GroverCode: Code Canonicalization and Clustering Applied to Grading

by

Stacey Terman

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by .....
Robert C. Miller
Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

GroverCode: Code Canonicalization and Clustering Applied to Grading

by

Stacey Terman

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Teachers of MOOCs need to analyze large quantities of student submissions. There are a few systems designed to provide feedback at scale. Adapting these systems for residential courses would provide a substantial benefit for instructors, as a large residential course might still have several hundred students. OverCode, one such system, clusters and canonicalizes student submissions that have been marked correct by an autograder. We present GroverCode, an expanded version of OverCode that canonicalizes incorrect student submissions as well, and includes interface features for assigning grades to submissions. GroverCode was deployed in 6.0001, an introductory Python programming course, to assist teaching staff in grading exams. Overall reactions to the system were very positive.

Thesis Supervisor: Robert C. Miller
Title: Professor of Computer Science and Engineering

Acknowledgments

This thesis would not exist without all the support I've received from the people around me.

First, a huge thank-you to Elena Glassman. Elena, without you I would not have a project to work on in the first place. More important than your (not inconsiderable!) technical expertise, however, is your continuing guidance and encouragement. I've lost track of the number of times I've thought to myself, "I'm so glad Elena is here" over the course of this project. If your future advisees appreciate your mentorship as much as I do, you are going to make an *incredible* adviser. Best of luck!

Second, many, many thanks to my parents. Dad, thank you for your practical advice and ready supply of hugs, and for occasionally prying into my business enough to keep me on track. Mom, thank you for your emotional support and copy editing, and for occasionally nagging me until I actually got work done. Also for having had already will have helped me with my mixed-up tenses. And, of course, thank you both for raising me with the curiosity, enthusiasm, and determination necessary to not merely succeed, but to thrive here at MIT. Now I am all set for life!

Third, thanks to the staff of 6.0001, especially Ana, for letting me use them as guinea pigs! I really appreciate everyone's willingness to try out GroverCode and provide feedback. I hope the system continues to help out the course in the future.

Fourth, my gratitude to the members of Rob Miller's research group for providing a steady stream of tips and suggestions. Further thanks for all the daily standup meetings that started the mornings with laughter.

Finally, a last thank-you to everyone who's given me a hug in the past year, especially when I was stressed. <3

Contents

1 Introduction

Interactive online learning experiences such as Intelligent Tutoring Systems (ITS) and MOOCs are becoming increasingly popular and produce vast numbers of beginner-level programs. A MOOC can have hundreds or thousands of students, and the analyzing and grading of every submission individually would require an infeasible amount of instructor time and effort. Consequently, there are a few systems designed to help teachers analyze student submissions and provide feedback at scale. Instructors of a residential course, unlike MOOC instructors, are expected to provide feedback and assign grades to every student. Residential courses are generally much smaller than online courses. However, a large residential programming course can still be as large as several hundred students. Residential teaching staff invest a great deal of time into grading and providing feedback to students. Adapting tools designed for large-scale online classes for residential courses would provide a substantial benefit for these instructors.

This thesis focuses specifically on grading exams in 6.0001, an introductory Python programming course. In this course, a portion of each exam involves submitting code online via the MITx platform. In the past, the MITx platform assigned a grade automatically based on a suite of unit tests. However, basing scores solely on unit test results can unfairly penalize some students while rewarding others. For example, a student might write a function that performs the correct logic, but makes a small error such as returning a floating point number instead of the required integer. This student receives no credit from an autograder, despite showing an understanding of programming concepts. Conversely, a different student might write a function that always returns the same answer, and receive partial credit, despite showing no such understanding. For this reason, the teaching staff of 6.0001 now grade by looking at each student's code and adjusting the output of the autograder as necessary. This grading process is long and exhausting, often taking upwards of five hours per exam.

To ease this process, we present GroverCode, an expanded version of the OverCode system [3]. As originally designed, OverCode allows teachers to visualize clusters of

correct solutions that behave equivalently. It also renames variables to increase consistency between solutions. With this system, a teacher can get a sense of the space of student programming solutions and tailor feedback to reach as many students as possible. Originally evaluated on MOOC data, this tool has great potential for residential environments. The enhancements to GroverCode include handling incorrect submissions as well as correct ones, and applying the system to the task of grading. When grading, the staff must first understand what a student's code does and locate any mistakes in order to deduct points or provide feedback. Increasing the similarity between incorrect solutions, to simplify the understanding step, was a primary goal of this system.

In the GroverCode pipeline, we process incorrect solutions, that is, solutions which have been marked as incorrect by an autograder for failing one or more test cases. The original OverCode pipeline clustered correct solutions based on variable behavior and renamed common variables so that all correct solutions shared a namespace. The modified pipeline also analyzes the behavior of variables in incorrect solutions by extracting the sequence of values these variables take on and the structure of the lines of code in which they appear. Using this information, GroverCode can rename these variables as well. The system then displays all solutions within a consistent namespace, regardless of correctness. The GroverCode user interface includes features for assigning scores and comments to solutions. This combines viewing code and assigning grades into one interface.

We deployed the GroverCode system as a grading tool in the Spring 2016 semester of 6.0001. Approximately 200 students enrolled in this course. Nine instructors, consisting of a professor and eight Teaching Assistants (TAs), including the author, used GroverCode to help grade both the midterm and the final exam, which together contained seven programming problems. They used GroverCode to grade most, though not all, student submissions. The number of GroverCode-graded submissions per problem ranged from 133 to 189. Overall reactions to the system were very positive, especially when applied to simple problems.

The main contributions of this thesis are:

- An updated pipeline for the GroverCode system that analyzes the behavior of variables in correct and incorrect solutions and renames them into a common namespace
- An interface for displaying and grading canonicalized code
- Two field deployments in which instructors of an introductory Python course used GroverCode to grade student exams and provided positive feedback about the system

2 Related Work

2.1 OverCode

The original OverCode [3] helps teachers and other graders explore variation in student submissions in large-scale programming classes such as MOOCs. It analyzes only submissions marked as correct by an autograder, and clusters hundreds or thousands of submissions by analyzing variable behavior on a single test case. It also canonicalizes student submissions by renaming variables to increase the similarity between submissions and to improve human readability. Glassman et al. tested OverCode with a group of Teaching Assistant (TA) graders and found that the interface helped them quickly assess students' understanding. Modifications and expansions to GroverCode described in this paper support the specific task of exam grading in a residential introductory Python course, 6.0001. GroverCode adds the ability to run student submissions on multiple test cases, rather than limiting to a single test case. It augments the pipeline to handle submissions that are marked as incorrect by an autograder. GroverCode does not cluster incorrect submissions; however, it renames the variables in incorrect submissions to share names with the variables in correct submissions. Finally, it adds interface support for assigning grades and adding comments to submissions.

2.2 Cluster Assisted Grading

Basu et. al [1] use clustering to explore the idea of "grading on a budget," i.e., maximizing the impact of a small number of human actions when grading short answer questions. They train a classifier to label pairs of submissions as similar or different, then break the set of submissions into a fixed number of clusters and subclusters, automatically marking clusters and subclusters as correct by clustering answer key items with student submissions. Teachers can flip the labels of clusters, subclusters, and individual submissions as appropriate. This approach of assigning grades to whole groups of submissions at one time amplifies the teacher's effort. It can also reveal common modes of misunderstanding, because submissions within incorrect subclusters often contain similar mistakes.

GroverCode uses a suite of grader-supplied unit tests to initially mark submissions as correct or incorrect, rather than training a classifier. It clusters submissions marked as correct into stacks, based on variable behavior and syntax. It calculates a metric of similarity between submissions, but does not use that metric to cluster, only to change the order in which it displays submissions. It groups submissions according to the number of test cases they pass. Often these groups display similar mistakes, as with the subclusters in the system described by Basu et al.

Brooks et al. [2] build on the work of Basu et al. presenting a web interface for displaying clusters and subclusters, and assigning grades and feedback to them. They evaluate this interface with a group of expert graders, who report that the clustered interface is faster, easier to use, and more enjoyable than a flat interface for grading the same problems.

The GroverCode interface focuses on displaying student code in a human-friendly fashion, while Brooks et al. focus on displaying clusters and subclusters in a meaningful way. GroverCode only clusters submissions that behave identically, so there is no need to visualize variation within a cluster. Both interfaces aim to facilitate hand-grading of student answers.

Gross et al. [4] use prototype-based clustering and examine the effectiveness of comparing student submissions to the prototype submissions. They use the Relational Neural Gas technique (RNG) to cluster graded submissions and find prototypes. They provide feedback on each new submission by highlighting the differences between that submission and the closest prototype submission. Expert graders determined that seeing these differences can help students debug their code.

Unlike GroverCode, Gross et al. focus on providing feedback to a single student at a time. They cluster submissions already graded by experts, and use these clusters and their prototypes to help identify problems in a new submission. In contrast, GroverCode processes ungraded submissions and helps staff assign grades to a whole body of submissions at one time. However, GroverCode does highlight differences between submissions to help pinpoint problems, although it is staff, rather than students, who view these differences.

2.3 Hint generation

Singh et al. [7] use program synthesis techniques to attempt to automatically correct student code. Given a specification and an error model, their system can correct a large fraction of incorrect submissions. They use the Sketch synthesizer to apply the changes suggested by the error model to a student submission and generate a set of candidate programs. Of all these candidate programs, they consider those that match a provided reference implementation and pick the one that requires the minimum number of corrections. Singh et al. find that a single addition to the error model could enable correcting hundreds of additional submissions, indicating that students often make overlapping mistakes. One benchmark error model corrected 65% of student submissions to a particular problem. However, this system requires teachers to define an error model specific to each problem, and only supports a subset of Python.

GroverCode does not attempt to fix students' bugs automatically, but instead aims to support teaching staff as they grade code manually. It considers the fact

that students make overlapping errors, by grouping submissions based on test case output, since submissions which pass the same set of test cases often include the same mistake. In addition, the task of generating a rubric to grade a particular problem is similar to that of generating an error model.

Rivers & Koedinger [6] generate hints by locating incorrect student submissions in a space of possible submissions to a given problem. They generate this submission space from previous student submissions by extracting the Abstract Syntax Tree (AST) for each submission, and normalizing it using semantics-preserving operations, including anonymizing variable names. Rivers & Koedinger generate feedback for an incorrect student submission by finding the closest correct submission using string edit distance, and then determining the necessary changes to convert the student's submission into a correct submission.

Like GroverCode, Rivers & Koedinger rename variables to increase the similarity of submissions. However, GroverCode also attempts to increase the human readability of submissions by renaming variables based on common names chosen by students. Rather than automatically generating hints based on similar submissions to a student submission, GroverCode orders submissions based on similarity and displays them to teaching staff to assist them in assigning grades.

3 Dictionary of Terms

We define several terms below, used throughout Sections 4 and 5.

Test Case A single Python function call designed to test some aspect of a student's code. This may be a call to a student-defined function, or a call to a grader-defined function that calls a student's function and performs computation on the results.

Error Signature A particular pattern of passes and failures on a set of test cases.

Submission A single student's answer to a particular coding problem.

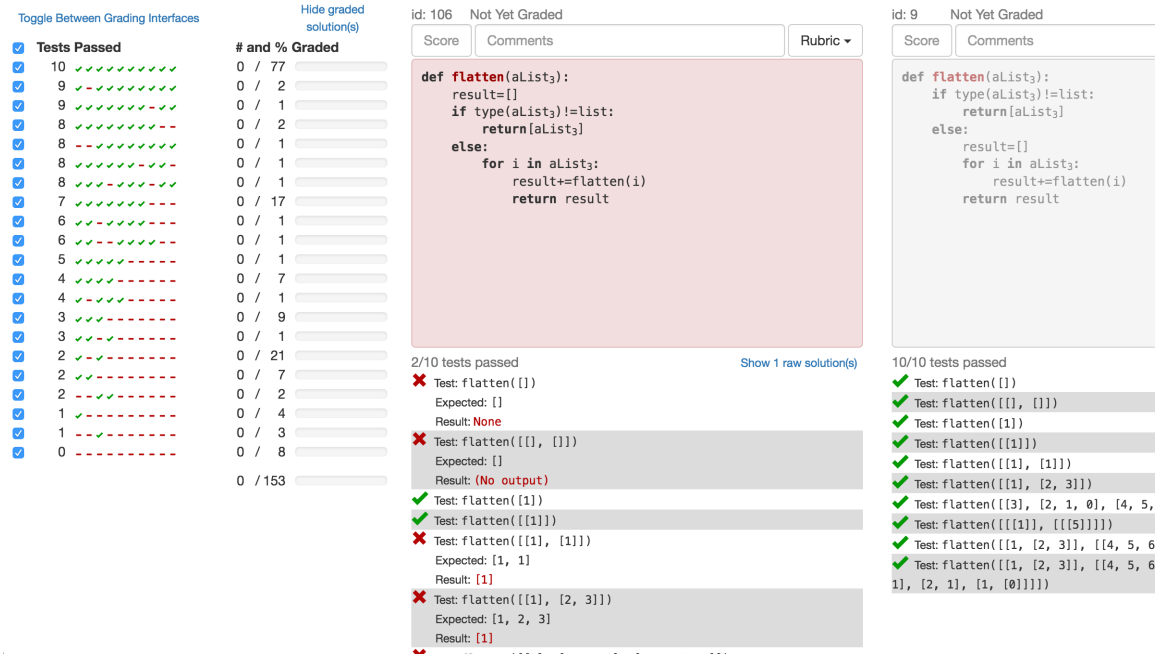


Figure 1: The GroverCode user interface, showing data for a problem from a previous semester of an introductory Python programming course. The problem asks students to flatten a nested list of arbitrary depth.

Correct Submission A submission that has been marked correct by an autograder, i.e., a submission that passes every test case in a set of test cases.

Incorrect Submission A submission that has been marked incorrect by an autograder, i.e., a submission that fails at least one test case in a set of test cases.

Variable Instance A single global or local variable within a submission.

Abstract Variable A collection of one or more equivalent variable instances. Variable instances are considered equivalent if they take on an identical sequence of values.

Cleaning Removing comments from a submission and reformatting it to ensure consistent line indentation and spacing.

Canonicalizing Renaming the variable instances within a submission such that the submission shares a namespace with all other submissions.

Stack A collection of one or more cleaned and canonicalized submissions. A *correct stack* contains one or more equivalent correct submissions. See Section 5 for an explanation of which submissions are considered equivalent. An *incorrect stack* contains a single incorrect submission.

4 Design

The GroverCode user interface is the result of several iterations of prototypes. At each stage in the design process, members of our research group and members of the teaching staff of 6.0001, the introductory Python course, tested the interface during intermediate design stages. The final design, shown in Figure 1, incorporates their feedback. The GroverCode user interface includes two sections: a filter panel on the left and a stack display on the right. The filter panel contains one row for each observed error signature, which is a particular pattern of test case passes and failures. Each row lists the number of passed test cases in the error signature, a visual representation of the error signature, the total number of stacks exhibiting that error signature, the number and percentage of graded stacks exhibiting that error signature, and a progress bar for that error signature. An additional row at the bottom displays the total number of stacks, the total number of graded stacks, and an overall progress bar.

The stack display contains one column per stack, arranged horizontally. The horizontal alignment facilitates comparing code and test case results between stacks. Each column features a large center panel containing the code of the stack. A red panel means that the stack failed one or more test cases. Above the code panel is the interface for entering grades, with two textboxes for entering a score and a comment, respectively, and a dropdown menu to show the rubric for the current problem. There is also a numeric identifier for the stack, and an indication of whether the stack has been graded. Test case information appears below the code panel. This information includes a description of each test case and a green checkmark or red X to denote a passed or failed test case, respectively. Failed test cases also include

the expected output and the actual output produced by the stack. Finally, a link beneath the code panel on the right toggles the display of *raw* submissions, that is, the uncleaned and uncanonicalized code of each submission associated with the stack. Raw submissions include some small amount of metadata, but are otherwise unmodified from the student's original submission (See Figure 2).

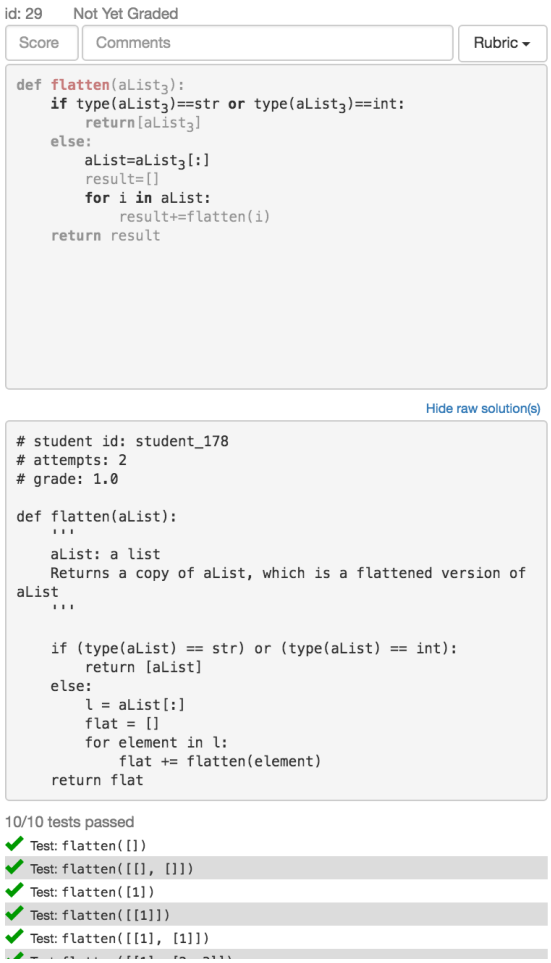


Figure 2: A correct stack with the corresponding raw submission displayed. The top panel, with syntax highlighting, is the code panel. The bottom panel is the raw submission. The dimmed lines of code in the code panel are shared between this stack and the previous stack, not shown here. The three comments at the top of the raw submission contain metadata added by the script that parses the CSV file of all student submissions. The rest of the raw submission is exactly as submitted by the student. The third line of metadata, grade, is the score that the student sees after clicking "Submit" on the MITx platform while taking the exam. In the residential course, the score is always 1.0, regardless of the correctness of a student's code.

4.1 Stacks

A stack contains at least one cleaned and canonicalized student submission. An incorrect stack, indicated by code with a red background, always represents a single submission. A correct stack may represent multiple similar submissions grouped together. The process of cleaning, canonicalizing, and grouping submissions is described in Section 5. The number of submissions a stack contains appears beneath the code panel, in the link to toggle the display of raw submissions. If a stack contains multiple submissions, clicking this link causes all the associated raw submissions to appear, listed vertically. Grading a stack assigns the same score and comment to every submission contained in the stack.

4.2 Filtering

With the left-hand panel, graders can filter the displayed stacks based on error signature by selecting the checkboxes next to the desired rows. The additional checkbox next to the headings is a "select all" checkbox: selecting or clearing this checkbox automatically selects or clears the checkboxes for all error signatures. Another option allows toggling the display of previously graded stacks. The displayed stacks appear in an order that maximizes similarity between adjacent stacks, as described below.

4.3 Ordering

The display order of the stacks depends on a pairwise metric of similarity, described in Section 5. GroverCode finds the pair of stacks with the largest similarity and displays these as the first two stacks, arbitrarily selecting one of these as the first stack. It selects the third and all subsequent stacks by finding the remaining stack that is most similar to the previously chosen stack. Ordering stacks so that neighboring stacks are similar minimizes the cognitive load of moving between stacks.

4.4 Highlighting Differences Between Stacks

Similar to [3], GroverCode uses dimming to highlight differences between stacks. It compares each stack to its previous neighbor, unlike [3], which compares every stack to a single reference stack. Lines of code shared with the previous stack become dim, making differences between neighboring stacks more apparent (See Figure 2).

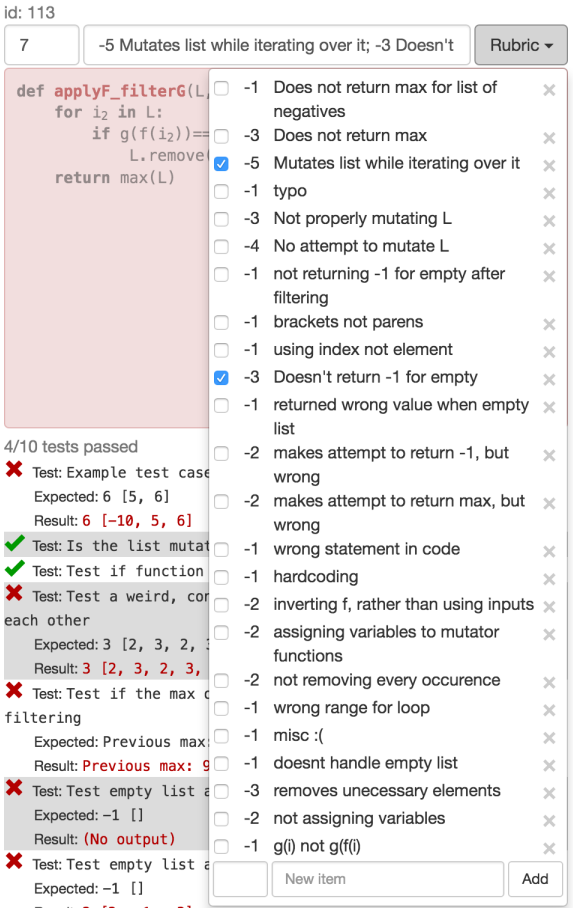


Figure 3: A stack with the rubric dropdown menu open. This rubric comes from the second field deployment of GroverCode (See Section 8.3). Text for each of the two checked items appears in the comment box.

4.5 Grading

Each column in the stack display has space to enter a score and a comment for the stack. After a grader enters a score, the status of the stack changes to graded. GroverCode removes the label "Not Yet Graded" and updates the appropriate progress

bar in the filter pane. A grader can also enter a comment explaining the score. In addition to or instead of typing a comment manually, a grader can select items from the rubric dropdown menu (see Figure 3) to apply to the stack. A rubric item consists of a score delta and a description, e.g., "-1, does not handle empty lists." In the dropdown menu, a checkbox appears next to each item. Selecting a checkbox adds text to the comment textbox.

A grader can also add new items to the rubric via a set of textboxes at the bottom of the dropdown menu. After entering a score delta and a description, clicking "Add" or pressing Enter adds the new item. New rubric items are immediately visible to all staff grading a particular problem. To delete a rubric item, a grader can click the X next to the item in the dropdown. This does not change the comments on previously graded stacks.

5 Implementation

GroverCode processes student submissions with a multi-stage pipeline. The system first cleans each submission, and executes it on a series of grader-supplied test cases using the Python execution logger described in [5]. GroverCode then canonicalizes these submissions using the results of the execution, and groups submissions into stacks. Each step of the pipeline is described in detail below.

1. Preprocess submissions. In the first stage of the pipeline, GroverCode cleans and executes student submissions as described below. Executing student submissions requires one or more test cases. Test cases must be supplied by the human grader in the form of a file with one test case per line. A test case consists most often of a single call to a student-defined function. However, it is not necessary to limit test cases to this format. To introduce more complicated test cases, a grader can define a set of test functions in a separate file.

a. Clean submissions. As in the original OverCode [3], GroverCode reformats submissions to ensure consistent line indentation and token spacing, and removes

comments. Although students often use comments to clarify their intent, in 6.0001, graders ignore such additional notes. Analyzing comments remains future work.

b. Augment submissions. GroverCode automatically appends any test case definitions that appear in a separate file to each student's submission. The system also appends any additional code that a student's submission relies on, such as the definition of a Python class of which the student created a subclass.

c. Execute submissions. GroverCode runs each submission through the Python execution logger described in [5], one time per test case. The names and values of each variable instance at every step in the execution are recorded in a data structure called a program trace, as in [3]. The execution logger generates one program trace for each test case. It also intercepts the submission's `stdout` and records the submission's output instead of printing it. Next, GroverCode serializes each program trace and intercepted output and writes it out to a pickle file. Because of this serialization step, analyzing additional submissions does not require rerunning any previously analyzed submission. This is particularly useful when applied to exam grading, where a small number of students could take a makeup exam at a later date.

2. Extract variable sequences. As in [3], for every submission, GroverCode extracts the sequence of values each variable instance takes on. A separate sequence of values is extracted for each test case. Additionally, in this step, GroverCode separates correct and incorrect submissions by comparing the intercepted output from each test case to the output of a supplied correct answer.

3. Identify abstract variables in correct submissions. GroverCode next analyzes the extracted sequences of values for each variable instance in every correct submission, and identifies all distinct abstract variables. Incorrect submissions are not considered here, because the values of variable instances in an incorrect submission are unreliable. For example, a submission may raise an exception after initializing

a local variable. The sequence of values of the variable instance in this case would contain just a single value.

4. Rename abstract variables in correct submissions. GroverCode assigns each abstract variable the name that it takes on most often across all program traces. If there is more than one abstract variable that takes on the same name, GroverCode adds a modifier to the variable that appears less frequently. This is what [3] calls a Common/Common collision. Instead of using letters as modifiers as in [3], GroverCode appends three underscores and a number; for example, changing the name of the second-most-common variable called `i` to `i___2` instead of `iB`. In the UI, these modifiers appear as subscripts, so `i___2` appears as $\texttt{i}_2$. Unlike in [3], abstract variables that appear in only a single submission are not treated as a special case.

5. Analyze behavior of individual lines. Next, GroverCode considers the behavior of both correct and incorrect submissions at the level of lines rather than variables. By analyzing the program trace and the AST, each line of code is split into three components:

1. A *template*, a string consisting of the text of the line of code with the variable names replaced with blanks.

2. An ordered list of variables, one per blank in the template. These may be either abstract variables or variable instances. The first variable belongs in the first blank in the template, the second variable in the second blank, etc.

3. An ordered list of the sequences of values that appear in each blank.

6. Stack correct submissions. GroverCode gathers together correct submissions into stacks as in [3], placing submissions into the same stack if they share a set of lines of code. Rather than comparing the literal string representation of each line, however, the system compares the line templates and the sequences of values each blank in the template takes on. GroverCode chooses one submission arbitrarily to represent the stack.

7. Rename variable instances in incorrect submissions. The next step canonicalizes incorrect submissions. GroverCode attempts to rename the variable instances in incorrect submissions so that they share the same namespace as correct submissions. In step 3, when the system identifies abstract variables, it is clear that all submissions in which those variables appear are functionally correct. However, when choosing names for incorrect variable instances, that assumption no longer holds, so name choice cannot depend solely on the values a variable instance takes on. There are several steps of the renaming process.

a. Characterize behavior of variable instances and abstract variables. Step 5 identifies where variables appear in each line of code in each submission. For each abstract variable identified in step 3, GroverCode assembles a set of *template-location pairs*, that is, the set of templates in which the abstract variable appears and the location of that abstract variable in each template. The location is represented as the index or indices of the blank(s) occupied by that variable instance, as illustrated in the example below. The system repeats this process for each variable instance in each incorrect submission as well. This information about variable behavior is important for determining which abstract variable is the best match for each incorrect variable instance.

b. Assign scores to template-location pairs. GroverCode assigns to each template-location pair a score inversely proportional to the frequency with which it appears among abstract variables in correct submissions. This score calculation is $\log_2(1/p)$, where p is the probability of a given template-location pair, i.e., the number of times that particular template-location pair appears across all abstract variables in correct submissions, divided by the total number of template-location pairs. A threshold calculation separates template-location pairs that appear in only one abstract variable from template-location pairs that appear in multiple abstract variables.

Example: All templates and locations in which the abstract variable `exp`, the second argument to a recursive `power` function, appears. A location represents the index or indices of the blanks that the abstract variable occupies, where the first blank is index 0, the second is index 1, and so on. The second and third columns together form a template-location pair.

Example line of code	Template	Location
`def power(base, exp):`	`def power(__,__):`	1
`while index <= exp:`	`while __<=__:`	1
`return 1.0*base*power(base, exp-1)`	`return 1.0*__*__*power(__,__-1)`	3
`return base*power(base, exp-1)`	`return __*power(__,__-1)`	2
`return power(base, exp-1)*base`	`return power(__,__-1)*__`	1
`ans = base*power(base, exp-1)`	`__=__*power(__,__-1)`	3
`if exp <= 0:`	`if __<=0:`	0
`if exp == 0:`	`if __==0:`	0
`if exp >= 1:`	`if __ >= 1:`	0
`assert type(exp) is int and exp >= 0`	`assert type(__) is int and __>=0`	0, 1

c. Match incorrect variable instances to abstract variables. GroverCode then attempts to match each variable instance in an incorrect submission with an abstract variable. For each such variable instance, GroverCode searches for an abstract variable that fulfills one of two criteria. The system begins by looking for an abstract variable that fulfills the first criterion, and only moves on to the second if no such abstract variable is found. The criteria are as follows:

1. The abstract variable takes on an identical sequence of values to the variable instance in question. It is possible for parts of a program to be correct even if the overall output is incorrect, so it is important to consider this criterion first.

2. The abstract variable appears in an identical set of template-location pairs to the variable instance in question.

If such an abstract variable is found, GroverCode assigns the name of that abstract variable to the variable instance in question. Otherwise, GroverCode uses the scores

calculated in the previous step to estimate which abstract variable is the closest match for that variable instance. The *overall score* of the match between an abstract variable and a variable instance is defined as the sum of the individual scores of each template-location pair that is shared between the abstract variable and the variable instance. The closest match for the variable instance in question is the match that yields the highest overall score, with ties resolved arbitrarily. Another method of resolving ties, such as minimizing the string edit distance between the original name and the possible matches, could yield better results. This remains future work. If the overall score of the closest match is below the threshold found in the previous step, then it is not possible to unambiguously map the variable instance to a single abstract variable (see example below); in that case, GroverCode does not rename the variable instance, instead using the student's original name. Otherwise, GroverCode assigns the name of the abstract variable in the closest match.

In all cases, if multiple variable instances in a single submission would be assigned the same name, the system appends a capital letter modifier to distinguish them.

Example: This is the recursive `power` problem. One submission includes a variable instance `ans`, shown in the table below. The first template-location pair does not appear in any abstract variables, so contributes nothing to the overall score. The second template-location pair, however, appears in two different abstract variables, so there is not enough information to determine which abstract variable name more closely matches the student's intent. Do not rename this variable, but use the student's original variable name.

Line of code	Template	Location
ans = base**exp	__=__**__	0
return ans	return __	0

8. Find pairwise similarity between stacks. *Relevant submissions* include the subset of correct submissions that are representatives of the stacks found in step 6, and every incorrect submission. These are the submissions that the view renders. For each relevant submission, GroverCode calculates a metric of similarity with each

other relevant submission, as follows:

$$\frac{\#\text{ of shared phrases}}{\text{total number of phrases}} + \frac{\#\text{ of shared variable names}}{\text{total number of names}}$$

Here, "shared phrases" and "shared variable names" mean phrases or variable names contained in both submissions, and "total" means the total across both submissions. A phrase is a line of code with all variables renamed as described above in step 4 or 7 as appropriate. Two identical submissions have a similarity metric of 1.0, while two submissions which share no variables or lines of code have a metric of 0.

6 Dataset

We evaluated the pipeline using data from the two field deployments (see Section 8). We obtained data from the spring 2016 semester of 6.0001, a residential introductory Python course, from both the midterm and the final exam for the course. Students submitted answers to three coding problems on the midterm and four on the final, described below. The number of submissions submitted for each problem appears in Figure 4. An example staff-written correct solution for each problem appears after its description.

6.1 Midterm Problems

- Question 4: `power`. Write a recursive function to calculate the exponential `base` to the power `exp`.

```
def power(base, exp):
    """
    base: int or float.
    exp: int >= 0
    returns: int or float, base^exp
    """
    if exp <= 0:
        return 1
    return base * power(base, exp - 1)
```

- Question 5: `give_and_take`. Given a dictionary `d` and a list `L`, return a new dictionary that contains the keys of `d`. Map each key to its value in `d` plus one if the key is contained in `L`, and its value in d minus one if the key is not contained in L.

```
def give_and_take(a_dict, L):
    '''
    L: list of integers
    d: dictionary mapping int:int
    Returns a new dictionary. *** The original d should not be mutated. ***
    The keys in the new dictionary are the keys that are in d.
    The value associated with each key in the new dicitonary is:
        one more that the value associated with that key in d if the key
            occurs in L
        one less than the value associated with the key in d if the key
            does not occur in L
    '''
    new_dict = {}
    for key in a_dict:
        if key in L:
            new_dict[key] = a_dict[key] + 1
        else:
            new_dict[key] = a_dict[key] - 1
    return new_dict
```

- Question 6: `closest_power`. Given an integer base and a target integer `num`, find the integer exponent that minimizes the difference between `num` and `base` to the power of exponent, choosing the smaller exponent in the case of a tie.

```
def closest_power(base, n):
    '''
    base: base of the exponential, integer > 1
    n: number you want to be closest to, integer > 0
    Find the integer exponent such that base**exponent is closest to n.
    Note that the base**exponent may be either greater or smaller than n.
    In case of a tie, return the smaller value.
    Returns the exponent.
    '''
    exp = 0

    while True:
        if base**exp >= n:
            break
        exp += 1

    if abs(n - base**exp) >= abs(n - base**(exp - 1)):
         return exp - 1
    elif abs(n - base**exp) < abs(n - base**(exp - 1)):
        return exp
```

6.2 Final Exam Problems

- Question 4: `deep_reverse`. Write a function that takes a list of lists of integers `L`, and reverses `L` and each element of `L` in place.

```
def deep_reverse(L):
    """ assumes L is a list of lists whose elements are ints
    Mutates L such that it reverses its elements and also
    reverses the order of the int elements in every element of L.
    It does not return anything.
    """
    L.reverse()
    for subL in L:
        subL.reverse()
```

- Question 5: `applyF_filterG`. Write a function that takes three arguments: a list of integers `L`, a function `f` that takes an integer and returns an integer, and a function `g` that takes an integer and returns a boolean. Remove elements from `L` such that for each remaining element `i`, `f(g(i))` returns `True`. Return the largest element of the mutated list, or -1 if the list is empty after mutation.

```
def applyF_filterG(L, f, g):
    """
    Assumes L is a list of integers
    Assume functions f and g are defined for you.
    f takes in an integer, applies a function, returns another integer
    g takes in an integer, applies a Boolean function,
        returns either True or False
    Mutates L such that, for each element i originally in L, L contains
        i if g(f(i)) returns True, and no other elements
    Returns the largest element in the mutated L or -1 if the list is empty
    """
    to_remove = []
    for s in L:
        if not g(f(s)):
            to_remove.append(s)

    for s in to_remove:
        L.remove(s)
    return max(L) if L != [] else -1
```

- Question 6: `MITCampus`. Given the definitions of two classes: `Location`, which represents a two-dimensional coordinate point, and `Campus`, which represents a college campus centered at a particular `Location`, fill in several methods in the `MITCampus` class, a subclass of `Campus` that represents a college campus with tents at various `Locations`.

```
class MITCampus(Campus):
    """ A MITCampus is a Campus that contains tents """
    def __init__(self, center_loc, tent_loc = Location(0,0)):
        """ Assumes center_loc and tent_loc are Location objects
        Initializes a new Campus centered at location center_loc
        with a tent at location tent_loc """
        Campus.__init__(self, center_loc)
        self.tents = [tent_loc]

    def add_tent(self, new_tent_loc):
        """ Assumes new_tent_loc is a Location
        Adds new_tent_loc to the campus only if the tent is at least 0.5
        distance  away from all other tents already there. Campus is
        unchanged otherwise. Returns True if it could add the tent, False
        otherwise. """
        for t in self.tents:
            if t.dist_from(new_tent_loc) < 0.5:
                return False
        self.tents.append(new_tent_loc)
        return True

    def remove_tent(self, tent_loc):
        """ Assumes tent_loc is a Location
        Removes tent_loc from the campus.
        Raises a ValueError if there is not a tent at tent_loc.
        Does not return anything """
        try:
            self.tents.remove(tent_loc)
        except:
            raise ValueError

    def get_tents(self):
        """ Returns a list of all tents on the campus. The list should contain
        the string representation of the Location of a tent. The list should
        be sorted by the x coordinate of the location. """
        res = []
        for t in self.tents:
            res.append((t.getX(),t.getY()))
        res.sort()
        ans = []
        for t in res:
            ans.append(str(Location(t[0], t[1])))
        return ans
```

- Question 7: `longest_run`. Write a function that takes a list of integers `L`, finds the longest run of either monotonically increasing or monotonically decreasing integers in `L`, and returns the sum of this run.

```
def longest_run(L):
    """
    Assumes L is a list of integers containing at least 2 elements.
    Finds the longest run of numbers in L, where the longest run can
    either be monotonically increasing or monotonically decreasing.
    In case of a tie for the longest run, choose the longest run
    that occurs first.
    Does not modify the list.
    Returns the sum of the longest run.
    """
    def get_sublists(L, n):
        result = []
        for i in range(len(L)-n+1):
            result.append(L[i:i+n])
        return result

    for i in range(len(L), 0, -1):
        possibles = get_sublists(L, i)
        for p in possibles:
            if p == sorted(p) or p == sorted(p, reverse=True):
                return sum(p)
```

Problem	Number of submissions
power	193
give_and_take	193
closest_power	193
deep_reverse	175
applyF_filterG	173
MITCampus	170
longest_run	165

Figure 4: Number of submissions for each problem in the dataset. Several students dropped the course between the midterm and the final exam. The decreasing number of submissions for each final exam question occurred because some students ran out of time before they could submit answers to the later problems.

7 Pipeline Evaluation

The GroverCode pipeline cannot process every student submission. The execution logger cannot handle submissions that contain syntax errors or that use too much memory, for example by calling `range` with a very large value. As shown below, the

pipeline does process most student submissions, although longer submissions are less likely to succeed.

	Quiz			Final			
	q4	q5	q6	q4	q5	q6	q7
Number of submissions	193	193	193	175	173	170	165
Avg. number of lines per submission	9.9	16.9	19.8	12.3	20.9	50.0	41.8
Number of submissions successfully processed by GroverCode pipeline	186 (96%)	189 (98%)	168 (87%)	170 (97%)	166 (96%)	134 (79%)	133 (81%)

Exam questions increased in difficulty between the midterm and the final and between the beginning and end of each exam. With an increase in difficulty, the number of correct submissions that can be grouped together in stacks decreases. The pipeline's ability to analyze incorrect submissions thus becomes more important for harder problems. The table below shows the number and percentage of submissions marked correct by the autograder, that is, submissions that pass every test case. It also shows the number of test cases, and the number of distinct error signatures, where an error signature is the number and order of test cases passed for a particular submission. The last row of the table displays the number of submissions that are grouped into stacks containing more than one submission.

	Quiz			Final			
	q4	q5	q6	q4	q5	q6	q7
Number of correct submissions	182 (94%)	160 (82%)	94 (49%)	96 (55%)	49 (28%)	16 (9%)	12 (7%)
Number of incorrect submissions	4	29	74	74	117	118	121
Number of test cases	10	15	25	11	10	17	28
Number of distinct error signatures	6	16	36	12	38	57	42
Number of correct stacks	40	84	93	47	46	16	12
Number of stacks containing > 1 submission	13	18	1	8	2	0	0
Number of submissions collapsed into stacks	151	94	2	57	5	0	0

GroverCode identifies abstract variables in correct submissions by comparing the

sequences of values they take on, as described in Section 5. The number of variable instances across all submissions and the number of abstract variables across correct submissions appear below.

	Quiz			Final			
	q4	q5	q6	q4	q5	q6	q7
Total number of variable instances	388	857	942	555	822	674	964
Number of variable instances in correct solutions	373	708	460	266	272	115	105
Number abstract variables	21	36	131	78	65	38	75
Number of name clashes resolved when renaming abstract variables	11	9	44	41	33	11	14

GroverCode manages to rename many of the variable instances in incorrect solutions. See Section 5 for more detail about this renaming process. The table below shows the number of variables renamed in this fashion as well.

	Quiz			Final			
	q4	q5	q6	q4	q5	q6	q7
Number of variable instances in incorrect submissions	15	149	482	289	550	559	859
Number of variable instances renamed based on values	14	84	266	97	246	97	187
Number of variable instances renamed based on templates	0	58	166	136	264	188	489
Number of variable instances not renamed	1	7	50	56	40	274	183
Number of name clashes resolved when renaming variable instances	0	13	38	14	16	10	156

7.1 Limitations

The GroverCode pipeline handles simple Python programs. Stacking correct submissions becomes much less effective on more complicated problems, as shown by the small number of stacked submissions in the later problems. Python classes pose additional problems. The execution logger [5] does not include information about instance variables in the generated trace, which makes renaming these variables impossible within the current pipeline architecture. Although Question 6 on the final exam involves classes, exploring how non-primitive variable values affect the renaming process remains future work.

8 Field Deployments

The teaching staff of 6.0001, an introductory residential Python programming class, used GroverCode as a tool to help with grading the course's two exams: midterm and final. Approximately 100-200 students take this course each semester. Each exam has two components: a paper portion with multiple choice and short answer questions, and an online portion where students submit code via the MITx platform. GroverCode deals only with the online portion. The online portion of the midterm has three problems, and the final exam has four (see Section 6). After each exam, an instructor extracts all student submissions in CSV format. Course staff grade student submissions by hand, assisted by a set of unit tests for each problem. The 6.0001 members of the teaching staff who participated in the field deployments included one professor, seven graduate student Teaching Assistants (TAs) including the author, and one undergraduate TA. The goal was to reduce the burden of hand-grading student code submissions.

8.1 Usual Grading Method

Course staff gather together in a conference room several days after an exam to assign grades. One staff member grades the paper component, while the remaining

staff grade the online component. The staff grading the online portion split into small groups of two or three, and each group chooses a problem to grade. Staff use a Google spreadsheet to record grades. They assign grades by examining the the CSV of all student submissions in a text editor such as Sublime Text, and by copying and pasting individual students' submissions into MITx to view unit test output. A rubric for each problem, recorded either at the top of the spreadsheet of grades or in a separate document, builds up incrementally. Assigning grades to all of the submissions is a long and exhausting process, often taking from five to eight hours.

8.2 First Field Deployment: Midterm exam

193 students submitted code for the online portion of the midterm, which consisted of three problems.

Preparation. The day after the exam, an instructor exported student submissions in CSV format from MITx. In preparation for the field deployment, we parsed the CSV and ran the resulting submissions through the GroverCode pipeline, and created three instances of GroverCode, one for each question. This allowed setting of question-specific parameters, such as the location from which to read submission data.

Interface. This field deployment used a prototype interface which more closely resembles the original OverCode interface described in [3] (see Figure 5). In this interface, stacks display in two columns, with correct stacks on the right and incorrect stacks on the left. Users can designate a single stack as the pinned stack. All other stacks display in order of decreasing similarity with the pinned stack. For each other stack, lines of code that share stacks with the pinned stack become dim to highlight differences between them. It is possible to change the pinned stack any time.

Training. Course staff received an overview of the GroverCode user interface the week before the field deployment, including the concepts of stacks, raw submissions, and pinning, followed by a brief refresher on the morning of the field deployment.

Process. At the start of the grading process, staff divided into groups of two or three, and each group claimed a problem to grade (see Section 6 for problem descriptions). All groups assigned to a particular problem began by looking at submissions

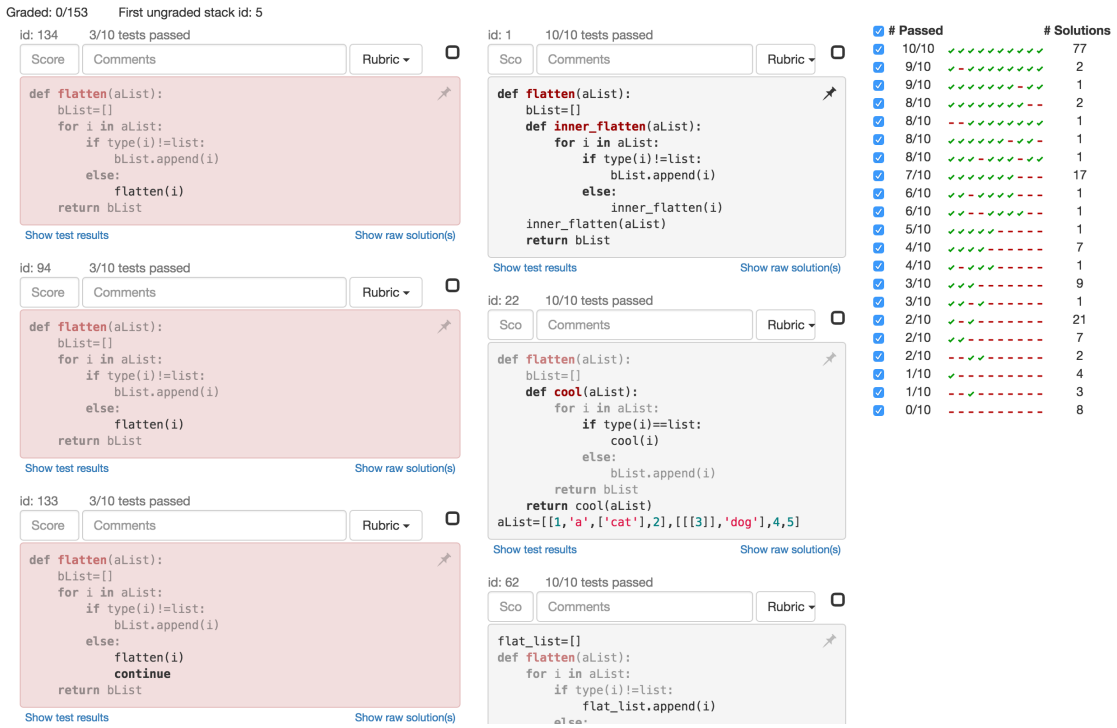


Figure 5: The prototype user interface used in the first field deployment.

together and reaching a consensus on what score to assign. Staff began by examining the submissions marked as correct by an autograder. Most submissions that passed every test case received full credit; however, in some cases points were deducted. For example, for midterm Question 4, students needed to implement a recursive function; however, some students submitted iterative solutions. For each error, staff discussed how many points to deduct, and added a new item to the built-in rubric. After grading a few submissions, groups divided up the remaining submissions and began working independently. To avoid duplicating work, groups either pinned different submissions, or pinned the same submission and had one group grade starting from the top and another starting from the bottom. Some groups used the pinning function often; however, several staff expressed displeasure with the amount of time it took the interface to update after pinning a new submission. Staff members also requested the ability to see specific tests cases and their output within the GroverCode interface, instead of relying on the MITx platform for test results. Extensive interface updates made between the first and second field deployments addressed these concerns.

GroverCode writes a log entry after each change to a grade or comment. The entry includes a timestamp, the ID of the associated submission, and the contents of the grade and comment fields for that submission. Each question has a separate log. The amount of time spent grading each question is shown in Figure 6. Very early in the process of grading Question 6, after grading only two submissions, staff discovered a discrepancy between the number of failed test cases reported by GroverCode and by MITx. Staff eventually found the problem: an error in the test suite used by MITx. However, while resolving the discrepancy, no grading took place for Question 6. After applying a fix, staff regraded the two early submissions.

Problem	Timestamp of first logged event	Timestamp of last logged event	Approximate time to grade
Question 4: power	9:21	9:45	0h 24m
Question 5: give_and_take	9:46	11:05	1h 19m
Question 6: closest_power	10:35 *	13:52	3h 17m

Figure 6: Approximate time to grade each of the three midterm questions. Currently, GroverCode does not log the time when staff begin looking at a particular problem, so there is no record of the length of discussion time before the first logged event. The cell marked with an asterisk indicates the timestamp of the first event after the discrepancy between GroverCode and MITx was resolved.

8.3 Second Field Deployment: Final Exam

175 students submitted submissions to the coding portion of the final exam, which consisted of four problems. Preparation for this field deployment was the same as for the first field deployment (see Section 8.2).

Interface. Between the two field deployments, there was significant overhaul to the GroverCode interface. For a description of the final interface, see Section 4.

Training. On the morning of the field deployment, staff received a brief explanation of the differences between the old user interface and the new user interface.

Process. As in the first field deployment, grading staff divided into small groups. Question 4 was set aside for an absent staff member, but picked up later in the day by a different TA. Again, each group independently decided to start by grading submissions

that passed every test case. Figure 7 shows a scatterplot of each logged event versus the number of test cases the associated submission passed. The grading staff used the filters extensively. Each small group selected a single error signature to view and graded every submission with that error signature before selecting a new signature. When viewing an error signature with many submissions, groups often graded the first few submissions together, then split up to grade the rest individually. A common approach involved one staff member starting from the first displayed submission while another started from the last displayed submission, consulting each other as necessary if they encountered a novel error.

The method of choosing the next error signature to grade varied from group to group. The staff grading Question 5, for example, graded error signatures with the most submissions first, then worked in descending order ending with submissions with unique error signatures (see Figure 8). In contrast, the staff grading Question 4 proceeded in order of most test cases passed to fewest test cases passed. In addition, some staff members switched from grading submissions which passed few test cases to submissions which passed many submissions to reduce the monotony of grading. Staff also made heavy use of the built-in rubric, adding deductions whenever they encountered a new mistake. Unfortunately, they encountered a bug in the rubric infrastructure for Question 7 which caused previously-added deductions to sometimes disappear.

When staff finished grading all submissions for a particular question, they switched to a different question. For example, staff finished grading Question 5 at approximately 12:30, and then moved to Question 7. This led to many staff members working on a single question. As Question 7 was the hardest to grade, most staff members ended up working on this question. To split up the work, some staff members graded error signatures in ascending order of number of test cases passed while others graded in descending order. This can be seen by the triangular structure of the plot in Figure 7.

These results show two styles of grading supported by the GroverCode interface: grading in order of number of test cases passed and grading in order of number of

submissions that passed the same test cases.

9 User Feedback

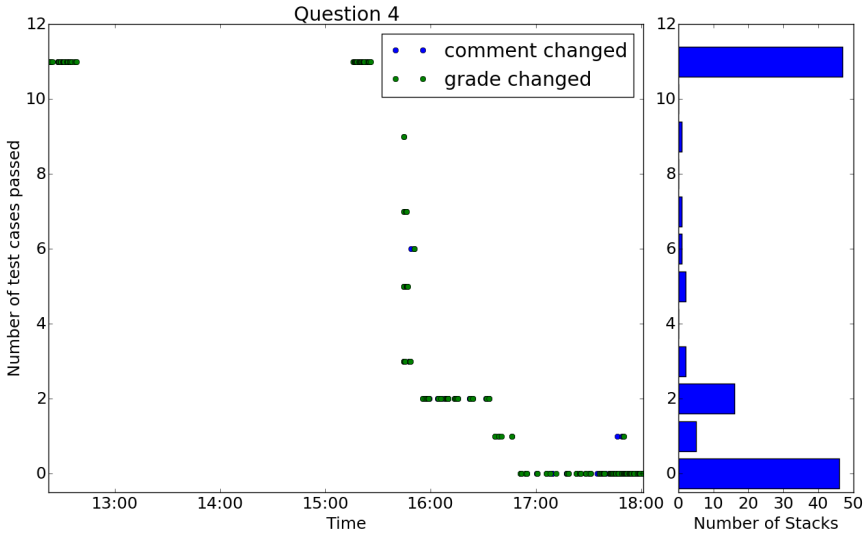
We interviewed six of the eight grading staff (not including the author) to get their high-level opinions about GroverCode and its features. Questions to staff included what they liked about using GroverCode, what they did not like, and what suggestions or feature requests, if any, they had for the future. Staff also answered questions about several specific features of the GroverCode interface: variable renaming, highlighting differences between neighboring stacks, filtering stacks by error signature, and displaying test case results underneath the associated code. Finally, staff members with prior grading experience commented on the differences between grading using GroverCode and the old method of grading using a CSV and a spreadsheet. Responses were very positive overall, with mixed reactions to various features. In general, staff thought that GroverCode gave them more control over the grading process and increased consistency while grading. Staff found GroverCode most useful when grading simple problems. Three of the six respondents remarked that they would like to continue using GroverCode in the future.

9.1 Responses to specific features

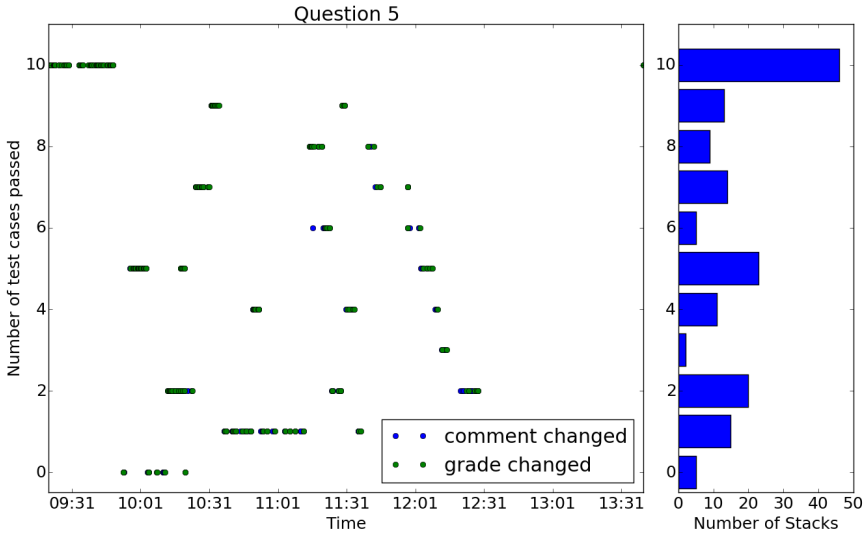
Horizontal alignment. Four of the six respondents offered positive opinions, without prompting, about the horizontal alignment of submissions. They found that comparing submissions and test case output was easier in the horizontal display than the vertical display.

Built-in rubric. Similarly, five of the six respondents mentioned that they liked the rubric feature, despite some confusion over the bug described in Section 8. One respondent remarked that it increased her self-consistency. Another found it useful for double-checking scores before entering them. However, a third respondent mentioned that although she found the rubric useful at the beginning of grading a particular problem, eventually the rubric became so full that it was easier to simply

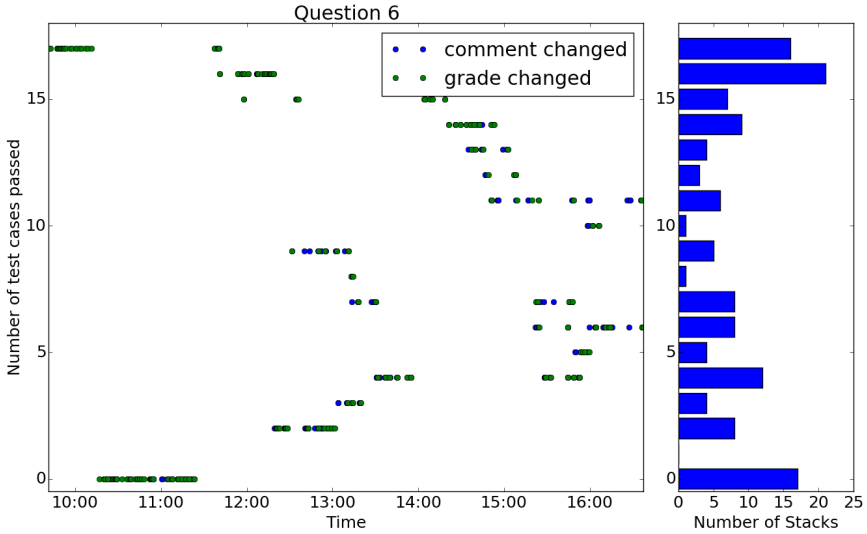
Figure 7: Grading events versus test cases passed. In each plot, the scatterplot on the left plots the time when grade or comment changed versus the number of test cases the associated stack passed. The bar chart on the right shows the number of stacks that passed that many test cases. The plots make no distinction between different error signatures that passed the same number of test cases.



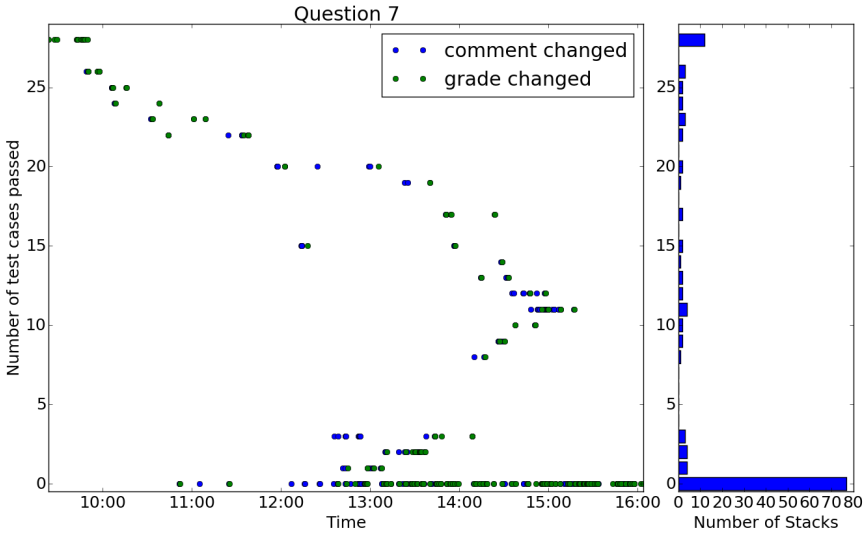
(a) Question 4. The TA assigned to this question stopped working at about 12:40 because of another commitment, and resumed grading at about 15:15. The densely clustered group of points from 15:15 to about 15:30 shows the relative ease of grading submissions marked correct compared to those marked incorrect. Additional staff members began grading at approximately 17:30, which explains the increased density after that point.



(b) Question 5. These results show an alternate grading strategy. Rather than choosing a group of submissions to grade based on the number of test cases passed, staff members chose the group containing the most submissions. See Figure 8b.

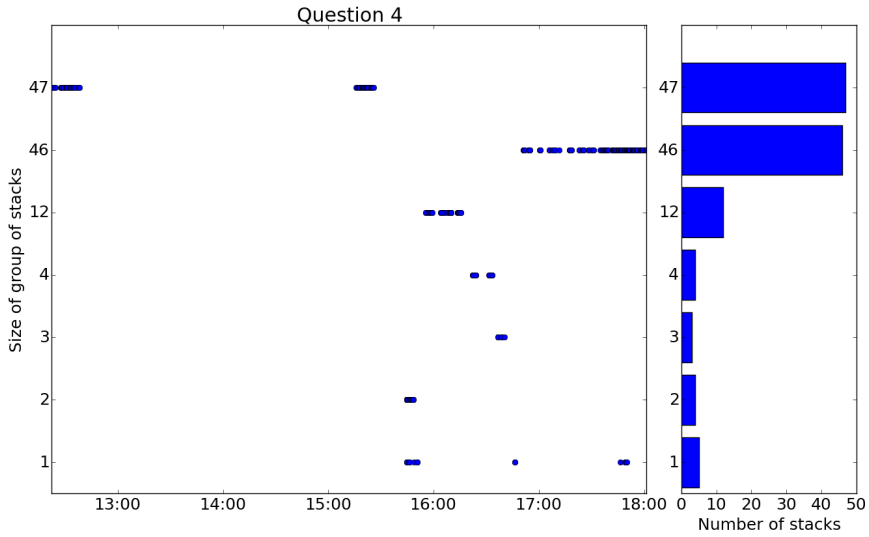


(c) Question 6.

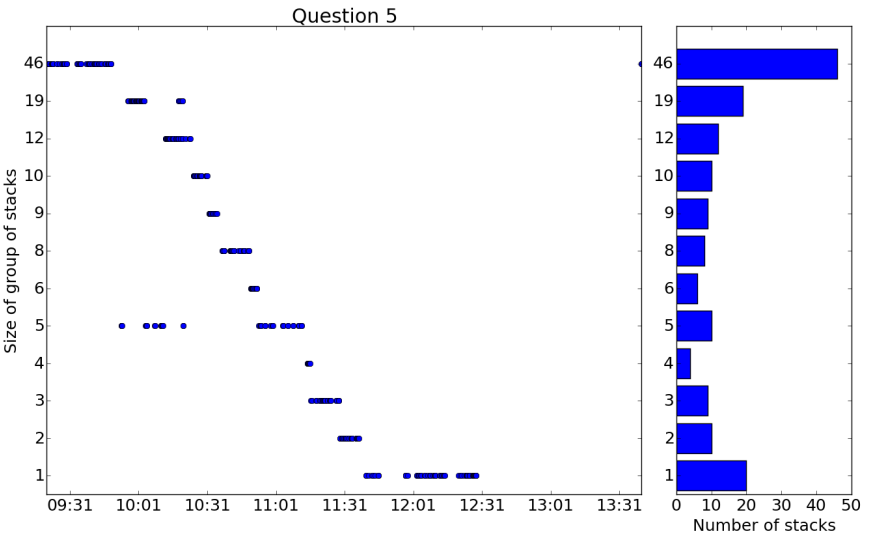


(d) Question 7. A large number of submissions fail every test case. Many such submissions nevertheless show a minimal understanding of the material and so deserve a non-zero score. See Figure 9.

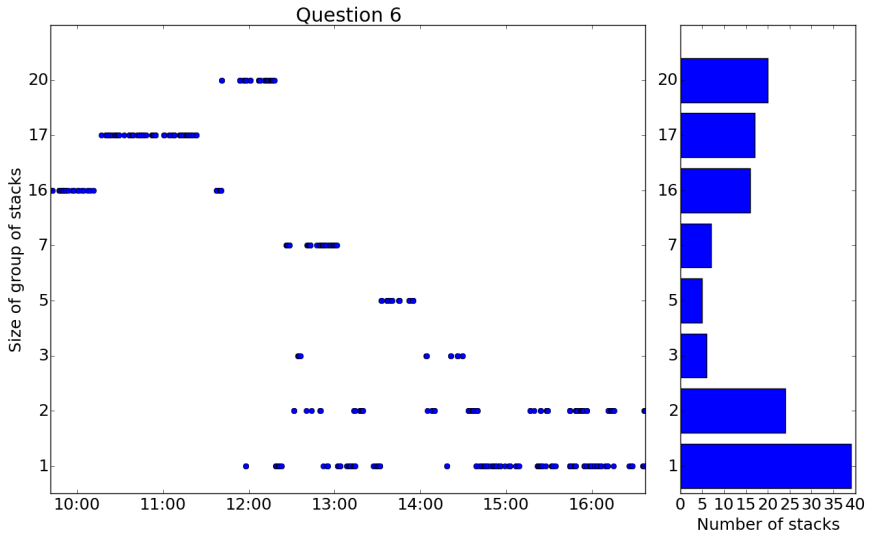
Figure 8: Grading events versus size of group. In each plot, the scatterplot on the left plots the time at which a grade or comment changed versus the number of stacks sharing an error signature with the associated stack. The bar chart on the right shows the number of stacks in groups of that size.



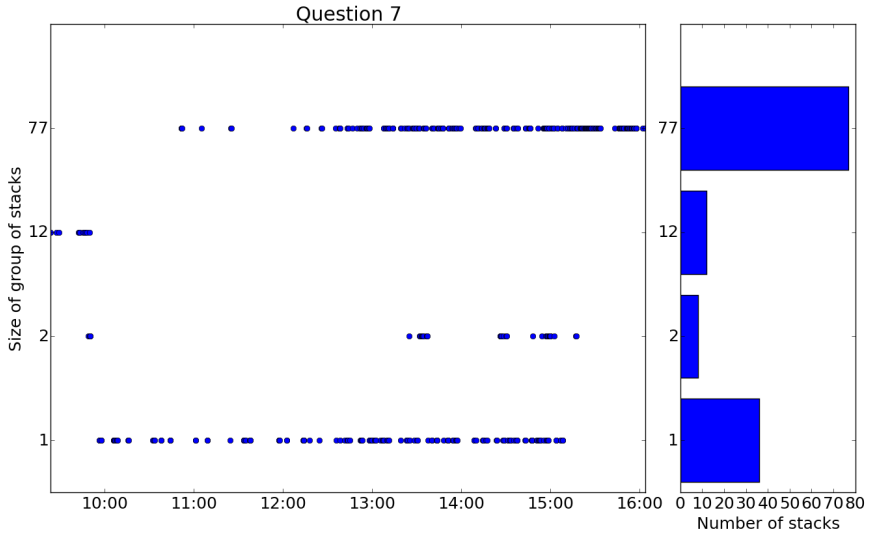
(a)



(b)

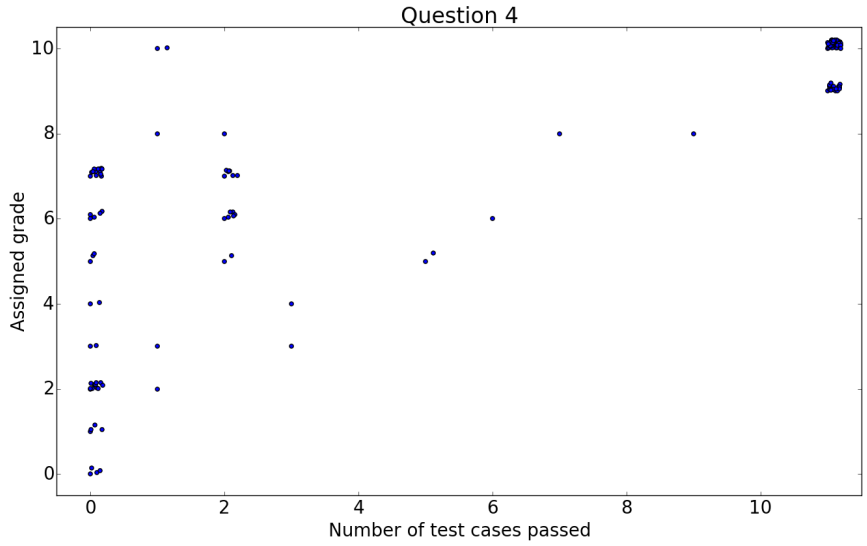

Question 6
Size of group of stacks
20
17
16
7
5
3
2
1
10:00
11:00
12:00
13:00
14:00
15:00
16:00
0 5 10 15 20 25 30 35 40
Number of stacks

(c)

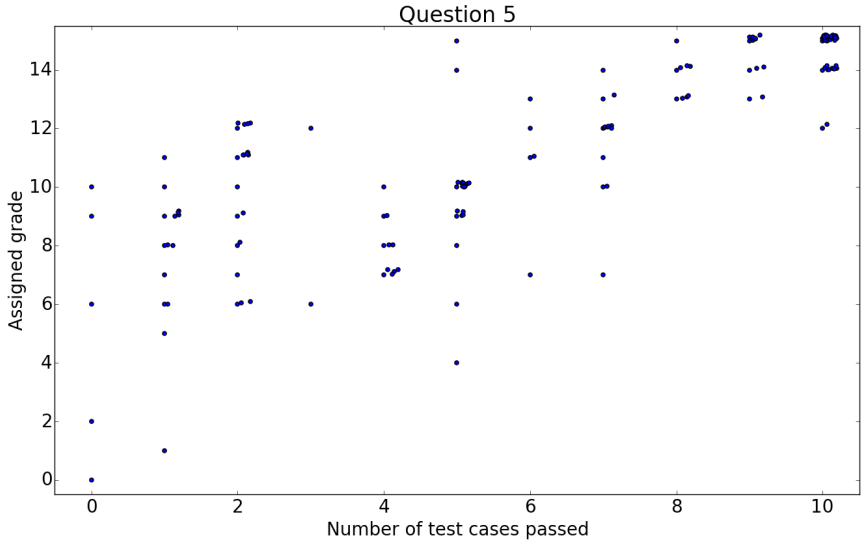

Question 7
Size of group of stacks
77
12
2
1
10:00
11:00
12:00
13:00
14:00
15:00
16:00
0 10 20 30 40 50 60 70 80
Number of stacks

(d)

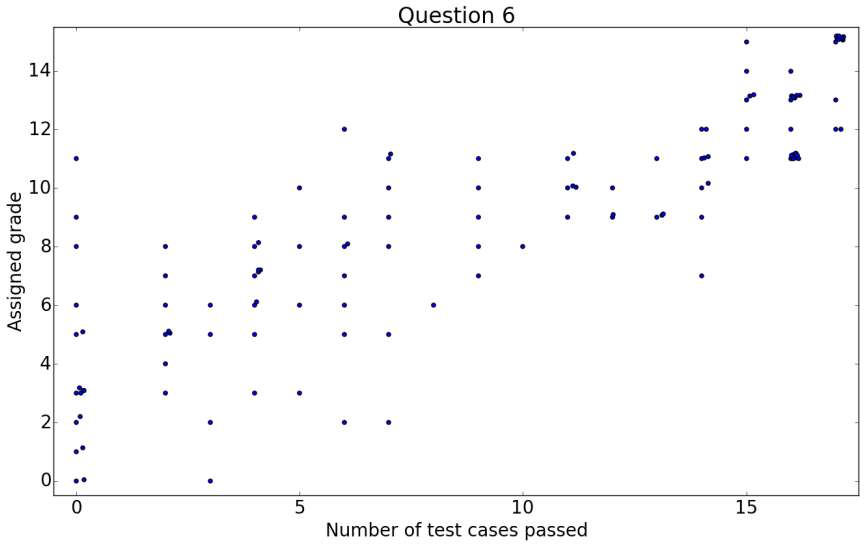
Figure 9: These plots graph the number of test cases each stack passed versus the score assigned by teaching staff. An autograder would assign the same score to every submission that passed the same number of test cases. These plots illustrate the importance of manually grading submissions.



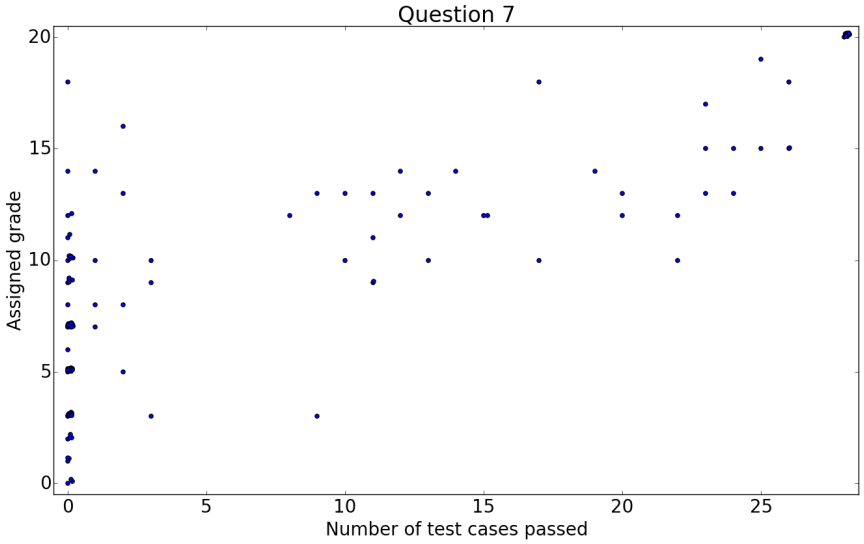
(a)



(b)


Question 6
Assigned grade
Number of test cases passed
0
2
4
6
8
10
12
14
0
5
10
15

(c)


Question 7
Assigned grade
Number of test cases passed
0
5
10
15
20
0
5
10
15
20
25

(d)

type comments manually rather than searching for particular items in the rubric.

Progress bars. Three of the six respondents expressed appreciation for the progress bars. One called this feature a "motivating factor."

Filtering stacks by error signature. This feature was absent during the first field deployment, but added by request for the second field deployment. After the second field deployment, three of the six respondents found the ability to filter by error signature helpful. The other three thought that this ability was useful in theory, but did not help them during grading because there were too many submissions with unique error signatures. However, all six respondents remarked that grouping together stacks with the same error signature made grading the whole group relatively easy, since such stacks often made the same mistake. One respondent cited this feature as a major advantage over the old method of grading submissions in a CSV file and a spreadsheet, saying that it "gave grading structure." Additionally, she reported that it eliminated a common source of frustration, where a grader encountered a submission similar to a previously graded one but could not remember what deductions applied to the previous submission. Interestingly, one respondent remarked that he associated certain failed test cases with specific mistakes, and could therefore use the error signature as "a sort of visual hash." Another respondent said exactly the opposite: she did not associate particular error signatures with specific mistakes, so the visual representation of error signatures was not especially helpful.

Renaming variables. Respondents had mixed reactions to this feature. Five of the six respondents expressed frustration with certain variable names chosen by Grover-Code, particularly names featuring subscripts. They reported that submissions with several renamed variables became less readable. Some respondents resorted to looking at the raw submissions rather than the canonicalized code for such examples. However, one of these respondents also said that she found variable renaming useful when the names were not confusing, since it increased consistency across submissions. A different respondent remarked that he initially forgot about the variable renaming and consequently wondered why all students seemed to choose the same names for their variables. He reported that after he remembered, he did not notice the vari-

able renaming while grading unless a name was particularly confusing. However, in retrospect, he thinks the feature was helpful.

Difference highlighting. One respondent said that he found this feature very helpful because it was easier to compare submissions. The other respondents reported that they either did not notice this feature or were indifferent to it.

9.2 Frustrations

Each respondent expressed frustration with one or two aspects of GroverCode, although there were several overlapping opinions. As mentioned in Section 9.1, respondents did not like grading submissions that contained several variable names with subscripts. An additional unpopular aspect of variable renaming was GroverCode's occasional renaming of the arguments of a function. Finally, four of the six respondents remarked that loading submissions and scrolling the view was occasionally too slow.

9.3 Comparison to Previous Grading Tools

Three of the six respondents had previous grading experience. All three preferred GroverCode to grading with a CSV file and a spreadsheet. These respondents all commented that they appreciated the ability to look at student code and test case results as well as enter grades via one unified interface. One respondent remarked that despite the frustrations mentioned in Section 9.2, "[Gr]overCode is already a huge improvement over the primitive tools [we used before]."

Two of the remaining respondents had no prior experience grading, but graded a small number of submissions using a CSV file during the field deployments, since GroverCode could not run every student's code. One of these respondents remarked that based on this small experience, he thought that the old method of grading would be "much more time consuming." The other respondent expressed concern about the prospect of grading using a CSV file and a spreadsheet in the future, and remarked, "I've never graded the old way, but I like [Gr]overCode."

9.4 Suggestions

Respondents offered several suggestions for features they would like to see in the future. These include activity indicators that show when a submission is being actively graded, automatic calculation of a submission's score based on the associated rubric items, the ability to add test cases on the fly, an integrated console for rerunning submissions on user-specified inputs, the ability to reorder rubric items, the ability to filter submissions based on particular rubric items, and syntax highlighting that more closely resembles other familiar tools such as IDLE or the Sublime Text editor.

10 Discussion

The two field deployments described in Section 8 demonstrate GroverCode's usefulness in a residential course. In general, the staff of 6.0001 found GroverCode helpful during the grading process. However, GroverCode is more useful when grading simple problems than when grading complex problems. As shown in Section 7, as problems get more complex, fewer students submit correct answers, and fewer submissions are grouped together into stacks. Staff found grading incorrect submissions, especially submissions that failed every test case, much more difficult than grading correct submissions. The canonicalization of incorrect submissions sometimes contributed to this difficulty, as staff often had trouble reading code that contained subscripts. Future work to increase the readability of such submissions would be very beneficial.

Staff also expressed frustration with the speed and responsiveness of the GroverCode user interface. We did not focus on these aspects in our implementation, however they are both important considerations that should not be neglected in future work. One staff member remarked, "I get impatient when grading," illustrating that speed is indeed a concern.

During the second field deployment in particular (see Section 8.3), a great deal of time was spent grading the last, most complex question. Although the increased readability mentioned above would be helpful, more investigation may suggest further ways to reduce the time required to grade such questions. Improving GroverCode's

usefulness for complex problems would be a worthwhile effort. A staff member expressed a similar sentiment, saying, "If you can perfect the more complicated [problems] it would be a really awesome tool, and it *was* a really awesome tool for more simple things."

11 Future Work

The goal of GroverCode was assisting graders of coding questions in an introductory programming course. Although course staff found GroverCode helpful, further iteration of the user interface would improve the grader experience. Features that improve the efficiency of grading would be especially helpful, such as the ability to assign a grade to multiple similar submissions simultaneously, or the ability to copy-and-paste a comment from a previously graded submission.

The problems tested with GroverCode ranged from simple to complex. Teaching staff found that for hard problems, GroverCode's attempts to increase the similarity of submissions did not reduce the cognitive load of debugging as much as they had hoped. The GroverCode pipeline would benefit from additional support for complex problems, especially problems involving classes.

GroverCode could make use of more advanced clustering algorithms and other methods from the Machine Learning domain. The ability to learn metrics of similarity between submissions rather than using empirically determined heuristics could be very powerful for clustering submissions. This ability could also be helpful for finding correct submissions that resemble incorrect submissions to help graders pinpoint errors. If GroverCode could learn which rubric items are associated with certain lines of code or other program features, it could suggest rubric items during grading.

GroverCode could also benefit from integrating with automated hint generation tools, such as those discussed in Section 2. Currently, graders find debugging incorrect student submissions the most time-consuming and difficult part of grading exams. Offloading at least some of the burden of debugging to automated tools would save teaching staff a great deal of time and effort.

Finally, several possible avenues of future work suggested in [3] but not yet implemented could be explored. These include expanding OverCode to other programming languages besides Python, integrating OverCode with an autograder such as that used by the MITx platform, and adjusting the OverCode interface to provide benefit to students as well as staff.

12 Conclusion

The original OverCode system described in [3] is novel because it canonicalizes student code to increase human readability. We expand GroverCode to accommodate submissions marked incorrect by an autograder, and to canonicalize such submissions in a similar fashion. We also add interface features for grading submissions. A group of residential instructors found GroverCode a helpful tool for grading exams. We hope GroverCode will continue to provide benefit to residential course staff in the future.

Bibliography

[1] Sumit Basu, Chuck Jacobs, and Lucy Vanderwende. Powergrading: a clustering approach to amplify human effort for short answer grading. *TACL*, 1:391–402, 2013.

[2] Michael Brooks, Sumit Basu, Charles Jacobs, and Lucy Vanderwende. Divide and correct: using clusters to grade short answers at scale. In *Learning at Scale*, pages 89–98, 2014.

[3] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 22(2):7, 2015.

[4] Sebastian Gross, Bassam Mokbel, Barbara Hammer, and Niels Pinkwart. Feedback provision strategies in intelligent tutoring systems based on clustered solution spaces. *DeLFI 2012: Die 10. e-Learning Fachtagung Informatik*, 2012.

[5] Philip J. Guo. Online Python Tutor: Embeddable web-based program visualization for CS education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM.

[6] Kelly Rivers and Kenneth R Koedinger. Automatic generation of programming feedback: A data-driven approach. In *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)*, page 50, 2013.

[7] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26, 2013.