A

OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale

ELENA L. GLASSMAN, MIT CSAIL
JEREMY SCOTT, MIT CSAIL
RISHABH SINGH, MIT CSAIL
PHILIP J. GUO, MIT CSAIL and University of Rochester
ROBERT C. MILLER, MIT CSAIL

In MOOCs, a single programming exercise may produce thousands of solutions from learners. Understanding solution variation is important for providing appropriate feedback to students at scale. The wide variation among these solutions can be a source of pedagogically valuable examples, and can be used to refine the autograder for the exercise by exposing corner cases. We present OverCode, a system for visualizing and exploring thousands of programming solutions. OverCode uses both static and dynamic analysis to cluster similar solutions, and lets teachers further filter and cluster solutions based on different criteria. We evaluated OverCode against a non-clustering baseline in a within-subjects study with 24 teaching assistants, and found that the OverCode interface allows teachers to more quickly develop a high-level view of students' understanding and misconceptions, and to provide feedback that is relevant to more students' solutions.

Categories and Subject Descriptors: H.5.m. [**Information Interfaces and Presentation (e.g., HCI)**]: Miscellaneous

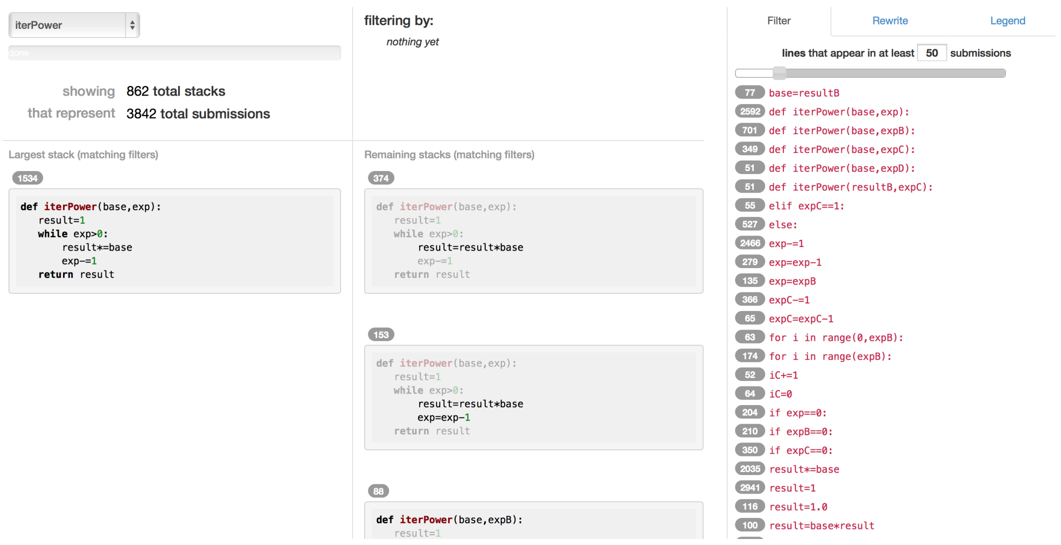
1. INTRODUCTION

Intelligent tutoring systems (ITSes), Massive Open Online Courses (MOOCs), and websites like Khan Academy and Codecademy are now used to teach programming courses at a massive scale. In these courses, a single programming exercise may produce thousands of solutions from learners, which presents both an opportunity and a challenge. For teachers, the wide variation among these solutions can be a source of pedagogically valuable examples [Marton et al. 2013], and understanding this variation is important for providing appropriate, tailored feedback to students [Basu et al. 2013; Huang et al. 2013]. The variation can also be useful for refining evaluation rubrics and exposing corner cases in automatic grading tests.

Sifting through thousands of solutions to understand their variation and find pedagogically valuable examples is a daunting task, even if the programming exercises are simple and the solutions are only tens of lines of code long. Without tool support, a teacher may not read more than 50-100 of them before growing frustrated with the tedium of the task. Given this small sample size, teachers cannot be expected to develop a thorough understanding of the variety of strategies used to solve the problem, or produce instructive feedback that is relevant to a large proportion of learners, or find unexpected interesting solutions.

An information visualization approach would enable teachers to explore the variation in solutions at scale. Existing techniques [Gaudencio et al. 2014; Huang et al. 2013; Nguyen et al. 2014] use a combination of clustering to group solutions that are semantically similar, and graph visualization to show the variation between these clusters. These clustering algorithms perform pairwise comparisons that are quadratic in both the number of solutions and in the size of each solution, which scales poorly to thousands of solutions. Graph visualization also struggles with how to label the graph node for a cluster, because it has been formed by a complex combination of code features. Without meaningful labels for clusters in the graph, the rich information of the learners' solutions is lost and the teacher's ability to understand variation is weakened.

In this paper we present OverCode, a system for visualizing and exploring the variation in thousands of programming solutions. OverCode is designed to visualize correct solutions, in the sense that they already passed the automatic grading tests typically used in a programming class at scale. The autograder cannot offer any further feedback on these correct solutions, and yet there may still be good and bad variations on correct solutions that are pedagogically valuable to highlight and discuss. OverCode aims to help teachers understand solution variation so that they can provide appropriate feedback to students at scale.



showing 862 total stacks
that represent 3842 total submissions

filtering by:
nothing yet

Largest stack (matching filters)

1534
```
def iterPower(base,exp):
   result=1
   while exp>0:
       result*=base
       exp-=1
   return result
```

Remaining stacks (matching filters)

374
```
def iterPower(base,exp):
   result=1
   while exp>0:
       result=result*base
       exp-=1
   return result
```

153
```
def iterPower(base,exp):
   result=1
   while exp>0:
       result=result*base
       exp=exp-1
   return result
```

88
```
def iterPower(base,expB):
   result=1
```

lines that appear in at least 50 submissions

```
77    base=resultB
2592  def iterPower(base,exp):
701   def iterPower(base,expB):
349   def iterPower(base,expC):
51    def iterPower(base,expD):
51    def iterPower(resultB,expC):
55    elif expC==1:
527   else:
2466  exp-=1
279   exp=exp-1
135   exp=expB
366   expC-=1
65    expC=expC-1
63    for i in range(0,expB):
174   for i in range(expB):
52    iC+=1
64    iC=0
204   if exp==0:
210   if expB==0:
350   if expC==0:
2035  result*=base
2941  result=1
116   result=1.0
100   result=base*result
```

Fig. 1. The OverCode user interface. The top left panel shows the number of clusters, called *stacks*, and the total number of solutions visualized. The next panel down in the first column shows the largest stack, while the second column shows the remaining stacks. The third column shows the lines of code occurring in the cleaned solutions of the stacks together with their frequencies.

OverCode uses a novel clustering technique that creates clusters of identical cleaned code, in time linear in both the number of solutions and the size of each solution. The cleaned code is readable, executable, and describes every solution in that cluster. The cleaned code is shown in a visualization that puts code front-and-center (Figure 1). In OverCode, the teacher reads through code solutions that each represent an entire cluster of solutions that look and act the same. The differences between clusters are highlighted to help teachers discover and understand the variations among submitted solutions. Clusters can be filtered by the lines of code within them. Clusters can also be merged together with *rewrite rules* that collapse variations that the teacher decides are unimportant.

A cluster in OverCode is a set of solutions that perform the same computations, but may use different variable names or statement order. OverCode uses a lightweight dynamic analysis to generate clusters, which scales linearly with the number of solutions. It clusters solutions whose variables take the same sequence of values when executed on test inputs and whose set of constituent lines of code are syntactically the same. An important component of this analysis is to rename variables that behave the same across different solutions. The renaming of variables serves three main purposes. First, it lets teachers create a mental mapping between variable names and their behavior which is consistent across the entire set of solutions. This may reduce the cognitive load for a teacher to understand different solutions. Second, it helps clustering by reducing variation between similar solutions. Finally, it also helps make the remaining differences between different solutions more salient.

In two user studies with a total of 24 participants who each looked at thousands of solutions from an introductory programming MOOC, we compared the OverCode interface with a baseline interface that showed original unclustered solutions. When using OverCode, participants felt that they were able to develop a better high-level view of the students' understandings and misconceptions. While participants didn't necessarily read more lines of code in the OverCode interface than in the baseline, the code they did read came from clusters containing a greater percentage of all the submitted solutions. Participants also drafted mock class forum posts about common good and bad solutions that were relevant to more solutions (and the students who wrote them) when using OverCode as compared to the baseline.

The main contributions of this paper are:

— a novel visualization that shows similarity and variation among thousands of solutions, with cleaned code shown for each variant.
— an algorithm that uses the behavior of variables to help cluster solutions and generate the cleaned code for each cluster of solutions.
— two user studies that show this visualization is useful for giving teachers a birds-eye view of thousands of students' solutions.

2. RELATED WORK

There is a growing body of work on both the frontend and backend required to manage and present the large volumes of solutions gathered from MOOCs, intelligent tutors, online learning platforms, and large residential classes. The backend necessary to analyze solutions expressed as code has followed from prior work in fields such as program analysis, compilers, and machine learning. A common goal of this prior work is to help teachers monitor the state of their class, or provide solution-specific feedback to many students. However, there has not been much work on developing interactive user interfaces that enable a teacher to navigate the large space of student solutions.

We first present here a brief review of the state of the art in the backend, specifically about analyzing code generated by students who are independently attempting to implement the same function. This will place our own backend in context. We then review the information visualization principles and systems that inspired our frontend contributions.

2.1. Related Work in Program Analysis

2.1.1. Canonicalization and Semantics-Preserving Transformations. When two pieces of code have different syntax, and therefore different abstract syntax trees (ASTs), they may still be semantically equivalent. A teacher viewing the code may want to see those syntactic differences, or may want to ignore them in order to focus on semantic differences. Semantics-preserving transformations can reduce or eliminate the syntactic differences between code. Applying semantics-preserving transformations, sometimes referred to as canonicalization or standardization, has been used for a variety of applications, including detecting clones [Baxter et al. 1998] and automatic "transform-based diagnosis" of bugs in students' programs written in programming tutors [Xu and Chee 2003].

OverCode also canonicalizes solutions, using variable renaming. OverCode's canonicalization is novel in that its design decisions were made to maximize *human readability* of the resulting code. As a side-effect, syntactic differences between answers are also reduced.

2.1.2. Abstract Syntax Tree-based Approaches. Huang et al. [2013] worked with short Matlab/Octave functions submitted online by students enrolled in a machine learning MOOC. The authors generate an AST for each solution to a problem, and calculate the tree edit distance between all pairs of ASTs, using the dynamic programming edit distance algorithm presented by Shasha et al. [1994]. Based on these computed edit distances, clusters of syntactically similar solutions are formed. The algorithm is quadratic in both the number of solutions and the size of the ASTs. Using a computing cluster, the Shasha algorithm was applied to just over a million solutions.

Calculating tree-edit distances between all pairs of ASTs allows Huang et al. to analyze differences within each line. It's also computationally expensive, with quadratic complexity both in the number of solutions and the size of the ASTs [Huang et al. 2013]. The OverCode analysis pipeline does not reason about differences any finer than a line of code, but it has linear complexity in the number of solutions and in the size of the ASTs.

Codewebs [Nguyen et al. 2014] created an index of "code phrases" for over a million submissions from the same MOOC and semi-automatically identified equivalence classes across these phrases, using a data-driven, probabilistic approach. The Codewebs search engine accepts queries in the form of subtrees, subforests, and contexts that are subgraphs of an AST. A teacher labels a set of AST subtrees considered semantically meaningful, and then queries the search engine to extract all equivalent subtrees from the dataset. OverCode does analyze the AST of student solutions but only in order to reformat code and rename variables that behave similarly on a test case. All further code comparison is done through string matching lines of code that have consistent formatting and variable names.

Both Codewebs [Nguyen et al. 2014] and Huang et al. [2013] use unit test results and AST edit distance to identify clusters of submissions that could potentially receive the same feedback from a teacher. These are non-interactive systems that require hand-labeling in the case of Codewebs, or a computing cluster in the case of Huang et al. In contrast, OverCode's pipeline does not require hand-labeling and runs in minutes on a laptop, then presents the results in an interactive user interface.

2.1.3. Supervised Machine Learning and Hierarchical Pairwise Comparison. Semantic equivalence is another way of saying that two solutions have the same schema. A *schema*, in the context of programming, is a high-level cognitive construct by which humans understand or generate code to solve problems [Soloway and Ehrlich 1984]. For example, two programs that implement bubble sort have the same schema, bubble sort, even though they may have different low-level implementations. Taherkhani et al. [2012; 2013] used supervised machine learning methods to successfully identify which of several sorting algorithms a solution used. Each solution is represented by statistics about language constructs, measures of complexity, and detected roles of variables. Variable roles are determined based on variable behavior. OverCode identifies common variables based on variable behavior as well. Both methods consider the sequence of values that variables are assigned to, but OverCode does not attempt to categorize variable behavior as one of a set of predefined roles. Similarly, Taherkhani et al.'s method can identify sorting algorithms that have already been analyzed and included in its training dataset. OverCode, in contrast, handles problems for which the algorithmic schema is not already known.

Luxton-Reilly et al. [2013] label types of variations as structural, syntactic, or presentation-related. The structural similarity between solutions in a dataset is captured by comparing their control flow graphs. If the control flow of two solutions is the same, then the syntactic variation within the blocks of code is compared by looking at the sequence of token classes. Presentation-based variation, such as variable names and spacing, is only examined when two solutions are structurally and syntactically the same. In contrast, our approach is not hierarchical, and uses dynamic information in addition to syntactic information.

2.1.4. Program Synthesis. There has also been work on analyzing each student solution individually to provide more precise feedback. Singh et al. [2013] use a constraint-based synthesis algorithm to find the minimal changes needed to make an incorrect solution functionally equivalent to a reference implementation. The changes are specified in terms of a problem-specific error model that captures the common mistakes students make on a particular problem.

Rivers and Koedinger [2013] propose a data-driven approach to create a solution space consisting of all possible paths from the problem statement to a correct solution. To project code onto this solution space, the authors apply a set of normalizing program transformations to simplify, anonymize, and order the program's syntax. The solution space can then be used to locate the potential learning progression for a student submission and provide hints on how to correct their attempt. Unlike OverCode's variable renaming method, which reflects the most common names chosen by students, Rivers and Koedinger replace student variable names with arbitrary symbols, i.e. `daysInMonth` might be mapped to `v0`.

Singh et al. and Rivers and Koedinger focus on providing hints to students along their path to a correct solution. Instead of providing hints, the aim of our work is to help instructors navigate the space of *correct* solutions and therefore techniques based on checking only the functional correctness are not helpful in computing similarities and differences between such solutions.

2.1.5. Code Comparison Tools. File comparison tools, such as Apple FileMerge, Microsoft WinDiff, and Unix diff, are a class of tools that analyze and present differences between files. Highlighting indicates inserted, deleted, and changed text. Unchanged text is collapsed. Some of these tools are customized for analyzing code, such as Code Compare. They are also integrated into existing integrated development environments (IDE), including IntelliJ IDEA and Eclipse. These code-specific comparison tools may match methods rather than just comparing lines. Three panes side-by-side are used to show code during three-way merges of file differences. There are tools, e.g. KDiff3, which will show the differences between four files when performing a distributed version control merge operation, but that appears to be an upper limit. These tools do not scale beyond comparing a handful of programs simultaneously. OverCode can show hundreds or thousands of solutions simultaneously, and its visualization technique dims the lines that are shared with the most common solution, rather than using colors to indicate inserted or deleted lines.

MOSS [Schleimer et al. 2003] is a widely used system for finding similarities across student solutions for detecting plagiarism. MOSS uses a windowing technique to select fingerprints from hashes of k-grams from a solution. It first creates an index mapping fingerprints to corresponding locations for all solutions. It then fingerprints each solution again to compute the list of matching fingerprints for the solution. Finally, it rank-orders the fingerprint matches by their size for each pair of solution match. This algorithm enables MOSS to find partial matches between two solutions that are in different positions with good accuracy. OverCode, on the other hand, uses a simple linear algorithm to create stacks of solutions with the same canonical form. It uses an equivalence based on the set of statements in a solution to capture position-independent statement matches.

2.2. Related Work in User Interfaces for Solution Visualization

Several user interfaces have been designed for providing grades or feedback to students at scale, and for browsing large collections in general, not just student solutions.

Basu et al. [2013] provide a novel user interface for *powergrading* short-answer questions. Powergrading means assigning grades or writing feedback to many similar answers at once. The backend uses machine learning that is trained to cluster answers, and the frontend allows teachers to read, grade or provide feedback to those groups of similar answers simultaneously. Teachers can also discover common misunderstandings. The value of the interface was verified in a study of 25 teachers looking at their visual interface with clustered answers. When compared against a baseline interface, the teachers assigned grades to students substantially faster, gave more feedback to students, and developed a “high-level view of students’ understanding and misconceptions” [Brooks et al. 2014].

At the intersection of information visualization and program analysis is Cody[1], an informal learning environment for the Matlab programming language. Cody does not have a teaching staff but does have a *solution map* visualization to help students discover alternative ways to solve a problem. A solution map plots each solution as a point against two axes: time of submission on the horizontal axis, and code size on the vertical axis, where *code size* is the number of nodes in the parse tree of the solution. Despite the simplicity of this metric, solution maps can provide quick and valuable insight when assessing large numbers of solutions [Glassman et al. 2013].

OverCode has also been inspired by information visualization projects like WordSeer [Muralidharan and Hearst 2013; Muralidharan et al. 2013] and CrowdScape [Rzeszotarski and Kittur 2012]. WordSeer helps literary analysts navigate and explore texts, using query words and phrases [Muralidharan and Hearst 2011]. CrowdScape gives users an overview of crowd-workers’ performance on tasks. An overview of crowd-workers each performing on a task, and an overview of submitted code, each executing a test case, are not so different, from an information presentation point of view.

3. OVERCODE

We now describe the OverCode user interface. OverCode is an information visualization application for teachers to explore student program solutions. The OverCode interface allows the user to scroll, filter, and *stack* solutions. OverCode uses the metaphor of stacks to denote collections of similar solutions, where each stack shows a *cleaned* solution from the corresponding collection of identical

[1]`mathworks.com/matlabcentral/cody`

cleaned solutions it represents. These cleaned solutions have strategically renamed variables and can be filtered by the cleaned lines of code they contain. Cleaned solutions can also be rewritten when users compose and apply a *rewrite rule*, which can eliminate differences between cleaned solutions and therefore combine stacks of cleaned solutions that have become identical.

We iteratively designed and developed the OverCode interface based on continuous evaluation by the authors, feedback from teachers and peers, and by consulting principles from the information visualization literature. A screenshot of OverCode visualizing `iterPower`, one of the problems from our dataset, is shown in Figure 1. In this section, we describe the intended use cases and the user interface. In Section 4, the backend program analysis pipeline is described in detail.

3.1. Target Users and Applications

The target users of OverCode are teaching staff of introductory programming courses. Teaching staff may be undergraduate lab assistants who help students debug their code; graduate students who grade assignments, help students debug, and manage recitations and course forums; and lecturing professors who also compose the major course assessments. Teachers using OverCode may be looking for common misconceptions, creating a grading rubric, or choosing pedagogically valuable examples to review with students in a future lesson.

3.1.1. Misconceptions and Holes in Students' Knowledge. Students just starting to learn programming can have a difficult time understanding the language constructs and different API methods. They may use them suboptimally, or in non-standard ways. OverCode may help instructors identify these common misconceptions and holes in knowledge, by highlighting the differences between stacks of solutions. Since the visualized solutions have already been tested and found correct by an autograder, these highlighted differences between cleaned solutions may be convoluted variations in construct usage and API method choices that have not been flagged by the Python interpreter or caused the failure of a unit test. Convoluted code may suggest a misconception.

3.1.2. Grading Rubrics. It is a difficult task to create grading rubrics for checking properties such as design and style of solutions. Therefore most autograders resort to checking only functional correctness of solutions by testing them against a test suite of input-output pairs. OverCode enables teachers to identify the style, structure, and relative frequency of the variation within correct solutions. Unlike traditional ways of creating a grading rubric, where an instructor may go through a set of solutions, revising the rubric along the way, instructors can use OverCode to first get a high-level overview of the variations before designing a corresponding rubric. Teachers may also see incorrect solutions not caught by the autograder.

3.1.3. Pedagogically Valuable Examples. There can be a variety of ways to solve a given problem and express it in code. If an assignment allows students to generate different solutions, e.g., recursive or iterative, to fulfill the same input-output behavior, OverCode will show separate stacks for each of these different solutions, as well as stacks for every variant of those solutions. OverCode helps teachers filter through solutions to find different examples of solutions to the same problem, which may be pedagogically valuable. According to Variation Theory [Marton et al. 2013], students can learn through concrete examples of these multiple solutions, which vary along various conceptual dimensions.

3.2. User Interface

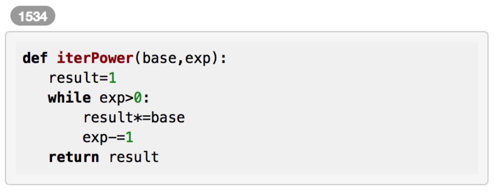
The OverCode user interface is the product of an iterative design process with multiple stages, including paper prototypes and low-fidelity web-browser-based prototypes. Prototype iterations were used and critiqued by members of our research group and by several teaching staff of an introductory Python programming course. While exploring the low-fidelity prototypes, these teachers talked aloud about their hopes for what the tool could do, frustrations with its current form, and their frustrations with existing solution-viewing tools and processes. This feedback was incorporated into the final design.

The OverCode user interface is divided into three columns. The top-left panel in the first column shows the problem name, the *done* progress bar, the number of stacks, the number of visualized stacks given the current filters and rewrite rules, and the total number of solutions those visualized stacks contain. The panel below shows the largest stack that represents the most common solution. Side by side with the largest stack, the remaining solution stacks appear in the second panel. Through scrolling, any stack can be horizontally aligned with the largest stack for easier comparison. The third panel has three different tabs that provide static and dynamic information about the solutions, and the ability to filter and combine stacks.

As shown in Figure 1, the default tab shows a list of lines of code that occur in different cleaned solutions together with their corresponding frequencies. The stacks can be filtered based on the occurrence of one or more lines (Filter tab). The column also has tabs for *Rewrite* and *Legend*. The Rewrite tab allows a teacher to provide rewrite rules to collapse different stacks with small differences into a larger single stack. The Legend tab shows the dynamic values that different program variables take during the execution of programs over a test case. We now describe different features of OverCode in more detail.

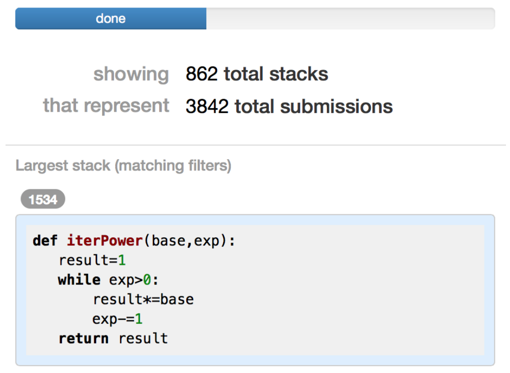
3.2.1. Stacks. A stack in OverCode denotes a set of similar solutions that are grouped together based on a similarity criterion defined in Section 4. For example, a stack for the `iterPower` problem is shown in Figure 2(a). The size of each stack is shown in a pillbox at the top-left corner of the stack. The count denotes how many solutions are in the stack, and can also be referred to as the stack *size*. Stacks are listed in the scrollable second panel from largest to smallest. The solution on the top of the stack is a cleaned solution that describes all the solutions in the stack. See Section 4 for details on the cleaning process.

Each stack can also be clicked. After clicking a stack, the border color of the stack changes and the *done* progress bar is updated to reflect the percentage of total solutions clicked, as shown in Figure 2(b). This feature is intended to help users remember which stacks they have already read or analyzed, and keep track of their progress. Clicking on a large stack, which represents a significant fraction of the total solutions, is reflected by a large change in the *done* progress bar.



```
def iterPower(base,exp):
    result=1
    while exp>0:
        result*=base
        exp-=1
    return result
```

(a)



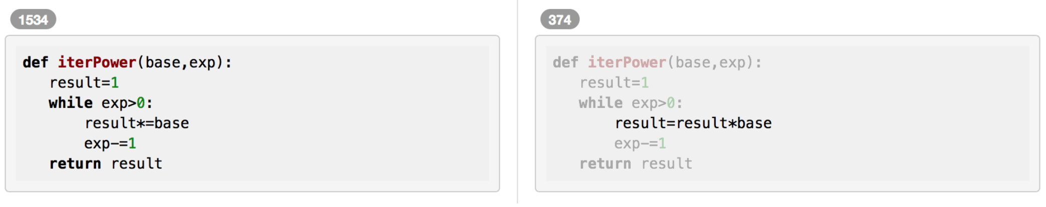
showing 862 total stacks
that represent 3842 total submissions

```
def iterPower(base,exp):
    result=1
    while exp>0:
        result*=base
        exp-=1
    return result
```

(b)

Fig. 2. (a) A stack consisting of 1534 similar `iterPower` solutions. (b) After clicking a stack, the border color of the stack changes and the done progress bar denotes the corresponding fraction of solutions that have been checked.

3.2.2. Showing Differences between Stacks. OverCode allows teachers to compare smaller stacks, shown in the second column, with the largest stack, shown in the first column. The lines of code in the second column that also appear in the set of lines in the largest stack are dimmed so that only the differences between the smaller stacks and the largest stack are apparent. For example, Figure 3 shows the differences between the cleaned solutions of the two largest stacks. In earlier



1534
```
def iterPower(base,exp):
   result=1
   while exp>0:
       result*=base
       exp-=1
   return result
```

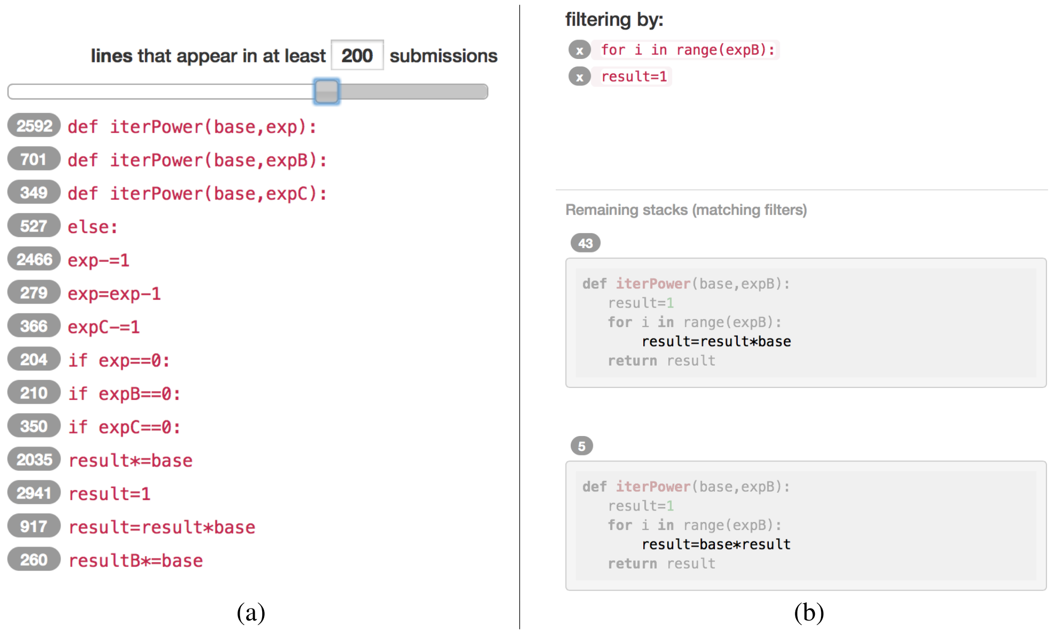
374
```
def iterPower(base,exp):
   result=1
   while exp>0:
       result=result*base
       exp-=1
   return result
```

Fig. 3. Similar lines of code between two stacks are dimmed out such that only differences between the two stacks are apparent.

iterations of the user interface, lines in stacks that were not shared with the largest stack were highlighted in yellow, but this produced a lot of visual noise. By dimming the lines in stacks that *are* shared with the largest stack, we reduced the visible noise, while still keeping differences between stacks salient.

3.2.3. Filtering Stacks by Lines of Code. The third column of OverCode shows the list of lines of code occurring in the solutions together with their frequencies (numbered pillboxes). The interface has a slider that can be used to change the threshold value, which denotes the number of solutions in which a line should appear for it to be included in the list. For example, by dragging the slider to 200 in Figure 4(a), OverCode only shows lines of code that are present in at least 200 solutions. This feature was added as a response to the length of the unfiltered list of code lines, which was long enough to make skimming for common code lines difficult.

Users can filter the stacks by selecting one or more lines of code from the list. After each selection, only stacks whose cleaned solutions have those selected lines of code are shown. Figure 4(b) shows a filtering of stacks that have a `for` loop, specifically the line of code `for i in range(expB)`, and that assign 1 to the variable `result`.



lines that appear in at least 200 submissions

2592	`def iterPower(base,exp):`
701	`def iterPower(base,expB):`
349	`def iterPower(base,expC):`
527	`else:`
2466	`exp-=1`
279	`exp=exp-1`
366	`expC-=1`
204	`if exp==0:`
210	`if expB==0:`
350	`if expC==0:`
2035	`result*=base`
2941	`result=1`
917	`result=result*base`
260	`resultB*=base`

(a)

filtering by:

x `for i in range(expB):`

x `result=1`

Remaining stacks (matching filters)

43
```
def iterPower(base,expB):
   result=1
   for i in range(expB):
       result=result*base
   return result
```

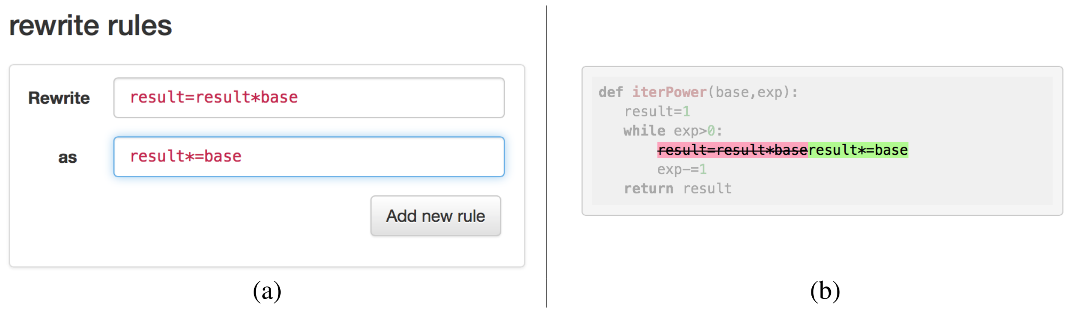
5
```
def iterPower(base,expB):
   result=1
   for i in range(expB):
       result=base*result
   return result
```

(b)

Fig. 4. (a) The slider allows filtering of the list of lines of code by the number of solutions in which they appear. (b) Clicking on a line of code adds it to the list of lines by which the stacks are filtered.

3.2.4. Rewrite Rules. There are often small differences between the cleaned solutions that can lead to a large number of stacks for a teacher to review. OverCode provides *rewrite rules* by which users can collapse these differences and ignore variation they do not need to see. This feature comes from experience with early prototypes. After observing a difference between stacks, like the use of `xrange` instead of `range`, users wanted to ignore that difference in order to more easily find other differences.

A rewrite rule is described with a left hand side and a right hand side as shown in Figure 5(a). The semantics of a rewrite rule is to replace all occurrences of the left hand side expression in the cleaned solutions with the corresponding right hand side. As the rewrite rules are entered, OverCode presents a preview of the changes in the cleaned solutions as shown in Figure 5(b). After the application of the rewrite rules, OverCode collapses stacks that now have the same cleaned solutions because of the rewrites. For example, after the application of the rewrite rule in Figure 5(a), OverCode collapses the two biggest iterPower stacks from Figure 1 of sizes 1534 and 374, respectively, into a single stack of size 1908. Other pairs of stacks whose differences have now been removed by the rewrite rule are also collapsed into single stacks. As shown in Figure 6(a), the number of stacks now drop from 862 to 814.

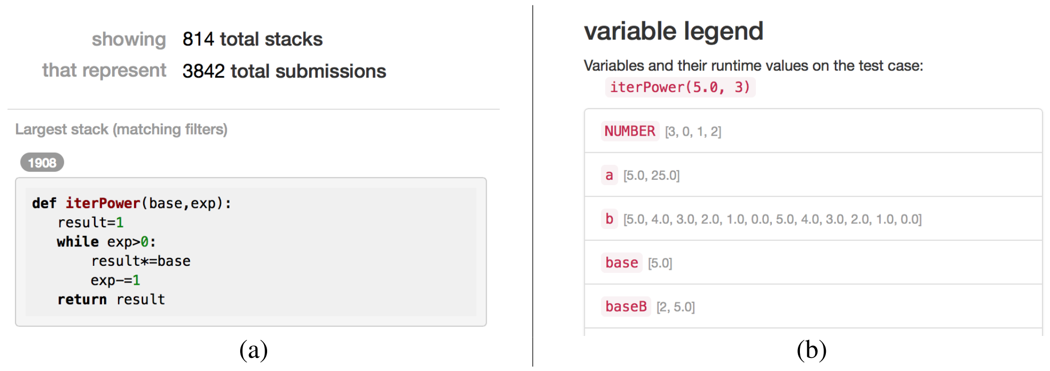


Rewrite `result=result*base`

as `result*=base`

```
def iterPower(base,exp):
    result=1
    while exp>0:
        ~~result=result*base~~result*=base
        exp-=1
    return result
```

Fig. 5. (a) An example rewrite rule to replace all occurrences of statement `result = base * result` with `result *= base`. (b) The preview of the changes in the cleaned solutions because of the application of the rewrite rule.



showing 814 total stacks

that represent 3842 total submissions

Largest stack (matching filters)

1908

```
def iterPower(base,exp):
    result=1
    while exp>0:
        result*=base
        exp-=1
    return result
```

variable legend

Variables and their runtime values on the test case:
`iterPower(5.0, 3)`

NUMBER	[3, 0, 1, 2]
a	[5.0, 25.0]
b	[5.0, 4.0, 3.0, 2.0, 1.0, 0.0, 5.0, 4.0, 3.0, 2.0, 1.0, 0.0]
base	[5.0]
baseB	[2, 5.0]

Fig. 6. (a) The merging of stacks after application of the rewrite rule shown in Figure 5. (b) The variable legend shows the sequence of dynamic values that all program variables in cleaned solutions take over the course of execution on a given test case.

3.2.5. Variable Legends. OverCode also shows the sequence of values that variables in the cleaned solutions take on, over the course of their execution on a test case. As described in Section 4, a variable is identified by the sequence of values it takes on during the execution of the test case. Figure 6(b) shows a snapshot of the variable values for the `iterPower` problem. The goal of presenting this dynamic information associated with common variable names is to help users understand the behavior of each cleaned solution, and further explore the variations among solutions that do not have the same common variables. When this legend was originally added to the user interface, clicking on a common variable name would filter for all solutions that contained an instance of that variable. Some pilot users found this feature confusing, rather than empowering. As a result, it was removed from OverCode before running both user studies. At least one study participant, upon realizing the value of the legend, wished that the original click-to-filter-by-variable functionality existed; it may be re-instated in future versions.

4. IMPLEMENTATION

The OverCode user interface depends on an analysis pipeline that canonicalizes solutions in a manner designed for human readability, referred to here as *cleaning*. The pipeline then creates stacks of solutions that have become identical through the cleaning process. The pipeline accepts, as input, a set of solutions, expressed as function definitions for $f(a, ...)$, and one test case $f(a_1, ...)$. We refer to the solutions that enter the pipeline as *raw*, and the solutions that exit the pipeline as *clean*. To illustrate this pipeline, we will have a few running examples, beginning with `iterPower`.

4.1. Analysis Pipeline

OverCode is currently implemented for Python, but the pipeline steps described below could be readily generalized to other languages commonly used to teach programming.

1. Reformat solutions. For a consistent appearance, the solutions are reformatted[2] to have consistent line indentation and token spacing. Comments and empty lines are also removed. These steps not only make solutions more readable, but also allow exact string matches between solutions, after additional cleaning steps later in the pipeline. Although comments can contain valuable information, the variation in comments is so great that clustering and summarizing them will require significant additional design, which remains future work.

The following example illustrates the effect of this reformatting:

Student A Raw Code

```
def iterPower(base, exp):
    '''
    base: int or float.
    exp: int >= 0
    returns: int or float, base^exp
    '''
    result = 1
    for i in range(exp):
        result *= base
    return result
```

Student A Reformatted

```
def iterPower(base,exp):
    result=1
    for i in range(exp):
        result*=base
    return result
```

2. Execute solutions. Each solution is executed once, using the same test case. During each step of the execution, the names and values of local and global variables, and also return values from functions, are recorded as a program trace. There is one program trace per solution. For the purposes of illustrating this pipeline, we will use the example of executing definitions of `iterPower` on a

[2]We used the PythonTidy package, by Charles Curtis Rhode. **https://pypi.python.org/pypi/PythonTidy/** Since our datasets are in Python, we use a Python-specific reformatting script. However, our approach is not language-specific.

base of 5.0 and an exp of 3.

Student Code with Test Case

```
def iterPower(base,exp):
    #student code here
iterPower(5.0, 3)
```

3. Extract variable sequences. During the previous step, the Python execution logger [Guo 2013] records the values of all in-scope variables after every statement execution in the Python program. The resulting log is referred to as the program trace. For each variable in a program trace, we extract the sequence of values it takes on, without considering how many statements were executed before the variable's value changed.

Student A Code with Test Case

```
def iterPower(base,exp):
    result=1
    for i in range(exp):
        result*=base
    return result
iterPower(5.0, 3)
```

Program Trace for Student A Code

```
iterPower(5.0, 3)
 base : 5.0, exp : 3
result=1
 base : 5.0, exp : 3, result : 1
for i in range(exp):
 base : 5.0, exp : 3, result : 1, i : 0
result*=base
 base : 5.0, exp : 3, result : 5.0, i : 0
for i in range(exp):
 base : 5.0, exp : 3, result : 5.0, i : 1
result*=base
 base : 5.0, exp : 3, result : 25.0, i : 1
for i in range(exp):
 base : 5.0, exp : 3, result : 25.0, i : 2
result*=base
 base : 5.0, exp : 3, result : 125.0, i : 2
return result
 value returned: 125.0
```

Variable Sequences for Student A Code

```
base: 5.0
exp: 3
result: 1, 5.0, 25.0, 125.0
i: 0, 1, 2
```

Variable sequence extraction also works for purely functional programs, in which variables are never reassigned, because each recursive invocation is treated as if new values are given to its parameter variables. For example, in spite of the fact that the iterPower problem asked students to compute the exponential $\texttt{base}^{\texttt{exp}}$ *iteratively*, 60 of the 3842 iterPower solutions in the dataset were in fact recursive. One of these recursive examples is shown below, along with the variable sequences observed for the recursive function's parameters.

Recursive Example

```
def iterPower(base,exp):
   if exp==0:
       return 1
   else:
       return
           base*iterPower(base,exp-1)
iterPower(5.0, 3)
```

Program Trace for Recursive Example

```
iterPower(5.0, 3)
 base : 5.0, exp : 3
if exp==0:
 base : 5.0, exp : 3
return base*iterPower(base,exp-1)
 base : 5.0, exp : 3
iterPower(5.0, 2)
 base : 5.0, exp : 2
if exp==0:
 base : 5.0, exp : 2
return base*iterPower(base,exp-1)
 base : 5.0, exp : 2
iterPower(5.0, 1)
 base : 5.0, exp : 1
if exp==0:
 base : 5.0, exp : 1
return base*iterPower(base,exp-1)
 base : 5.0, exp : 1
iterPower(5.0, 0)
 base : 5.0, exp : 0
if exp==0:
 base : 5.0, exp : 0
return 1
 base : 5.0, exp : 0
return base*1
 base : 5.0, exp : 1
return base*5
 base : 5.0, exp : 2
return base*25
 base : 5.0, exp : 3
value returned: 125.0
```

Variable Sequences for Recursive Example

```
base: 5.0
exp: 3, 2, 1, 0, 1, 2, 3
```

4. Identify common variables. We analyze all program traces, identifying which variables' sequences are identical. We define a *common variable* to denote those variables that have identical sequences across two or more program traces. Variables which occur in only one program trace are called *unique variables*.

Student B Code with Test Case

```
def iterPower(base,exp):
    r=1
    for k in xrange(exp):
        r=r*base
    return r
iterPower(5.0, 3)
```

Variable Sequences for Student B Code

```
base: 5.0
exp: 3
r: 1, 5.0, 25.0, 125.0
k: 0, 1, 2
```

Student C Code with Test Case

```
def iterPower(base,exp):
   result=1
   while exp>0:
       result*=base
       exp-=1
   return result
iterPower(5.0, 3)
```

Variable Sequences for Student C Code

```
base : 5.0
exp: 3
result : 1, 5.0, 25.0, 125.0
exp : 3, 2, 1, 0
```

For example, in Student A's code and Student B's code, `i` and `k` take on the same sequence of values: 0,1,2. They are therefore considered the same *common variable*.

Common Variables
(across Students A, B, and C)

— `5.0`:
`base` (Students A, B, C)
— `3`:
`exp` (Students A, B)
— `1, 5.0, 25.0, 125.0`:
`result` (Students A, C)
`r` (Student B)
— `0,1,2`:
`i` (Student A)
`k` (Student B)

Unique Variables
(across Students A, B, and C)

— `3,2,1,0`:
`exp` (Student C)

5. Rename common and unique variables. A common variable may have a different name in each program trace. The name given to each common variable is the variable name that is given most often to that common variable across all program traces.

There are exceptions made to avoid three types of name collisions described in Section 4.2 that follows. In the running example, the unique variable's original name, `exp`, has a double underscore appended to it as a modifier to resolve a name collision with the common variable of the same name, referred to here as a Unique/Common Collision.

Common Variables, Named

— `base: 5.0`
— `exp: 3`
— `result: 1, 5.0, 25.0, 125.0`
— `i: 0,1,2` (common name tie broken by random choice)

Unique Variables, Named

— `exp__: 3,2,1,0`

After common and unique variables in the solutions are renamed, the solutions are now called *clean*.

Clean Student A Code (After Renaming)

```python
def iterPower(base,exp):
    result=1
    for i in range(exp):
        result*=base
    return result
```

Clean Student B Code (After Renaming)

```python
def iterPower(base,exp):
    result=1
    for i in xrange(exp):
        result=result*base
    return result
```

Clean Student C Code (After Renaming)

```python
def iterPower(base,exp__):
   result=1
   while exp__>0:
       result*=base
       exp__-=1
   return result
```

6. Make stacks. We iterate through the clean solutions, making stacks of solutions that share an identical *set* of lines of code. We compare *sets* of lines of code because then solutions with arbitrarily ordered lines that do not depend on each other can still fall into the same stack. (Recall that the variables in these lines of code have already been renamed based on their dynamic behavior, and all the solutions have already been marked input-output correct by an autograder, prior to this pipeline.) The solution that represents the stack is randomly chosen from within the stack, because all the clean solutions within the stack are identical, with the possible exception of the order of their statements.

In the examples below, the clean C and D solutions have the exact same set of lines, and both provide correct output, with respect to the autograder. Therefore, we assume that the difference in order of the statements between the two solutions does not need to be communicated to the user. The two solutions are put in the same stack, with one solution arbitrarily chosen as the visible cleaned code. However, since Student A and Student B use different functions, i.e., `xrange` vs. `range`, and different operators, i.e., `*=` vs. `=`, `*`, the pipeline puts them in separate stacks.

Stack 1 Clean Student A (After Renaming)

```
def iterPower(base,exp):
    result=1
    for i in range(exp):
        result*=base
    return result
```

Stack 2 Clean Student B (After Renaming)

```
def iterPower(base,exp):
    result=1
    for i in xrange(exp):
        result=result*base
    return result
```

Stack 3 Clean Student C (After Renaming)

```
def iterPower(base,exp__):
    result=1
    while exp__>0:
        result=result*base
        exp__-=1
    return result
```

Stack 3 Clean Student D (After Renaming)

```
def iterPower(base,exp__):
    result=1
    while exp__>0:
        exp__-=1
        result=result*base
    return result
```

Even though all the solutions we process in this pipeline have already been marked correct by an autograder, the program tracing [Guo 2013] and renaming scripts occasionally generate errors while processing a solution. For example, the script may not have code to handle a particular but rare Python construct. Errors thrown by the scripts drive their development and are helpful for debugging. When errors occur while processing a particular solution, we exclude the solution from our analysis. Less than five percent of the solutions in each of our three problem datasets are excluded.

4.2. Variable Renaming Details and Limitations

There are three distinct types of name collisions possible when renaming variables to be consistent across multiple solutions. The first, which we refer to as a *common/common* collision, occurs when two common variables (with different variable sequences) have the same common name. The second, referred to here as a *multiple instances* collision, occurs when there are multiple different instances of the same common variable in a solution. The third and final collision, referred to as a *unique/common* collision, occurs when a unique variable's name collides with a common variable's name.

Common/common collision. If common variables cv_1 and cv_2 are both most frequently named i across all program traces, we append a modifier to the name of the less frequently used common variable. For example, if 500 program traces have an instance of cv_1 and only 250 program traces have an instance of cv_2, cv_1 will be named i and cv_2 will be named iB.

This is illustrated below. Across all thousand of `iterPower` definitions in our dataset, a subset of them created a variable that iterated through the values generated by `range(exp)`. Student A's code is an example. A smaller subset created a variable that iterated through the values generated by `range(1,exp+1)`, as seen in Student E's code. These are two separate common variables in our pipeline, due to their differing value sequences. The *common/common* name collision arises because both common variables are most frequently named `i` across all solutions to `iterPower`. To preserve the one-to-one mapping of variable name to value sequence across the entire `iterPower` problem dataset, the pipeline appends a modifier, `B`, to the common variable `i` found in fewer `iterPower` solutions. A common variable, also most commonly named `i`, which is found in even fewer `iterPower` definitions, will have a `C` appended, etc.

Student A (Represents 500 Solutions

```
def iterPower(base,exp):
    result=1
    for i in range(exp):
        result*=base
    return result
```

Clean Student A (After Renaming)

```
(unchanged)
```

Student E (Represents 250 Solutions

```
def iterPower(base,exp):
   result=1
   for i in range(1,exp+1):
       result*=base
   return result
```

Clean Student E (After Renaming)

```
def iterPower(base,exp):
   result=1
   for iB in range(1,exp+1):
       result*=base
   return result
```

Multiple-instances collision. We identify variables by their sequence of values (excluding consecutive duplicates), not by their given name in any particular solution. However, without considering the timing of variables' transitions between values, relative to other variables in scope at each step of a function execution, it is not possible to differentiate between multiple instances of a common variable within a single solution.

Rather than injecting a name collision into an otherwise correct solution, we chose to preserve the author's variable name choice for all the instances of that common variable in that solution. If an author's preserved variable name collides with any common variable name in any program trace and does not share that common variable's sequence of values, the pipeline appends a double underscore to the authors preserved variable name, so that the interface, and the human reader, do not conflate them.

In the following example, the solution's author made a copy of the `exp` variable, called it `exp1`, and modified neither. Both map to the same common variable, `expB`. Therefore, both have had their author-given names preserved, with an underscore appended to the local `exp` so it does not look like common variable `exp`.

Code with Multiple Instances of a Common Variable

```
def iterPower(base,exp):
   result=1
   exp1=abs(exp)
   for i in xrange(exp1):
       result*=base
   if exp<0:
       return 1.0/float(result)
   return result
iterPower(5.0,3)
```

Code with Multiple Instances Collision Resolved

```
def iterPower(base,exp__):
   result=1
   exp1=abs(exp__)
   for i in xrange(exp1):
       result*=base
   if exp__<0:
       return 1.0/float(result)
   return result
iterPower(5.0,3)
```

Common Variable Mappings

```
Both exp and exp1 map to common
    variable expB: 3
exp: 3
exp1: 3

All other variables map to common
    variables of same name
base: 5.0
i: 0, 1, 2
result: 1, 5.0, 25.0, 125.0
```

Unique/common collision. Unique variables, as defined before, take on a sequence of values that is unique across all program traces. If a unique variable's name collides with any common variable name in any program trace, the pipeline appends a double underscore to the unique variable name, so that the interface, and the human reader, do not conflate them.

In addition to the example of this collision in the description of common and variable naming in the previous section, we provide the example below. In this solution, the student added 1 to the exponent variable before entering a `while` loop. No other students did this. To indicate that the `exp` variable is *unique* and does not share the same behavior as the common variable also named `exp`, our pipeline appends double underscores to `exp` in this one solution.

```
def iterPower(base,exp__):
    result=1
    exp__+=1
    while exp__>1:
        result*=base
        exp__-=1
    return result
```

4.3. Complexity of the Analysis Pipeline

Unlike previous pairwise AST edit distance-based clustering approaches that have quadratic complexity both in the number of solutions and the size of the ASTs [Huang et al. 2013], our analysis pipeline has linear complexity in the number of solutions and in the size of the ASTs. The Reformat step performs a single pass over each solution for removing extra spaces, comments, and empty lines. Since we only consider correct solutions, we assume that each solution can be executed within a constant time that is independent of the number of solutions. The executions performed by the autograder for checking correctness could also be instrumented to obtain the program traces, so code is not unnecessarily re-executed. The identification of all common variables and unique variables across the program traces takes linear time as we can hash the corresponding variable sequences and then check for occurrences of identical sequences. The Renaming step, which includes handling name collisions, also performs a single pass over each solution. Finally, the Stacking step creates stacks of similar solutions by performing set-based equality of lines of code that can also be performed in linear time by hashing the set of lines of code.

5. DATASET

For evaluating both the analysis pipeline and the user interface of OverCode, we use a dataset of solutions from 6.00x, an introductory programming course in Python that was offered on edX in fall 2012. We chose Python solutions from three exercise problems, and this dataset consists of student solutions submitted within two weeks of the posting of the those three problems. We obtained thousands of submissions to these problems, from which we selected all correct solutions (tested over a set of test cases) for our analysis. The number of solutions analyzed for each problem is shown in Figure 7.

— **`iterPower`** The `iterPower` problem asks students to write a function to compute the exponential $\texttt{base}^{\texttt{exp}}$ iteratively using successive multiplications. This was an in-lecture exercise for the lecture on teaching iteration. See Figure 8 for examples.
— **`hangman`** The `hangman` problem takes a string `secretWord` and a list of characters `lettersGuessed` as input, and asks students to write a function that returns a string where all letters in `secretWord` that are not present in the list `lettersGuessed` are replaced with an underscore. This was a part of the third week of problem set exercises. See Figure 9 for examples.
— **`compDeriv`** The `compDeriv` problem requires students to write a Python function to compute the derivative of a polynomial, where the coefficients of the polynomial are represented as a python list. This was also a part of the third week of problem set exercises. See Figure 10 for examples.

We chose these three exercises for our dataset because they are representative of the typical exercises students solve in the early weeks of an introductory programming course. The three exercises

Problem Description	Total Submissions	Correct Solutions
iterPower	8940	3875
hangman	1746	1118
compDeriv	3013	1433

Fig. 7. Number of solutions for the three problems in our 6.00x dataset.

IterPower Examples

```
def iterPower(base, exp):
    result=1
    i=0
    while i < exp:
        result *= base
        i += 1
    return result
```

```
def iterPower(base, exp):
    t=1
    for i in range(exp):
        t=t*base
    return t
```

```
def iterPower(base, exp):
    x = base
    if exp == 0:
        return 1
    else:
        while exp >1:
            x *= base
            exp -=1
        return x
```

```
def iterPower(base, exp):
    x = 1
    for n in [base] * exp:
        x *= n
    return x
```

Fig. 8. Example solutions for the `iterPower` problem in our 6.00x dataset.

have varying levels of complexity and ask students to perform loop computation over three fundamental Python data types, integers (`iterPower`), strings (`hangman`), and lists (`compDeriv`). The exercises span the second and third weeks of the course in which they were assigned.

6. OVERCODE ANALYSIS PIPELINE EVALUATION

We now present the evaluation of OverCode's analysis pipeline implementation on our Python dataset. We first present the running time of our algorithm and show that it can generate stacks within few minutes for each problem on a laptop. We then present the distribution of initial stack sizes generated by the pipeline. Finally, we present some examples of the common variables identified by the pipeline and report on the number of cases where name collisions are handled during the cleaning process. The evaluation was performed on a Macbook Pro 2.6GHz Intel Core i7 with 16GB of RAM.

Running Time The complexity of the pipeline that generates stacks of solutions grows linearly in the number of solutions as described in Section 4.3. Figure 11 reports the running time of the pipeline on the problems in the dataset as well as the number of stacks and the number of common variables found across each of the problems. As can be seen from the Figure, the pipeline is able to clean thousands of student solutions and generate stacks within few minutes for each problem.

Distribution of Stacks The distribution of initial stack sizes generated by the analysis pipeline for different problems is shown in Figure 12. Note that the two axes of the graph corresponding to the size and the number of stacks are shown on a logarithmic scale. For each problem, we observe that there are a few large stacks and a lot of smaller stacks (particularly of size 1). The largest stack for `iterPower` problem consists of 1534 solutions, while the largest stacks for `hangman` and `compDeriv` consists of 97 and 22 solutions respectively. The two largest stacks with the corresponding cleaned solutions for each problem are shown in Figure 13.

The number of stacks consisting of a single solution for `iterPower`, `hangman`, and `compDeriv` are 684, 452, and 959 respectively. Some singleton stacks are the same as one of

Hangman Examples

```
def getGuessedWord(secretWord, lettersGuessed):
    guessedWord = ''
    guessed = False
    for e in secretWord:
        for idx in range(0,len(lettersGuessed)):
            if (e == lettersGuessed[idx]):
                guessed = True
                break
        # guessed = isWordGuessed(e, lettersGuessed)
        if (guessed == True):
            guessedWord = guessedWord + e
        else:
            guessedWord = guessedWord + '_ '
        guessed = False
    return guessedWord
```

```
def getGuessedWord(secretWord, lettersGuessed):
    if len(secretWord) == 0:
        return ''
    else:
        if lettersGuessed.count(secretWord[0]) > 0:
            return secretWord[0] + ' ' + getGuessedWord(secretWord[1:],
                lettersGuessed)
        else:
            return '_ ' + getGuessedWord(secretWord[1:], lettersGuessed)
```

Fig. 9. Example solutions for the `hangman` problem in our 6.00x dataset.

CompDeriv Examples

```
def computeDeriv(poly):
    der=[]
    for i in range(len(poly)):
        if i>0:
            der.append(float(poly[i]*i))
    if len(der)==0:
        der.append(0.0)
    return der
```

```
def computeDeriv(poly):
    if len(poly) == 1:
        return [0.0]
    fp = poly[1:]
    b = 1
    for a in poly[1:]:
        fp[b-1] = 1.0*a*b
        b += 1
    return fp
```

```
def computeDeriv(poly):
    if len(poly) < 2:
        return [0.0]
    poly.pop(0)
    for power, value in
        enumerate(poly):
        poly[power] = value *
            (power + 1)
    return poly
```

```
def computeDeriv(poly):
    index = 1
    polyLen = len(poly)
    result = []
    while (index < polyLen):
        result.append(float(poly[index]*index))
        index += 1
    if (len(result) == 0):
        result = [0.0]
    return result
```

Fig. 10. Example solutions for the `compDeriv` problem in our 6.00x dataset.

Problem	Correct Solutions	Running Time	Initial Stacks	Common Variables
`iterPower`	3875	15m 28s	862	38
`hangman`	1118	8m 6s	552	106
`compDeriv`	1433	10m 20s	1109	50

Fig. 11. Running time, and the number of stacks and common variables generated by the OverCode backend implementation on our dataset problems.

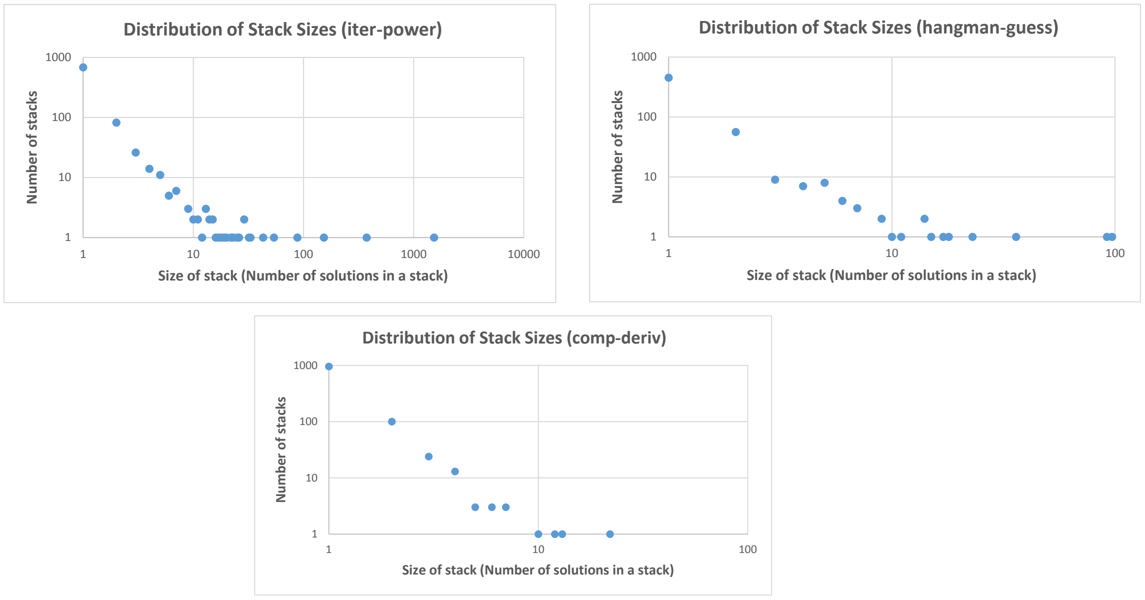


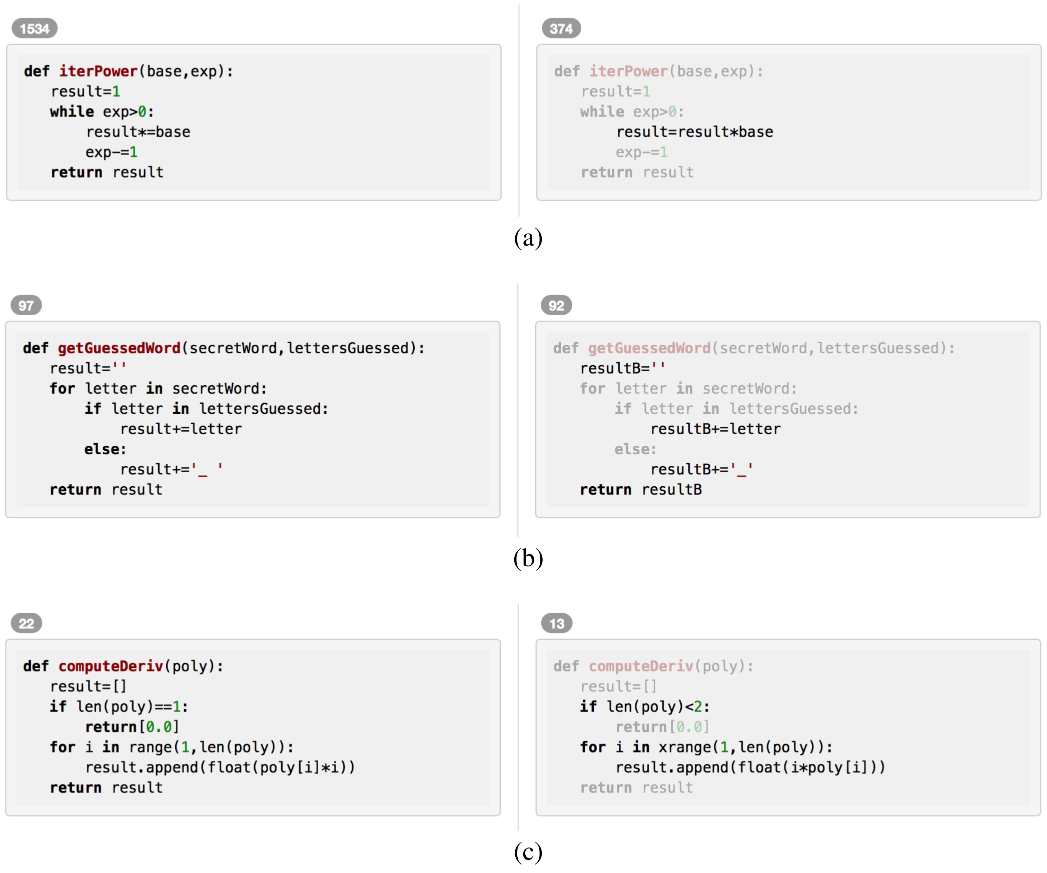
Fig. 12. The distribution of sizes of the initial stacks generated by our algorithm for each problem. We can observe a long tail distribution with a few large stacks and a lot of small stacks. Note that the two axis corresponding to the size of stacks and the number of stacks are in logarithmic scale.

the largest stacks, except for a unique choice, such as initializing a variable using several additional significant digits than necessary: `result=1.000` instead of `result=1` or `result=1.0`. Other singleton stacks have convoluted control flow that no other student used.

These variations are compounded by inclusion of unnecessary statements that do not affect input-output behavior. An existing stack may have all the same lines of code except for the unnecessary line(s), which cause the solution to instead be a singleton. These unnecessary lines may reveal misconceptions, and therefore are potentially interesting to teachers. In future versions, rewrite rules may be expanded to allow include line removal rules, so that teachers can remove inconsequential extra lines and cause singleton(s) to merge with other stacks.

The tail of singleton solutions is long, and cannot be read in its entirety by teachers. Even so, the user studies indicate that teachers still extracted significant value from OverCode presentation of solutions. It may be that the largest stacks are the primary sources of information, and singletons can be ignored without a significant effect on the value teachers get from OverCode. Future work will explore ways to suggest rewrite and removal rules that maximally collapse stacks.

Common Variables There exists a large variation among the variable names used by students to denote variables that compute the same set of values. The Variable Renaming step of the analysis renames these equivalent variables with the most frequently chosen variable name so that a teacher can easily recognize the role of variables in a given solution. The number of common variables found by the pipeline on the dataset problems is shown in Figure 11 and some examples of these



```
def iterPower(base,exp):
    result=1
    while exp>0:
        result*=base
        exp-=1
    return result
```

```
def iterPower(base,exp):
    result=1
    while exp>0:
        result=result*base
        exp-=1
    return result
```

```
def getGuessedWord(secretWord,lettersGuessed):
    result=''
    for letter in secretWord:
        if letter in lettersGuessed:
            result+=letter
        else:
            result+='_ '
    return result
```

```
def getGuessedWord(secretWord,lettersGuessed):
    resultB=''
    for letter in secretWord:
        if letter in lettersGuessed:
            resultB+=letter
        else:
            resultB+='_'
    return resultB
```

```
def computeDeriv(poly):
    result=[]
    if len(poly)==1:
        return[0.0]
    for i in range(1,len(poly)):
        result.append(float(poly[i]*i))
    return result
```

```
def computeDeriv(poly):
    result=[]
    if len(poly)<2:
        return[0.0]
    for i in xrange(1,len(poly)):
        result.append(float(i*poly[i]))
    return result
```

Fig. 13. The two largest stacks generated by the OverCode backend algorithm for the (a) `iterPower`, (b) `hangman`, and (c) `compDeriv` problems.

common variable names are shown in Figure 14. Figure 14 also presents the number of times such a variable occurs across the solutions of a given problem, the corresponding variable sequence value on a given test input, and a subset of the original variable names used in the student solutions.

Collisions in Variable Renaming The number of Common/Common, Multiple Instances, and Unique/Common collisions discovered and resolved while performing variable renaming is shown in Figure 15. A large majority of the collisions were Common/Common Collisions. For example, Figure 14 shows the common variable name `exp` for two different sequence values $[3, 2, 1, 0]$ and $[3]$ for the `iterPower` problem. Similarly, the common variable name `i` corresponds to sequences $[-13.9, 0.0, 17.5, 3.0, 1.0$ and $[0, 1, 2, 3, 4, 5]$ for the `compDeriv` problem. There were also a few Multiple Instances collisions and Unique/Common collisions found: 1.5% for `iterPower`, 3% for `compDeriv`, and 10% for `hangman`.

7. USER STUDY 1: WRITING A CLASS FORUM POST

Our goal was to design a system that allows teachers to develop a better understanding of the variation in student solutions, and give feedback that is relevant to more students' solutions. We designed two user studies to evaluate our progress in two ways: (1) user interface satisfaction and (2) how many solutions teachers could read and produce feedback on in a fixed amount of time. Reading and providing feedback to thousands of submissions is an unrealistically difficult task for our con-

Common Variable Name	Occurrence Count	Sequence Value	Original Variable Names
iterPower			
`result`	3081	[1, 5.0, 25.0, 125.0]	`result`, `wynik`, `out`, `total`, `ans`, `acum`, `num`, `mult`, `output`, ···
`exp`	2744	[3, 2, 1, 0]	`exp`, `iterator`, `app`, `ii`, `num`, `iterations`, `times`, `ctr`, `b`, `i`, ···
`exp`	749	[3]	`exp`, `count`, `temp`, `exp3`, `exp2`, `exp1`, `inexp`, `old_exp`, `expr`, `x`, ···
`i`	266	[0, 1, 2]	`i`, `a`, `count`, `c`, `b`, `iterValue`, `iter`, `n`, `y`, `inc`, `x`,`times`, ···
hangman			
`letter`	817	['t', 'i', 'g', 'e', 'r']	`letter`, `char`, `item`, `i`, `letS`, `ch`, `c`, `lett`, ···
`result`	291	[' ', '_', '_i', '_i_', '_i_e', '_i_e_']	`result`, `guessedWord`, `ans`, `str1`, `anss`, `guessed`, `string`, ···
`i`	185	[0, 1, 2, 3, 4]	`i`, `x`, `each`, `b`, `n`, `counter`, `idx`, `pos` ···
`found`	76	[0, 1, 0, 1, 0]	`found`, `n`, `letterGuessed`, `contains`, `k`, `checker`, `test`, ···
compDeriv			
`result`	1186	[[0.0, 35.0, 9.0, 4.0]]	`result`, `output`, `poly_deriv`, `res`, `deriv`, `resultPoly`, ···
`i`	284	[-13.39, 0.0, 17.5, 3.0, 1.0]	`i`, `each`, `a`, `elem`, `number`, `value`, `num`, ···
`i`	261	[0, 1, 2, 3, 4, 5]	`i`, `power`, `index`, `cm`, `x`, `count`, `pwr`, `counter`, ···
`length`	104	[5]	`length`, `nmax`, `polyLen`, `lpoly`, `lenpoly`, `z`, `l`, `n`, ···

Fig. 14. Some examples of common variables found by our analysis across the problems in the dataset. The table also shows the frequency of occurrence of these variables, the common sequence values of these variables on a given test case, and a subset of the original variable names used by students.

Problem	Correct Solutions	Common/Common Collisions	Multiple Instances Collisions	Unique/Common Collisions
`iterPower`	3875	1298	25	32
`hangman`	1118	672	62	49
`compDeriv`	1433	928	21	23

Fig. 15. The number of common/common, multiple instances, and unique/common collisions discovered by our algorithm while renaming the variables to common names.

trol condition, so instead of measuring time to finish the entire set of solutions, we sought to measure what our subjects could accomplish in a fixed amount of time (15 minutes).

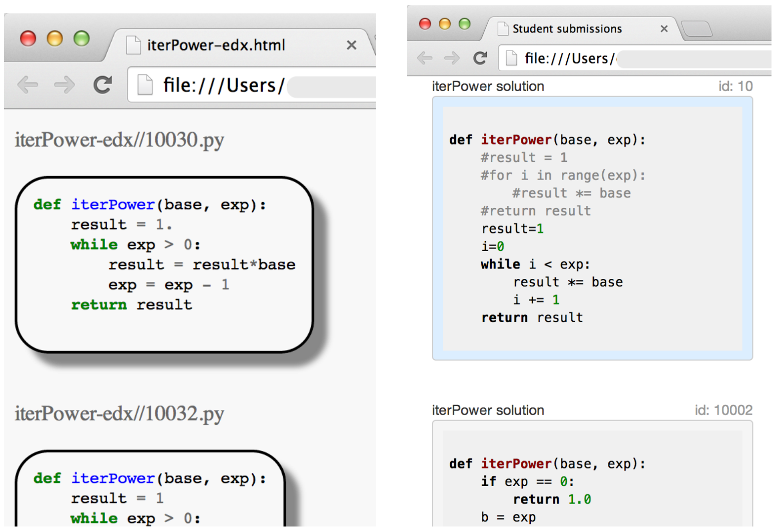
The first study was a 12-person within-subjects evaluation of interface satisfaction when using OverCode for a realistic, relatively unstructured task. Using either OverCode or a baseline interface, subjects were asked to browse student solutions to the programming problems in our dataset and then write a class forum post on the good and bad ways students solved the problem. Through this study, we sought to test our first hypothesis:

— **H1 Interface Satisfaction**: Subjects will find OverCode easier to use, more helpful and less overwhelming for browsing thousands of solutions, compared to the baseline.

7.1. OverCode and Baseline Interfaces

We designed two interfaces, referred to here as OverCode and the baseline. The OverCode interface and backend are described in detail in Section 3. The baseline interface was a single webpage with all student solutions concatenated in a random order into a flat list (Figure 16, left). We chose this design to emulate existing methods of reviewing solutions, and to draw out differences between browsing stacked and unstacked solutions. This is analogous to the "flat" interface chosen as a baseline for Basu et al.'s interface for grading clusters of short answers [Brooks et al. 2014]. Basu et al.'s assumption, that existing options for reviewing solutions are limited to going through solutions one-by-one, is backed by our pilot studies and interviews with teaching staff, and our own grading experiences. In fact, in edX programming MOOCs, teachers are not even provided with an interface for viewing all solutions at once; they can only look at one student's solution at a time. If the solutions can be downloaded locally, some teachers may use within-file search functions like the command line utility `grep`. Our baseline allows that too, through the in-browser find command.

In the baseline, solutions appeared visually similar to those in the OverCode interface (boxed, syntax-highlighted code), but the solutions were *raw*, in the sense that they were not normalized for whitespace or variable naming differences. As in the OverCode condition, subjects were able to use standard web-browser features, such as the within-page *find* action.



```
def iterPower(base, exp):
    result = 1.
    while exp > 0:
        result = result*base
        exp = exp - 1
    return result
```

```
def iterPower(base, exp):
    result = 1
    while exp > 0:
```

```
def iterPower(base, exp):
    #result = 1
    #for i in range(exp):
        #result *= base
    #return result
    result=1
    i=0
    while i < exp:
        result *= base
        i += 1
    return result
```

```
def iterPower(base, exp):
    if exp == 0:
        return 1.0
    b = exp
```

Fig. 16. The baseline interface used in the Forum Post study (left) and the Coverage study (right).

7.2. Participants

We recruited participants by reaching out to past and present programming course staff, and advertising on an academic computer science research lab's email list. These individuals were qualified to participate in our study because they met at least one of the following requirements: (1) were current teaching staff of a computer science course (2) had graded Python code before, or (3) had significant Python programming experience, making them potential future teaching staff.

Information about the subjects' backgrounds was collected during recruitment and again at the beginning of their one hour in-lab user study session. 12 people (7 male) participated, with a mean age of 23.5 ($\sigma = 3.8$). Subjects had a mean 4 years of Python programming experience ($\sigma = 1.8$), and 75% of participants had graded student solutions written in Python before. Half of the participants were graduate students, and other half were undergraduates.

7.3. Apparatus

Each subject was given $20 to participate in a 60 minute session with an experimenter, in an on-campus academic lab conference room. They used laptops running MacOS and Linux with screen sizes ranging from 12.5 to 15.6 inches, and viewed the OverCode and baseline interfaces in either Safari or Chrome. Data was recorded with Google Docs and Google Forms filled out by participants.

7.4. Conditions

Subjects performed the main task of browsing solutions and writing a class forum post twice, once in each interface condition, focusing on one of the three problems in our dataset (Section 5) each time. For each participant, the third remaining problem was used during training, to reduce learning effects when performing the two main tasks. The pairing and ordering of interface and problem conditions were fully counterbalanced, resulting in 12 total conditions. The twelve participants were randomly assigned to one of the 12 conditions, such that all conditions were tested.

7.5. Procedure

7.5.1. Prompt. The experimenter began by reading the following prompt, to give the participant context for the tasks they would be performing:

> We want to help TAs [teaching assistants] give feedback to students in programming classes at scale. For each of three problems, we have a large set of students' submissions (> 1000).
> All the submissions are correct, in terms of input and output behavior. We're going to ask you to browse the submissions and produce feedback for students in the class. You'll do this primarily in the form of a class forum post.

To make the task more concrete, participants were given an example[3] of a class forum post that used examples taken from student solutions to explain different strategies for solving a Python problem. They were also given print-outs of the prompts for each of the three problems in our dataset, to reference when looking at solutions.

7.5.2. Training. Given the subjects' extensive experience with web-browsers, training for the baseline interface was minimal. Prior to using the OverCode interface, subjects watched a 3-4 minute long training video demonstrating the features of OverCode, and were given an opportunity to become familiar with the interface and ask questions. The training session focused on the problem that would not be used in the main tasks, in order to avoid learning effects.

7.5.3. Tasks. Subjects then performed the main tasks twice, once in each interface, focusing on a different programming problem each time. They were given a fixed amount of time to both read solutions and provide feedback, so we did not measure task completion times, but instead the quality of their experience in providing feedback to students at scale.

— *Feedback for Students* (15 minutes) Subjects were asked to write a class forum post on the good and bad ways students solved the problem. The fifteen minute period included both browsing and writing time, as subjects were free to paste in code examples and write comments as they browsed the solutions.
— *Autograder Bugs* (2 min) Although the datasets of student solutions were marked as correct by an autograder, there may be holes in the autograder's test cases. Some solutions may deviate from the problem prompt, and therefore be considered incorrect by teachers, e.g., recursive solutions to `iterPower` when its prompt explicitly calls for an iterative solution. As a secondary task, we asked subjects to write down any bugs in the autograder they came across while browsing solutions. This was often performed concurrently with the primary task by the subject.

7.5.4. Surveys. Subjects filled out a post-interface condition survey about their experience using the interface. This was a short-answer survey, where they wrote about what they liked, what they did not like, and what they would like to see in a future version of the interface. At the end of the study, subjects rated agreement (on a 7-point Likert scale) with statements about their satisfaction with each interface.

7.6. Results

H1 is supported by ratings from the post-study survey (see Figure 17). Statistical significance was computed using the Wilcoxon Signed Rank test, pairing users' ratings of each interface. After using both interfaces to view thousands of solutions, subjects found OverCode easier to use (W=52, Z=2.41, $p<0.05$, r=0.70) and less overwhelming (W = 0, Z = -2.86, $p < 0.005$, r = 0.83) than the baseline interface. Finally, participants felt that OverCode "helped me get a better sense of my students' understanding" than the baseline did (W = 66, Z = 3.04, $p < 0.001$, r = 0.88).

[3]Our example was drawn from the blog "Practice Python: 30-minute weekly Python exercises for beginners," posted on Thursday, April 24, 2014, and titled "SOLUTION Exercise 11: Check Primality and Functions." (**`http://practicepython.blogspot.com`**)

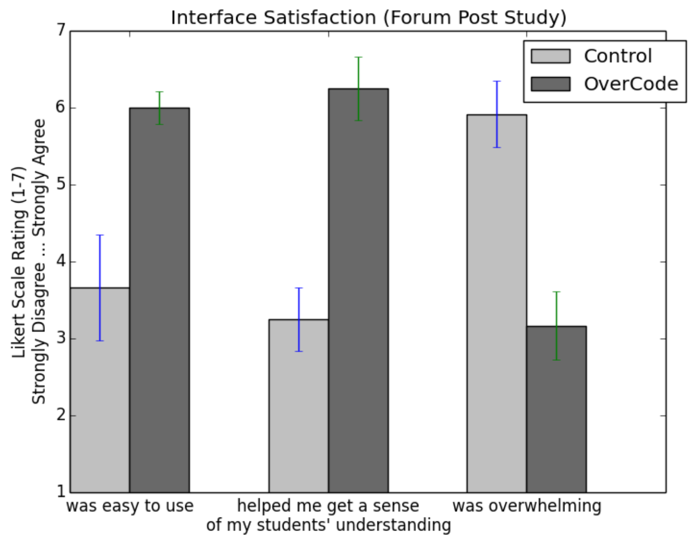


Fig. 17. **H1: Interface satisfaction** Mean Likert scale ratings (with standard error) for OverCode and baseline interfaces, after subjects used both to perform the forum post writing task.

From the surveys conducted after subjects completed each interface condition, we found that subjects found stacking and the ability to rewrite code to be useful and enjoyable features of OverCode:

— Stacking is an awesome feature. Rewrite tool is fantastic. Done feature is very rewarding–feels much less overwhelming. "Lines that appear in x submissions" also awesome.
— Really liked the clever approach for variable normalization. Also liked the fact that stacks showed numbers, so I'd know I'm focusing on the highest-impact submissions. Impressed by the rewrite ability... it cut down on work so much!
— I liked having solutions collapsed (not having to deal with variable names, etc), and having rules to allow me to collapse even further. This made it easy to see the "plurality" of solutions right away; I spent most of the time looking at the solutions that only had a few instances.

When asked for suggestions, participants gave many suggestions on stacks, filtering, and rewrite rules, such as:

— Enable the user to change the main stack that is being compared against the others.
— Suggest possible rewrite rules, based on what the user has already written, and will not affect the answers on the test case.
— Create a filter that shows all stacks that do *n*ot have a particular statement.

For both the OverCode and baseline interfaces, the feedback generated about `iterPower`, `hangman`, and `compDeriv` solutions fell into several common themes. One kind of feedback suggested new language features, such as using `*=` or the keyword `in`. Another theme identified inefficient, redundant, and convoluted control flow, such as repeated statements and unnecessary statements and variables. It was not always clear what the misconception was, though, as one participant wrote, *"The double iterator solution surely shows some lack of grasp of power of for loop, or range, or something."* Participants' feedback included comments on the relative goodness of different correct solutions in the dataset. This was a more holistic look at students' solutions as they varied along the dimensions of conciseness, clarity, and efficiency previously described.

Study participants found both noteworthy correct solutions and solutions they considered incorrect, despite passing the autograder. One participant learned a new Python function, `enumerate`, while looking at a solution that used it. The participant wrote, *"Cool, but uncalled for. I had to look it up :]. Use, but with comment."* Participants also found recursive `iterPower` and `hangman` solutions, which they found noteworthy. For what should have been an iterative `iterPower` function, the fact that this recursive solution was considered correct by the unit-test-based autograder was considered an autograder bug by some participants. Using the built-in Python exponentiation operator `**` was also considered correct by the autograder, even though it subverted the point of the assignment. It was also noted as an autograder bug by some participants who found it.

8. USER STUDY 2: COVERAGE

We designed a second 12-person study, similar in structure to the forum post study, but focused on measuring the coverage achieved by subjects in a fixed amount of time (15 minutes) when browsing and producing feedback on a large number of student solutions. The second study's task was more constrained than the first: instead of writing a freeform post, subjects were asked to identify the five most frequent strategies used by students and rate their confidence that these strategies occurred frequently in the student solutions. These changes to the task, as well as modifications to the OverCode and baseline interfaces, enabled us to measure coverage in terms of solutions read, the relevance of written feedback and the subject's perceived coverage. We sought to test the following hypotheses:

— **H2 Read coverage and speed** Subjects are able to read code that represents more student solutions at a higher rate using OverCode than with the baseline.
— **H3 Feedback coverage** Feedback produced when using OverCode is relevant to more students' solutions than when feedback is produced using the baseline.
— **H4 Perceived coverage** Subjects feel that they develop a better high-level view of students' understanding and misconceptions, and provide more relevant feedback using OverCode than with the baseline.

8.1. Participants, Apparatus, Conditions

The coverage study shared the same methods for recruiting participants, apparatus and conditions as the forum post study. 12 new participants (11 male) participated in the second study (mean age = 25.4, $\sigma = 6.9$). Across those 12 participants, the mean years of Python programming experience was 4.9 ($\sigma = 3.0$) and 9 of them had previously graded code (5 had graded Python code). There were 5 graduate students, 6 undergraduates, and 1 independent computer software professional.

8.2. Interface Modifications

Prior to the second study, both the OverCode and baseline interfaces were slightly modified (see differences in Figure 16) in order to enable measurements of read coverage, feedback coverage and perceived coverage.

— Clicking on stacks or solutions caused the box of code to be outlined in blue. This enabled the subject to mark them as *read*[4] and enabled us to measure read coverage.
— Stacks and solutions were all marked with an identifier, which subjects were asked to include with each piece of feedback they produced. This enabled us to more easily compute feedback coverage, which will be explained further in Section 8.4.
— All interface interactions were logged in the browser console, allowing us to track both the subject's read coverage over time, as well as their usage of other features, such as the creation of rewrite rules to merge stacks.
— Where it differed slightly before, we changed the styling of code in the baseline condition to exactly match the code in the OverCode condition.

[4]In the OverCode condition, this replaced the *done* checkboxes, in that clicking stacks caused the progress bar to update.

8.3. Procedure

8.3.1. Prompt. In the coverage study, the prompt was similar to the one used in the forum post study, explaining that the subjects would be tackling the problem of producing feedback for students at scale. The language was modified to shift the focus towards finding frequent strategies used by students, rather than any example of good or bad code used by a student.

8.3.2. Training. As before, subjects were shown a training video and given time to practice using OverCode's features prior to their trial in the OverCode condition.

8.3.3. Task. The coverage study's task consisted of a more constrained feedback task. Given 15 minutes with either the OverCode or baseline interface, subjects were asked to fill out a table, identifying the 5 most frequent strategies used by students to solve the problem. For each strategy they identified, they were asked to fill in the following fields in the table:

— A code example taken from the solution or stack.
— The identifier of the solution or stack.
— A short (1 sentence) annotation of what was good or bad about the strategy.
— Their confidence, on a scale of 1-7, that the strategy frequently occurred in the student solutions.

Importantly, subjects were also asked to mark solutions or stacks as *read* by clicking on them after they had 'processed' them, even if they weren't choosing them as representative strategies. Combined with interaction logging done by the system, this enabled us to measure read coverage.

8.3.4. Surveys. Although we measured interface satisfaction for a realistic task in the forum post study, we also measured interface satisfaction through surveys for this more constrained, coverage-focused task. Subjects filled out a post-interface condition survey in which they rated agreement (on a 7-point Likert scale) with positive and negative adjectives about their experience using the interface, and reflected on task difficulty. At the end of the study, subjects were asked to rate their agreement with statements about the usefulness of specific features of both the OverCode and baseline interfaces, and responded to the same interface satisfaction 7-point Likert scale statements used in the first study.

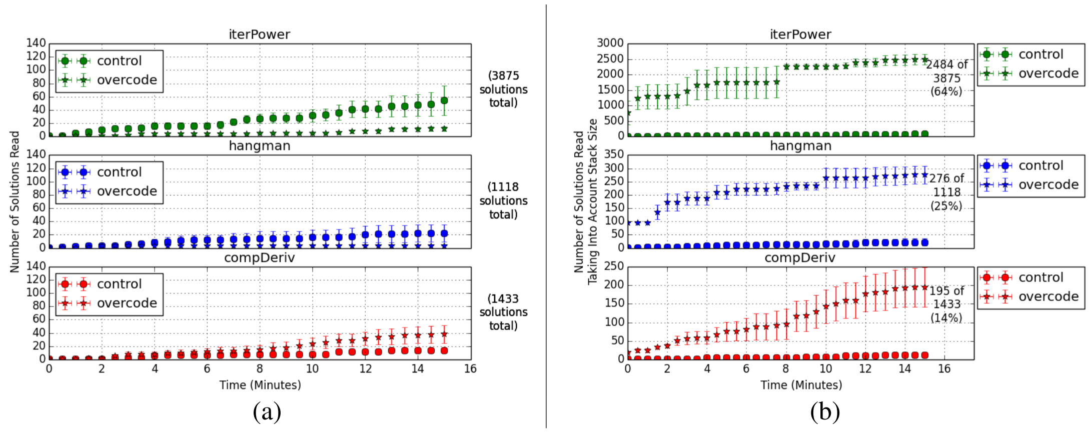


Fig. 18. In (a), we plot the mean number of cleaned solutions representing stacks read in OverCode over time versus the number of raw solutions read in the baseline interface over time while performing the 15 minute long Coverage Study task. In (b), we replace the mean number of cleaned solutions with the mean number of solutions (the size of the stacks) they represent. These are shown for each of the three problems in our dataset.

8.4. Results

8.4.1. H2: Read coverage and speed. This hypothesis is supported by our measurements of read coverage from this study. For each problem, subjects were able to view more cleaned and stacked solutions by the end of the 15 minute long main task using OverCode than raw solutions when using the baseline interface (Mann–Whitney U = 16, n1 = n2 = 4, $p<0.05$). Figure 18 shows the mean number of solutions read over time for each interface and each problem in our dataset. The curves show that subjects were able to read code that represented more raw solutions at a higher rate due to the stacking of similar solutions.

8.4.2. H3: Feedback coverage. Each subject reported on the 5 most frequent strategies in a set of solutions, by copying both a code example and the identifier of the solution (baseline) or stack (OverCode) that it came from. We define *feedback coverage* as the number of students for which the quoted code is relevant, in the sense that they wrote the same lines of code, ignoring differences in whitespace or variable names. We computed the coverage for each example using the following process:

— Reduce the quoted code down to only the lines referred to in the annotation. Often, the subject's annotation would focus on a specific feature of the quoted code, which sometimes had additional lines that were unrelated to the subject's feedback. For example, comments about iterating over a range function, while also quoting the contents of the for loop. This step meant we would be calculating the coverage of a more general (smaller) set of lines.
— Find the source stack that the quoted code comes from. This is trivial in the OverCode condition, where the stack ID is included in the subject's post. In the baseline, we used the solution ID included in the subject's post to find the stack that it was merged into by the backend pipeline.
— Find the cleaned version of each quoted line. The quoted lines of code may be raw code if they come from the baseline condition. By comparing the quoted code with the cleaned code of its source stack, we found the cleaned version of each line, with variable names and whitespace normalized.
— Find the raw solutions that include the set of cleaned lines, using a map from stacks to raw solutions provided by the backend pipeline.

Figure 19 shows the mean coverage of a set of feedback produced by a single subject, across problems and interface conditions. The feedback coverage is shown as the mean percentage of raw

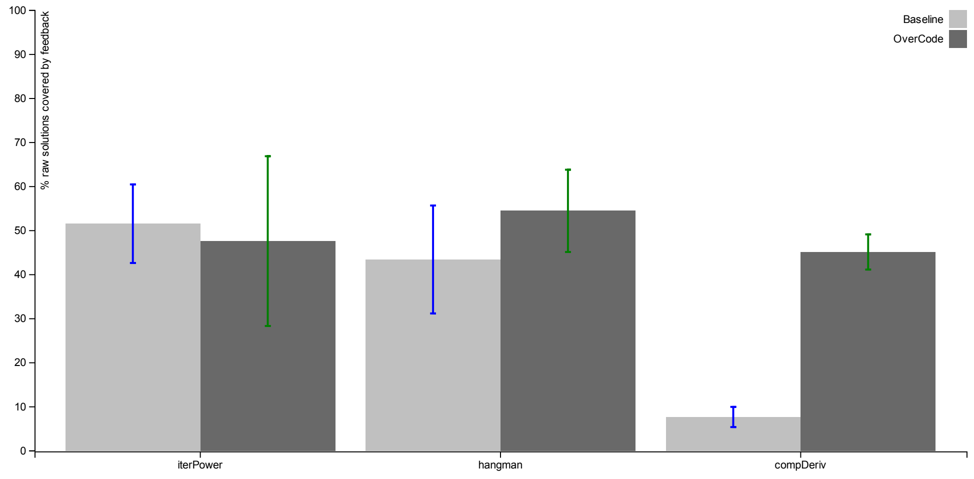
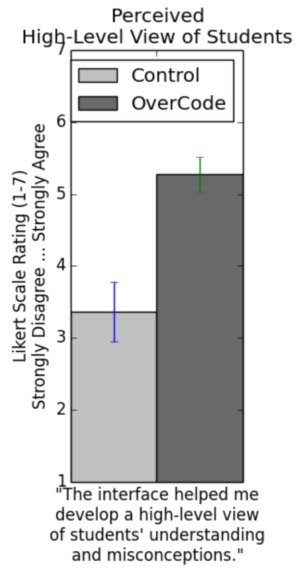
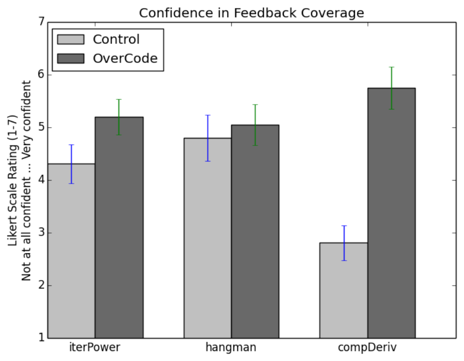


Fig. 19. Mean feedback coverage (percentage of raw solutions) per trial during the coverage study for each problem, in the OverCode and baseline interfaces.



(a)



(b)

Fig. 20. (a) Post-condition perception of coverage (excluding one participant) and (b) Confidence ratings that identified strategies frequently occurred (1-7 scale).

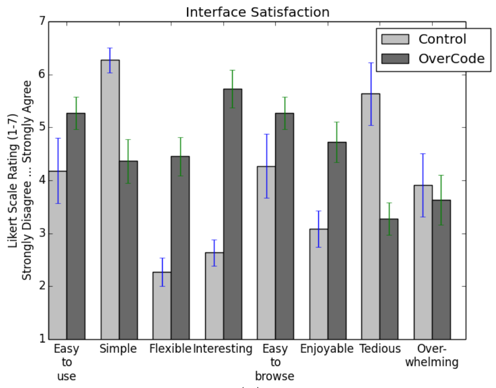
solutions for which the feedback was relevant. When giving feedback on the `iterPower` and `hangman` problems, there was not a statistically significant difference in the feedback coverage between interface conditions. However, on `compDeriv`, the problem with the most complex solutions, subjects using OverCode were able to achieve significantly more coverage of student solutions than when using the baseline interface (Mann–Whitney U = 0, n1 = n2 = 4, p< 0.05).

8.4.3. H4: Perceived coverage. Immediately after using each interface, we asked participants how strongly they agreed with the statement 'This interface helped me develop a high-level view of students' understanding and misconceptions,' which quotes the first part of our third hypothesis. Participants agreed with this statement after using OverCode significantly more than when using the baseline interface (W=63, Z=2.70,p<0.01,r=0.81). Statistical significance was computed using the Wilcoxon Signed Rank test, pairing users' ratings of each interface. The analysis was done on 11 participants' data, as one participant's data was lost. The mean rating (with standard error) for the responses is shown in Figure 20(a).

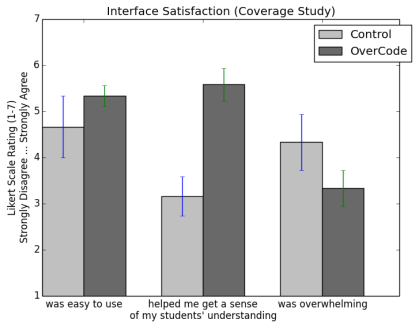
For each strategy identified by subjects, we asked them to rate their confidence, on a scale of 1-7, that the strategy was frequently used by students in the dataset. Mean confidence ratings on a per-problem basis are shown in Figure 20(b). We found that for `compDeriv`, subjects using OverCode were significantly more confident that their annotations were relevant to many students, compared to the baseline (Mann–Whitney U = 260.5, n1 = 18 n2 = 16, p< 0.0001).

8.4.4. H1: Interface satisfaction. Interface satisfaction was measured through multiple surveys, (1) immediately after using each interface and (2) after using both interfaces. Statistical significance was computed using the Wilcoxon Signed Rank test, pairing users' ratings of each interface.

Immediately after finishing the assigned tasks with an interface, participants rated their agreement with statements about the appropriateness of various adjectives to describe the interface they just used, on a 7-point Likert scale. While participants found the baseline to be significantly more simple (W=2.5, Z=-2.60,p<0.01,r=0.78), they found OverCode to be significantly more flexible (W=45, Z=2.84, p<0.005, r=0.86), less tedious (W=3.5, Z=-2.41, p<0.05, r=0.73), more interesting (W=66, Z=2.96, p<0.001, r=0.89), and more enjoyable (W=45, Z=2.83, p<0.005, r=0.85). The analysis was



(a)



(b)

Fig. 21. **H1: Interface satisfaction** Mean Likert scale ratings (with standard error) for OverCode and baseline, (a) immediately after using the interface for the Coverage Study task, and (b) after using both interfaces.

done on 11 participants' data, as one participant's data was lost. The mean ratings (with standard error) for the responses is shown in Figure 21.

After the completion of the Coverage Study, participants were asked again to rate their agreement with statements about each interface on a 7-point Likert scale. After using both interfaces to view thousands of solutions, there were no significant differences between how overwhelming or easy to use each interface was. However, participants did feel that OverCode "helped me get a sense of my students' understanding" more than the baseline (W=62.5, Z=-2.69, p<0.01, r=0.78). The mean ratings (with standard error) for the responses is shown in Figure 21.

8.4.5. Usage and Usefulness of Interface Features. In the second part of the post-study survey, participants rated their agreement with statements about the helpfulness of various interface features on a 7-point Likert scale. There were only two features to ask about in the baseline interface (in-browser find and viewing raw solutions), which were mixed in with statements about features in the OverCode interface. The OverCode feature of stacking equivalent solutions was found more helpful than the baseline's features of in-browser find (W=41, Z=2.07, p<0.05, r=0.60) and viewing raw students' solutions, comments included (W=45, Z=2.87,p<0.005,r=0.83). The OverCode feature of variable renaming and previewing rewrite rules were also both found significantly more helpful than seeing students' raw code (W=65.5,Z=2.09,p<0.05,r=0.61 and W=56.5, Z=2.14, p<0.05, r=0.62, respectively). The mean ratings for the features are shown in Figure 22.

In addition to logging `read` events, we also recorded usage of interface features, such as creating rewrite rules and filtering stacks. A common usage strategy was to read through the stacks linearly and mark them as `read`, starting with the largest reference stack, then rewrite trivial variations in expressions to merge smaller behaviorally-equivalent stacks into the largest stack. Stack filtering (Figure 4) was sometimes used to review solutions that contained a particularly conspicuous line (e.g. a recursive call to solve `iterPower`, or an extremely long expression). The filter's frequency slider (Figure 4a) and the variable legend (Figure 6b) were scarcely used.

All subjects wrote at least two rewrite rules, often causing stacks to merge that only differed in some trivial way, like reordering operands in multiplication statements (e.g. `result = result*base` vs. `result = base*result`). Some rewrite rules merged Python expressions that behaved similarly but differed in their verbosity (e.g. `for i in range(0, exp)` vs. `for i in range(exp)`) - variations that might be considered noteworthy or trivial by different teachers.

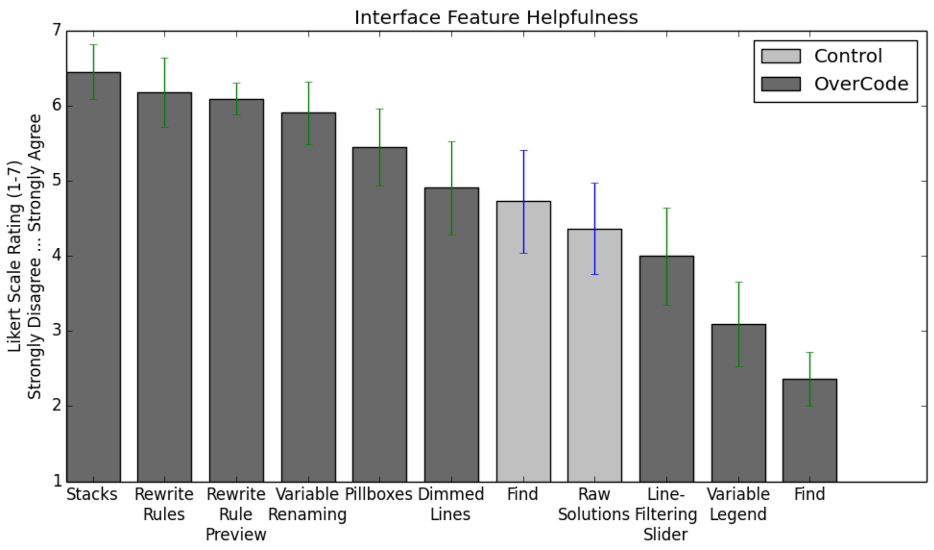


Fig. 22. Mean Likert scale ratings (with standard error) for the usefulness of features of OverCode and baseline. "Find" is the web browser's built-in find, which appears twice because users rated its usefulness for each condition.

9. DISCUSSION

Our three-part evaluation of OverCode's backend and user interface demonstrates its usability and usefulness in a realistic teaching scenario. Given that we focused on the first three weeks of an introductory Python programming course, our evaluation is limited to single functions, whose most common solutions were less than ten lines long. Some of these functions were recursive, while most were iterative. Variables were generally limited to booleans, integers, strings, and lists of these basic data types. All solutions had already passed the autograder tests, and study participants still found solutions that suggested students' misconceptions or knowledge gaps. Future work will address more complex algorithms and data types.

9.1. Read coverage

First, we observed that subjects were able to effectively read less in order to cover more ground (H1, read coverage). This is expected, because in OverCode each read stack represented tens or hundreds of raw solutions, while there was only a 1-to-1 mapping between read solutions and raw solutions in the baseline condition. OverCode's backend makes it possible to produce a single cleaned piece of code that represents many raw solutions, reducing the normally high cognitive load of processing all the raw solutions, with their variation in formatting and variable naming.

Figure 18(a) shows that in some cases, subjects were able to read nearly as many (`hangman`) or more (`iterPower`) function definitions in the baseline as in OverCode. In the case of `iterPower`, the raw solutions are repetitive because of the simplicity of the problem and the relatively small amounts of variation demonstrated by student solutions. This can explain the subjects' ability to move quickly through the set of solutions, reading as many as 90 solutions in 15 minutes.

Figure 18(b) shows the effective number of raw solutions read, when accounting for the number of solutions represented by each stack in the OverCode condition. In the case of (iterPower), subjects can say they have effectively read more than 30% of student solutions after reading the first stack. A similar statement can be made for `hangman`, where the largest stack represents roughly 10% of solutions. In the case of `compDeriv`, the small size of its largest stack (22 out of 1433 raw solutions) means that the curve is less steep, but the use of rewrite rules (avg. 4.5 rules written per `compDeriv` subject) enabled subjects to cover over 10x the solutions covered by subjects in the baseline condition.

9.2. Feedback coverage

We also found that subjects' feedback on solutions for the `compDeriv` problem had significantly higher coverage when produced using OverCode than with the baseline, but that this was not the case for the `iterPower` and `hangman` problems. `compDeriv` was a significantly more complicated problem than both `iterPower` and `hangman`, meaning that there was a greater amount of variation between student solutions. This high variation meant that any one piece of feedback might not be relevant to many raw solutions, unless it was produced after viewing the solution space as stacks and creating rewrite rules to simplify the space into larger, more representative stacks. Conversely, the simple nature of `iterPower` and `hangman` meant that there was less variation in student solutions. Therefore, regardless of whether the subject was using the OverCode or baseline condition, there was a higher likelihood that they would stumble across a solution that had frequently occurring lines of code, and the feedback coverage for these problems became comparable between the two problems.

9.3. Perceived coverage

In addition to the actual read and feedback coverage that subjects achieved, an important finding was that (i) subjects felt they had developed a better high-level understanding of student solutions and (ii) subjects stated they were more confident that identified strategies were frequent issues in the dataset. While a low self-reported confidence score did not necessarily correlate with low feedback coverage, these results suggest that OverCode enables the teacher to gauge the impact that their feedback will have.

10. FUTURE WORK

Currently, we have used OverCode to look at solutions from an introductory programming course. We have yet to explore how well it scales to increasingly complex code solutions. OverCode enabled higher feedback coverage on one of the two more complicated problems in the Coverage Study; applying the OverCode system to even more complicated code will demonstrate how well it scales up, and may also expose the need for new pipeline and interface features that be addressed. One foreseeable complication is the need to handle complex variable values. OverCode could also readily handle more programming languages. For a language to be displayable in OverCode, one would need (1) a logger of variable values inside tested functions, (2) a variable renamer and (3) a formatting standardization script.

OverCode could be integrated with the autograder that tests functions for input-output correctness. The execution could be performed once in such a way that it serves both systems, since both OverCode and many autograders require actually executing the code on specified test cases. If integrated into the autograder, users of OverCode could also give 'power feedback' by writing line- or stack-specific feedback to be sent back to students along with the input-output test results. This would require pipeline and user interface modifications, in order to account for the fact that not all solutions would be returning the same values for each test case.

OverCode may also provide benefit to students, not just the teaching staff, after students have solved the problem on their own, like Cody. However, the current interface may need to be adjusted for use by non-expert students, instead of teaching staff.

Our user study participants produced a variety of suggestions for additional features. In addition to those but unmentioned by users, variable renaming obscures pedagogically relevant information. The user-tested UI does not include access to raw solutions represented by a stack's cleaned code, or to the original variable names represented by a common variable name. This can be accomplished by adding tooltips and dropdown menus. This may also be part of better communicating to users that they are looking at cleaned, not raw, code.

When the program tracing, renaming, or reformatting scripts generate an error while processing a solution, we exclude the solution from our analysis. Less than five percent of solutions were excluded from each problem, but that can be reduced further by adding support for handling more special cases and language constructs to these library functions.

Also, our current backend analysis computes the set of common variables by comparing the sequence of values the variables take over a single test case, but this can be easily generalized and extended to work over multiple test cases. The definition of common variables would then be slightly modified to be those variables that take the same *set* of sequence of values.

11. CONCLUSION

We have designed the OverCode system for visualizing thousands of Python programming solutions to help teachers explore the variations among them. Unlike previous approaches, OverCode uses a lightweight static and dynamic analysis to generate stacks of similar solutions and uses variable renaming to present cleaned solutions for each stack in an interactive user interface. It allows teachers to filter stacks by line occurrence and to further merge different stacks by composing rewrite rules. Based on two user studies with 24 current and potential teaching assistants, we found OverCode allowed teachers to more quickly develop a high-level view of students' understanding and misconceptions, and provide feedback that is relevant to more students' solutions. We believe an information visualization approach is necessary for teachers to explore the variations among solutions at the scale of MOOCs, and OverCode is an important step towards that goal.

12. ACKNOWLEDGMENTS

This material is based, in part, upon work supported by the National Science Foundation Graduate Research Fellowship (grant 1122374), the Microsoft Research Fellowship, the Bose Foundation Fellowship, and by Quanta Computer as part of the Qmulus Project. Any opinions, findings, conclusions, or recommendations in this paper are the authors', and do not necessarily reflect the views of the sponsors.

REFERENCES

Sumit Basu, Chuck Jacobs, and Lucy Vanderwende. 2013. Powergrading: a Clustering Approach to Amplify Human Effort for Short Answer Grading. *TACL* 1 (2013), 391–402.

Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance (ICSM '98)*. IEEE Computer Society, Washington, DC, USA, 368–377.

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

Matheus Gaudencio, Ayla Dantas, and Dalton D.S. Guerrero. 2014. Can Computers Compare Student Code Solutions As Well As Teachers?. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. ACM, New York, NY, USA, 21–26.

Elena L. Glassman, Ned Gulley, and Robert C. Miller. 2013. Toward Facilitating Assistance to Students Attempting Engineering Design Problems. In *Proceedings of the Tenth Annual International Conference on International Computing Education Research (ICER '13)*. ACM, New York, NY, USA.

Philip J. Guo. 2013. 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)*. ACM, New York, NY, USA, 579–584.

Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas J. Guibas. 2013. Syntactic and Functional Variability of a Million Code Submissions in a Machine Learning MOOC. In *AIED Workshops*.

Andrew Luxton-Reilly, Paul Denny, Diana Kirk, Ewan Tempero, and Se-Young Yu. 2013. On the differences between correct student solutions. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education (ITiCSE '13)*. ACM, New York, NY, USA, 177–182.

F. Marton, A.B.M. Tsui, P.P.M. Chik, P.Y. Ko, and M.L. Lo. 2013. *Classroom Discourse and the Space of Learning*. Taylor & Francis.

Aditi Muralidharan and Marti Hearst. 2011. Wordseer: Exploring language use in literary text. *Fifth Workshop on Human-Computer Interaction and Information Retrieval* (2011).

Aditi Muralidharan and Marti A Hearst. 2013. Supporting exploratory text analysis in literature study. *Literary and linguistic computing* 28, 2 (2013), 283–295.

Aditi S. Muralidharan, Marti A. Hearst, and Christopher Fan. 2013. WordSeer: a knowledge synthesis environment for textual data. In *CIKM*. 2533–2536.

Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas J. Guibas. 2014. Codewebs: scalable homework search for massive open online programming courses. In *WWW*. 491–502.

Kelly Rivers and Kenneth R. Koedinger. 2013. Automatic Generation of Programming Feedback; A Data-Driven Approach. In *AIED Workshops*.

Jeffrey M. Rzeszotarski and Aniket Kittur. 2012. CrowdScape: interactively visualizing user behavior and output. In *UIST*. 55–62.

Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 76–85.

Dennis Shasha, JT-L Wang, Kaizhong Zhang, and Frank Y Shih. 1994. Exact and approximate algorithms for unordered tree matching. *IEEE Transactions on Systems, Man and Cybernetics* 24, 4 (1994), 668–678.

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

Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Trans. Softw. Eng.* 10, 5 (Sept. 1984), 595–609.

Ahmad Taherkhani, Ari Korhonen, and Lauri Malmi. 2012. Automatic recognition of students' sorting algorithm implementations in a data structures and algorithms course. In *Proceedings of the 12th Koli Calling International Conference on Computing Education Research*. ACM, 83–92.

Ahmad Taherkhani and Lauri Malmi. 2013. Beacon-and Schema-Based Method for Recognizing Algorithms from Students' Source Code. *JEDM-Journal of Educational Data Mining* 5, 2 (2013), 69–101.

Songwen Xu and Yam San Chee. 2003. Transformation-Based Diagnosis of Student Programs for Programming Tutoring Systems. *IEEE Trans. Softw. Eng.* 29, 4 (2003), 360–384.

13. AUTHOR STATEMENT ABOUT PRIOR PUBLICATIONS

Two work-in-progress abstracts by the same authors have appeared previously, both titled "Feature engineering for clustering student solutions." One abstract appeared in the Learning at Scale 2014 poster track, and the other in the CHI 2014 Learning Innovations at Scale workshop. Those abstracts used the same dataset of Python solutions used in this submission.

A third abstract appeared in the User Interface Software and Technology (UIST) poster track, which gave a very high-level overview of the OverCode user interface. Otherwise, virtually all of the current submission is new work which was not discussed in those abstracts: in-depth descriptions of the OverCode visualization, its implementation, and the empirical evaluations.