACM, 2016.

[35] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: Suggesting solutions to error messages," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, (New York, NY, USA), pp. 1019–1028, ACM, 2010.

[36] 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.

[37] 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*, vol. 27, 2017.

[38] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *arXiv preprint arXiv:1709.06182*, 2017.

[39] F. D. Davis, "Perceived usefulness, perceived ease of use, and user acceptance of information technology," *MIS Quarterly*, vol. 13, no. 3, pp. 319–340, 1989.

A APPENDIX

A.1 Assignment Descriptions

Pay Friend. Write your code in the file PayFriend.java, your file has to have this exact name with P and F capitalized. You must use the IO module to read the input and to output your answer. Imagine that you work for a payment processing service called PayFriend. PayFriend charges money receivers the following fees:

The first $100 has a flat fee of $5. Payments over $100 (but under $1000) have a fee of 3% or $6, whichever is higher. Payments of $1,000 (but under $10,000) and over have a fee of 1% or $15, whichever is higher. Payments of $10,000 and over are subject to (fees as follows): The first $10,000 have a fee of 1% The next $5,000 have an additional fee of 2% Anything more will demand an additional fee of 3% For example, an payment of $40,000 would be subject to $950 fee: 1% on the first $10,000 ($100 fee), 2% on the next $5,000 ($100 fee), and 3% on the last $25,000 ($750 tax).

Write a program that asks the user for the payment amount (real number) and outputs the fee owned (real number).

Example: java PayFriend 450.0

RESULT: 13.5

Two Smallest. Write your code in the file TwoSmallest.java, your file has to have this exact name with T and S capitalized. You must use the IO module to read inputs and to output your answers. Write a program that takes a set of numbers and determines which are the two smallest numbers. Ask the user for the following information, in this order: A terminating value (real number). The user will enter this value again to indicate that he or she is finished providing input. A sequence of real numbers. Keep asking for numbers until the terminating value is entered. Compute and output the smallest and second-smallest real number, in that order. It is possible for the smallest and second-smallest numbers to be the same (if the sequence contains duplicate numbers). There must be at least 2 (two) numbers in the list of numbers that is not the terminating value. If the user enters less than 2 (two) numbers, consider an error. Report the error input via IO.reportBadInput() and RE-ASK the user for the input until it is correctly entered.

Example: java TwoSmallest 123 [this is the terminating value, not part of the set of numbers] 17.0 23.5 10.0 15.2 30.0 8.0 16.0 123 [this is the terminating value again, indicating that the user is done]

RESULTS TO OUTPUT (in this order): 8.0 10.0

A.2 Complete statics for hints efficiency

In this paper, we discuss the most relevant results regarding the movement of student submissions throughout the different classes of errors in Section 5.4.1. For completion, in this section we include the percentages of all the possible pairs of submissions for each of the two assignments, and for each of the three semesters. In each table, the row label indicates the start error class and the column label indicates the end error class for a pair of code submissions. We calculate each percentage by counting the number of pairs and then diving it by the total number of pairs which started in the same error class.

Table 8. Movement between classes of error from one submission to another for PayFriend in spring 2016; CM - does not compile, IN - failed to follow instructions, DR - failed data representation, CF - failed control flow, FM - failed translating to formulas, CN - failed translating to conditional statements, NM - both CN and FM, NO - no concept failed/ passed all tests

	CM	**IN**	**DR**	**CF**	**CN**	**FM**	**NM**	**NO**
CM	**33.3%**	4.2%	8.3%	20.8%	8.3%	4.2%	4.2%	16.7%
IN	4.4%	**36.8%**	9.6%	10.5%	7.9%	2.6%	12.3%	**14.9%**
DR	0.8%	2.1%	**57.3%**	3.4%	2.9%	1.7%	9.2%	21.8%
CF	1.5%	3.1%	0.0%	**44.6%**	13.9%	4.6%	6.2%	24.6%
CN	0.0%	3.5%	6.9%	6.9%	**37.9%**	0.0%	6.9%	**34.5%**
FM	4.4%	0.0%	0.0%	2.2%	0.0%	**28.9%**	0.0%	2.2%
NM	0.9%	1.7%	2.7%	1.8%	5.3%	5.3%	**46.9%**	**35.4%**

Table 9. Movement between classes of error from one submission to another for PayFriend in spring 2017

	CM	**IN**	**DR**	**CF**	**CN**	**FM**	**NM**	**NO**
CM	**45.6%**	0.0%	5.1%	0.0%	10.1%	7.6%	13.9%	17.7%
IN	3.7%	**7.4%**	3.7%	3.7%	0.0%	33.3%	29.6%	**11.1%**
DR	3.8%	0.0%	**59.0%**	0.0%	10.3%	1.3%	3.8%	21.8%
CF	25.0%	0.0%	0.0%	**75.0%**	0.0%	0.0%	0.0%	0.0%
CN	9.1%	0.0%	0.0%	9.1%	**18.2%**	9.1%	18.2%	**18.2%**
FM	0.0%	0.0%	0.0%	0.0%	0.0%	**33.3%**	0.0%	66.7%
NM	1.7%	1.7%	0.0%	0.0%	3.4%	5.2%	**65.5%**	**22.4%**

Table 10. Movement between classes of error from one submission to another for PayFriend in spring 2018

	CM	**IN**	**DR**	**CF**	**CN**	**FM**	**NM**	**NO**
CM	**33.7%**	2.2%	4.3%	8.7%	7.6%	2.2%	18.5%	21.7%
IN	14.6%	**7.3%**	0.0%	7.3%	9.8%	2.4%	26.8%	**31.7%**
DR	5.3%	0.0%	**28.9%**	5.3%	18.4%	2.6%	15.8%	21.1%
CF	2.4%	0.0%	2.4%	**38.1%**	9.5%	4.8%	16.7%	26.2%
CN	1.7%	0.0%	1.7%	5.0%	**33.3%**	1.7%	5.0%	**51.7%**
FM	0.0%	0.0%	0.0%	0.0%	2.9%	**20.6%**	20.6%	55.9%
NM	0.0%	0.0%	0.0%	3.7%	12.1%	5.6%	**29.0%**	**49.5%**

Table 11. Movement between classes of error from one submission to another for TwoSmallest in spring 2016; CM - does not compile, IS - failed to follow instructions, IL - infinite loop, SQ - failed to read sequence, IN - failed to initialize min variables, UP - failed to update min variables, NO - no concept failed/ passed all tests

	CM	**IS**	**IL**	**IN**	**UP**	**SQ**	**SI**	**SU**	**NO**
CM	**0.0%**	33.3%	13.3%	6.7%	0.0%	6.7%	0.0%	6.7%	33.3%
IS	2.4%	**39.1%**	17.8%	1.6%	5.1%	7.5%	12.7%	1.6%	12.3%
IL	0.0%	11.3%	**58.6%**	2.7%	4.1%	2.7%	6.8%	1.8%	12.2%
IN	0.0%	5.3%	5.3%	**26.3%**	21.1%	10.5%	0.0%	0.0%	31.6%
UP	0.00%	7.3%	19.5%	0.0%	**29.3%**	0.0%	0.00%	0.00%	43.9%
SQ	1.3%	14.1%	18.0%	1.3%	0.0%	**21.8%**	6.4%	0.0%	**37.2%**
SI	0.6%	10.7%	14.1%	2.8%	6.2%	9.0%	**33.9%**	4.5%	18.1%
SU	4.2%	8.3%	12.5%	0.00%	16.7%	0.0%	20.8%	**16.7%**	20.8%

Table 12. Movement between classes of error from one submission to another for TwoSmallest in spring 2017

	CM	**IS**	**IL**	**IN**	**UP**	**SQ**	**SI**	**SU**	**NO**
CM	**20.70%**	0.00%	0.00%	3.40%	20.70%	13.80%	8.60%	1.70%	31.00%
IS	10.70%	**25.00%**	3.60%	0.00%	7.10%	28.60%	3.60%	3.60%	17.90%
IL	6.30%	0.00%	**25.00%**	6.30%	6.30%	31.30%	12.50%	0.00%	12.50%
IN	3.70%	0.00%	3.70%	**44.40%**	0.00%	3.70%	3.70%	3.70%	37.00%
UP	0.00%	0.00%	0.00%	8.70%	**39.10%**	4.30%	0.00%	0.00%	47.80%
SQ	6.10%	1.00%	3.00%	0.00%	4.00%	**57.60%**	14.10%	1.00%	**13.10%**
SI	2.80%	0.70%	0.70%	6.90%	5.50%	17.20%	**42.10%**	5.50%	18.60%
SU	2.70%	0.00%	0.00%	0.00%	16.20%	18.90%	10.80%	**40.50%**	10.80%

Table 13. Movement between classes of error from one submission to another for TwoSmallest in spring 2018

	CM	IS	IL	IN	UP	SQ	SI	SU	NO
CM	**28.60%**	3.90%	3.90%	9.10%	6.50%	14.30%	9.10%	3.90%	20.80%
IS	8.60%	**14.30%**	5.70%	17.10%	5.70%	20.00%	17.10%	0.00%	11.40%
IL	20.00%	8.00%	**20.00%**	4.00%	4.00%	24.00%	8.00%	4.00%	8.00%
IN	4.00%	4.00%	0.00%	**18.00%**	16.00%	8.00%	12.00%	2.00%	36.00%
UP	2.70%	1.40%	0.00%	2.70%	**41.10%**	1.40%	2.70%	2.70%	45.20%
SQ	4.00%	0.60%	0.00%	4.00%	6.30%	**52.60%**	7.40%	1.70%	**23.40%**
SI	6.20%	1.80%	0.90%	8.80%	12.40%	13.30%	**32.70%**	5.30%	18.60%
SU	2.90%	0.00%	2.90%	5.70%	40.00%	22.90%	0.00%	**20.00%**	5.70%